SAS® Business Orchestration Services 1.2: User's Guide

SAS® Documentation
January 17, 2019

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2019. *SAS® Business Orchestration Services 1.2: User's Guide*. Cary, NC: SAS Institute Inc.

SAS® Business Orchestration Services 1.2: User's Guide

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

January 2019

P1:eopug

Contents

1

Getting Started with SAS Business Orchestration Services

Overview

SAS Business Orchestration Services is a highly customizable platform for managing transaction flow and transformation to and from a decision engine, an alert triage or case system, or both. It works with real-time and batch transactions. Here are the primary functions when managing the transaction flow.

- Transformation: Conversion of a message into a format that can be consumed by the decision engine or another client system.
- Enrichment: Adding information to the message that is received from another data source.
- Endpoint: Sending the incoming message or messages that have been transformed and enriched to a decision engine or client system.
- Store and forward: Queueing messages if the endpoint is unavailable to receive messages and sending the messages once the endpoint is available.
- Stand-in response: Sending the last known good response for a message if the endpoint does not respond within a specified amount of time.

SAS Business Orchestration Services is a lightweight Java process that can be installed on the same server as the SAS OnDemand Scoring Engine or on a separate server. Run at least two instances of SAS Business Orchestration Services on separate servers to ensure high availability.

You are not required to change your client systems message layouts. SAS Business Orchestration Services receives messages in the native format and then transforms the messages as required by the endpoint. Your system must have the ability to send a request, receive a response, or both.

SAS Business Orchestration Services uses the SAS Fraud Management API library to communicate with SAS OnDemand Decision Engine. It is easier to understand and use SAS Business Orchestration Services if you already understand SAS Fraud Management.

Install SAS Business Orchestration Services

Introduction

You can install SAS Business Orchestration Services from one of the following files:

> **TIP** For a production environment, you should use the eop-<*version*>.x86_64.rpm package.

- eop-<*version*>.x86_64.rpm
- eop_<*version*>.tar.gz

> **TIP** If you do not have the installation file, then contact your SAS Fraud Management support personnel. For more information about configuring and installing on multiple nodes, see "SAS Business Orchestration Services Scalability and High Availability" on page 43.

eop-<*version*>.x86_64.rpm Package

To install SAS Business Orchestration Services using the eop-<*version*>.x86_64.rpm package, complete these steps:

Note: By default, SAS Business Orchestration Services installation is owned by the saseop user. If you want a different user to own it, then contact your SAS Fraud Management support personnel or change the SAS Business Orchestration Services file ownership after installation.

1. Create the Linux user ID named saseop.
2. With root privileges and as a group member of sasfraud, run the following command:

   ```
   rpm -ivh eop-<version>.x86_64.rpm
   ```

   To add a new user and a group:

   a. If the sasfraud group does not exist, run:

      ```
      sudo groupadd sasfraud
      ```

   b. Create user saseop and make that user a member of the sasfraud group:

      ```
      sudo useradd -G sasfraud saseop
      ```

   c. Set a password for user saseop:

      ```
      sudo passwd saseop
      ```

eop_<*version*>.tar.gz Package

To install SAS Business Orchestration Services using the eop_<*version*>.tar.gz package, complete these steps:

1 In your target deployment environment, change to the directory where you want to install SAS Business Orchestration Services.

2 Run the following:

```
tar -zxvf eop_<version>.tar.gz
```

The following folder and subfolders are extracted:

```
eop_<version>
   logs
   lib
   config
   Readme.txt
   bin
```

For more information about configuring and installing SAS Business Orchestration Services on multiple nodes, see "SAS Business Orchestration Services Scalability and High Availability" on page 43.

Environment Setup

SAS Business Orchestration Services requires JDK 1.8. Set the environment variable JAVA_HOME to point to your JDK installation directory. Add the Java interpreter to your PATH environment variable. The exact directory might vary from system to system. Check your local file system to be sure where Java is installed.

```
UNIX (bash/sh):
JAVA_HOME=/usr/local/java/jdk1.8.0_102; export JAVA_HOME
PATH=$JAVA_HOME/bin:$PATH; export PATH
UNIX (tcsh):
setenv JAVA_HOME=/usr/local/java/jdk1.8.0_102
setenv PATH $JAVA_HOME/bin:$PATH
```

Start SAS Business Orchestration Services

Once you have installed SAS Business Orchestration Services and have Java set up, you can start SAS Business Orchestration Services.

```
cd eop_<version>/bin
 ./eop.sh start
```

You can also tail the log to console during the SAS Business Orchestration Services start up by running the following command:

```
./eop.sh start tail
```

Verify That SAS Business Orchestration Services Is Running

Verify that SAS Business Orchestration Services is running using one of the following commands:

```
./eop.sh status
curl localhost:9090/ping; echo
```

A Pong message confirms it is running.

Note: The previous command assumes the default port 9090. For more information, see "Configure SAS Business Orchestration Services" on page 5. See the SAS Business Orchestration Services management console for status.

Stop SAS Business Orchestration Services

Stop SAS Business Orchestration Services using the following command:

```
./eop.sh stop
```

One of the following messages is displayed:

```
EOP stop
EOP Pid: 85344. Stopping it ...
EOP stopped.
```

```
EOP stop
EOP is not running.
```

Uninstall SAS Business Orchestration Services

To uninstall SAS Business Orchestration Services, complete these steps:

1. Stop SAS Business Orchestration Services if it is running.

   ```
   ./eop.sh stop
   ```

2. You uninstall the SAS Business Orchestration Services based on which package you installed.

   a. If the installation was done using the eop-<*version*>.x86_64.rpm package, then issue (with root privileges) the following command:

   ```
   rpm -evv <eop_rpm_pkg_name>
   ```

   TIP You can find the <*eop_rpm_pkg_name*> by issuing the following command: `rpm -qa |grep eop`

   b. If the installation was done using the eop_<*version*>.tar.gz package, then issue the following command:

   ```
   rm -rf <path_to_EOP>/eop_<version>
   ```

TIP If you have made any customizations to the **<*path_to_EOP*>/eop/config** folder, you should back it up.

CAUTION! This command deletes all of the executable files and configuration.

Configure SAS Business Orchestration Services

Configuration Overview

You can configure the following operations by updating the configuration file with minimal or no Java source code changes.

- communication between the SAS Business Orchestration Services and SAS Fraud Management
- logging
- monitoring
- performance tuning and scalability
- request data mapping, enrichment, and validation
- response data mapping
- services to expose
- stand-in response
- store and forward

Configure Endpoint/Service

SAS Business Orchestration Services is written in Java and it uses an open-source Spring Framework and Apache Camel. Both solutions are modular and lightweight, enabling you to use only those components that are needed without bringing in the rest. You should understand how to configure Spring Framework using XML to use SAS Business Orchestration Services.

Sample REST Services

In "Verify That SAS Business Orchestration Services Is Running" on page 4, you accessed a Ping service on SAS Business Orchestration Services.

```
curl localhost:9090/ping; echo
```

The Ping service is defined at the beginning of the camel-context-simple.xml file.

```
<rest>
   <get uri="/ping">
      <route id="Ping">
         <transform><constant>pong</constant></transform>
      </route>
   </get>
</rest>
```

This service does not do much. It exposes an HTTP GET and always returns a Pong message when invoked. However, it gives a quick indication that SAS Business Orchestration Services is reachable and running.

For a more interesting REST service configuration, see the `pingode` example in the same file:

```
<get uri="/pingode">
   <route id="Ping_ODE">
      <bean id="Ping_ODE_Transaction" ref="inboundMapper" method="toPingOdeTransaction"/>
      <to id="ping_ode_Transaction" uri="disruptor:sendTransactionToODE?timeout={{sla_timeout}}"/>
      <bean id="Out bound Mapper" ref="outboundMapper" />
   </route>
</get>
```

When /pingode is accessed, the route named Ping_ODE is used to process incoming requests. In this case, no input parameter is expected.

The route completes these steps:

1. Invoke the Spring bean named `inboundMapper` to generate a SAS Fraud Management Ping request (A special transaction request).
2. Send the Ping request to a child route named `sendTransactionToODE`.
3. Invoke the Spring bean named `outboundMapper` for processing the child route's response. Send the response to the caller.

More Transport Support

By using pluggable Camel components, SAS Business Orchestration Services can be configured to use all of the transports that are supported by Camel. For a complete list of transports, see Apache Camel Transport Components.

With these components readily available, SAS Business Orchestration Services can easily make endpoint/services accessible. It can also access other services. SAS Business Orchestration Services can be configured to do any of the following tasks:

- consume messages from a message queue
- expose REST API to process request data
- open sockets for receiving request data
- process batch files containing request data

To learn more about these examples, see **`eop_version/config/spring/camel-context.xml`**.

Configure Tokenization Service

In some deployment environments where transaction data is transmitted across the internet, stored in a public cloud, or both, an extra step is needed to tokenize this transaction data for safety. Before you can enable tokenization service, at least one SafeNet KeySecure server or a cluster of SafeNet KeySecure servers need to be installed and started. KeySecure servers can be deployed either on-premises, in a public cloud, or both. For more information about SafeNet KeySecure servers, see the Gemalto documentation.

To configure tokenization service in the SAS OnDemand Decision Engine, complete these steps:

1. Modify the NAE_IP.1, NAE_Port, and Protocol properties in the **`IngrianNAE.properties`** file. This file can be found in the **`<eop_install_dir>/lib/`** directory.
2. Make sure that the ol-tokenization.xml bean configuration file has been imported. Uncomment the following line in **`camel-context-simple.xml`**. It is commented out by default since there is no default SafeNet KeySecure server.

```
<!-- Uncomment the next line to init TokenService on start -->
<process id="tokenization" ref="tokenService"/>
```

3 Define one or more TransactionTokenizer beans in **`ol-tokenization.xml`**.

```
<bean id="TxnTokenizer_01"
            class="com.sas.finance.fraud.ol.tokenization.TransactionTokenizer">
<constructor-arg name="tokenService" ref="tokenService"/>
<constructor-arg name="items">
   <map>
      <entry key="AQO_ACCT_NUM" value-ref="TS_L4"/>
   </map>
</constructor-arg>
</bean>
```

The bean definition can take a map of a SAS Fraud Management field name to TokenSpec pairs, enabling you to specify one or more fields to be tokenized. Each field can be tokenized according to a different TokenSpec. The following TokenSpec formats are available for tokenization:

> **TIP** Tokenization can be applied to strings or digits from 1 to 128 characters.

- the whole string
- everything but the last 4 digits
- everything but the first 6 digits
- everything but the first 2 and last 4 digits
- everything but the first 6 and last 4 digits

4 Use TransactionTokenizer in the transaction processing route.

```
<!-- Uncomment this line if you need to tokenize the transaction. -->
<bean id="tokenize_card_numbers" ref="TxnTokenizer_01"/>
```

2

Transforming Messages

Inbound Data Transformation

Inbound Data Transformation Overview

This chapter explains how customer transaction data can be mapped before it is routed to SAS Fraud Management.

One of the main functions of SAS Business Orchestration Services is to receive customer transaction requests in native customer data formats, process the requests, and return responses in customer-preferred formats. This customer-centric approach aims to introduce minimal code change to a customer's existing software, shorten the integration time, and increase speed to market.

SAS Fraud Management requires messages to conform to the SAS Fraud Management specific data format. Incoming transactions from a client's system are in a different data format from the SAS Fraud Management data format. Therefore, inbound data mapping is needed to convert a client's transaction into a SAS Fraud Management transaction.

There is a wide range of messaging standards that are used by financial institutions and for data exchange among institutions. Even with the emergence of international standards such as FIX, ISO20022, SWIFT, and FpML, many institutions still use their proprietary syntaxes and semantics. As a result, SAS Business Orchestration Services needs to handle inbound data transformation to accommodate customer-specific formats and standard formats.

SAS Business Orchestration Services supports the following common formats:

- Byte streams
- CSV
- JSON
- XML
- ISO8583
- SWIFT MT

Mapping takes place in two steps:

1. Map input data into request field name-value pairs.
2. Map input request field names to SAS Fraud Management field names.

Mapping for XML Input

Many custom data formats and even some international standard formats such as ISO20022 use an XML format. XML data is self-descriptive. SAS Business Orchestration Services uses XPath and a string value represented by XPath to form name-value pairs. SAS Business Orchestration Services uses a subset of XPath functions to get name-value pairs from the input XML data.

To map the name-value pairs to a SAS Fraud Management transaction, SAS Business Orchestration Services needs additional information:

- Which part of the XML data maps to which SAS Fraud Management transaction field or fields.
- What are the default values for some of the SAS Fraud Management transaction fields.
- How to transform XML data (if needed).
- How to enrich data (if needed).

Sample mapping files can be found in **`eop_version/config/mappings/inbound/mapping_iso20022.xml`**.

Here is part of this file.

```
<application  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="https://www.sas.com/sfm/eop"
    xsi:schemaLocation="http://www.sas.com/sfm/eop mapping.xsd">

    <messages name="ISO 20022">
       <message messageType="PAIN" subType="001.001.03" mapperId=”iso20022_01”>
          <fields>
             <field name="TBT_GROUP_CNT">
                <srcField name="CstmrCdtTrfInitn/GrpHdr/NbOfTxs"/>
             </field>
```

```
            <field name="TBT_GROUP_AMT">
               <srcField name=" CstmrCdtTrfInitn/GrpHdr/CtrlSum"/>
            </field>
             ...
            <field name="smh_tran_type" default="TRX"/>
            <field name="rqo_tran_time_alt" default="17:14:00"/>
          </fields>
    </message>
</application>
```

In this file, it is important to understand the following concepts:

- A piece of data identified by an XML pathname is mapped to a SAS Fraud Management field.
- A SAS Fraud Management field can be assigned a default value even when there is no corresponding input.
- Mapping files can be version controlled by using name, type, and subtype combinations.
- A unique mapperId can also be used to identify a mapping.

The initial mapping is usually done by the SAS Fraud Management modeling team or business consultants. With the initial mapping created and the above understanding of the mapping file, you should have no problem making small changes, such as the following:

- adding a new field mapping
- changing default values
- deleting an existing field mapping
- modifying an existing field mapping

The mapping files are monitored by SAS Business Orchestration Services at run time. Changes are picked up and applied at run time. Any mapping errors are written to log files.

Mapping for JSON Input

JSON is another popular data format that might be used. Like XML, JSON is also self-describing, hierarchical, and easy to parse. JSON's popularity results from being more compact than XML.

The JSON hierarchical names are used for mapping an input data field to a SAS Fraud Management field.

The mapping file for JSON input is similar to what is used for XML input. In fact, if the JSON input has the same hierarchical names as that of XML input, then the mapping file is identical.

Mapping for CSV Input

For parsing input data in CSV format (using separators such as a comma, tab, tilde, or any other character string), a corresponding comma separated names (CSN) file needs to be configured in the SAS Business Orchestration Services. A CSN file tells SAS Business Orchestration Services about each field name in the incoming CSV input. CSN files need to be placed in the ***<eop_install_path>/config/mappings/csv/*** directory. You can also find sample CSN files in that directory.

The input CSV data is parsed using a CSN file to produce a field name to value pairs. If you are already familiar with the sample mapping file used in "Mapping for XML Input" on page 10, then you see that the names are used in `<srcField name="xxx"/>`. In the mapping for XML and JSON input, the XPath notation is used for the `srcField` name, but it is not a requirement imposed by the mapping file. Simple names work equally well in this case.

You need to create the mapping file with correct `<srcField name="xxx"/>` mapped to the SAS Fraud Management `<field name="xxx"/>`.

Here is an example of a CSN:

```
# This sample CSV mapping information contains a line for unique key for CSV formats, and
# another line for comma separated names.
# This is the CSV format for a ping request.
# The input CSV data could be: 04.03.08,1,CMD
# although the separator does not have to be a comma, it could be a ~ for example.
sample1
version,needResponse,tran type
```

For the sample CSN, here is an example corresponding mapping file:

```
<?xml version="1.0"?>
<!-- Sample mapping equivalent to a ping request. -->
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="https://www.sas.com/sfm/eop"
   xsi:schemaLocation="http://www.sas.com/sfm/eop mapping.xsd">

   <messages name="Ping">
      <message messageType="ping" subType="1.0" mapperId="ping">
         <fields>
            <field name="smh_msg_version" default="04.04.02">
               <srcField name="version"/>
            </field>

            <field name="smh_tran_type">
               <srcField name="tran type"/>
            </field>

            <field name="scc_cmd_code" default="0"/>

            <field name="smh_resp_req">
               <srcField name="needResponse" default="0"/>
            </field>
         </fields>
      </message>
  </messages>
</application>
```

If you are not familiar with what input field names should be mapped to the SAS Fraud Management field names, then contact your SAS Fraud Management support personnel.

Mapping a Byte Stream

Another commonly used input format is a binary data stream that can be converted into a text string. Unlike CSV data, this text string contains no information to indicate where each input field starts or ends. A separate schema file (or equivalent) is needed to describe how the data needs to be interpreted based on column positions.

By default, SAS Business Orchestration Services uses a simple file format to provide a schema. If you want to use a different format, then contact your SAS Fraud Management support personnel. Here is a sample schema file, which can be found at **<*eop_version*>/config/mappings/fbs/sample.fbs**.

```
# This file contains a fixed block schema for
# processing fixed block stream data.
# fbsId must be specified first, and it needs to be unique.
# Each additional line describes a field, and it has three parts:
#      Field name
#      Field start position
#      Field length
fbsId=simpleping
```

```
version, 0, 8
tran type, 8, 3
needResponse, 11, 1
```

Here is a sample input text string:

```
04.04.02CMD1
```

Using the previous sample schema, the parsing result is:

```
version = "04.04.02"
tran type = "CMD"
needResponse = "1"
```

Once the name-value pairs are produced, the rest of the processing is similar to what has been discussed in previous sections.

Mapping for Other Data Formats

Other data formats can be supported by SAS Business Orchestration Services with small code changes. Typically, you need to create a new Camel processor.

The new Camel processor's main function is to convert input data into name-value pairs. Once the name-value pairs are created, the rest of the processing is similar to what has been discussed in previous sections.

Data Transformation

Previous sections described how different types of input data can be mapped to SAS Fraud Management transactions. This section describes how each request field or fields can be mapped to a SAS Fraud Management field or fields with or without data transformation. Mappings from the sample mapping files are used in the examples in this section.

Direct Mapping

Direct mapping is the easiest case where the request field value can be directly used or mapped to a SAS Fraud Management field value. The format of the mapping is to put the name of the SAS Fraud Management field name first and assign it a value from a field in the incoming transaction. Here is direct mapping sample:

```
<field name="rua_4byte_string_002">
   <srcField name="FTSTOLIQ_TRN_SUF"/>
</field>
```

Mapping By Transforming One Request Field

In the following example, the request field value must be parsed before it is assigned to a SAS Fraud Management field. SAS Business Orchestration Services allows users to use a Groovy script to transform a value before it is assigned to a SAS Fraud Management field.

```
73          <field name="rua_ind_004" default="U">
74             <srcField name="FTSTOLIQ_STATUS">
75                <groovy>
76                   return value.substring(0, 1)
77                </groovy>
78             <srcField>
79          </field>
```

Lines 73 through 79 show that only the first character from FTSTOLIQ_STATUS should be assigned to rua_ind_004.

```
83 <-- Incoming request value format: 160916 (YYMMDD), need transform to 20160916 -->
84          <field name="rua_8byte_string_001">
85             <srcField name="FTSTOLIQ_VAL_DATE">
86                <groovy>
87                   return '20' + value
88                </groovy>
89             <srcField>
90          </field>
```

Lines 84 through 90 show that the value for FTSTOLIQ_VAL_DATE should be prefixed with 20 before it is assigned to the rua_8byte_string_001 field.

In the above mapping example, the variable named `value` represents the run-time value for the field name given in the `srcField` tag.

The run-time Groovy method has the following signature:

```
def getValue(String value, Map<String, String> records, Map<String, Object> cache)
```

In most cases, simple string methods should be sufficient for converting the passed in value. However, SAS Business Orchestration Services does not restrict you from using more advanced Groovy functions. How you use the other parameters, records, and cache are discussed in later sections.

Mapping Involving Multiple Request Fields

In some cases, the value that is assigned to a SAS Fraud Management field might depend on multiple input request fields. The following shows a multiple input request:

```
160          <field name="rua_10byte_string_002">
161            <srcField name="FTSTOLIQ_DR_FED_ABA">
162               <groovy>
163                  if (records['FTSTOLIQ_DR_ACCT_TYPE'] == 'M')
164                     return value
165                  else
166                      return ''
167               </groovy>
168            <srcField>
169         </field>
```

In this example, the rua_10byte_string_002 field depends on FTSTOLIQ_DR_FED_ABA and FTSTOLIQ_DR_ACCT_TYPE.

If the request field FTSTOLIQ_DR_ACCT_TYPE has a value of M, then the value for field FTSTOLIQ_DR_FED_ABA is assigned to the SAS Fraud Management field named rua_10byte_string_002. If it has another value, then it assigns an empty string value to the SAS Fraud Management field named rua_10byte_string_002.

The use of the passed in `Map<String, String>` records is from the Groovy method signature:

```
def getValue(String value, Map<String, String> records, Map<String, Object> cache)
```

In this example, two request field names are used. The records map contains complete name-value pairs from the input data, so all request fields are accessible. It is even possible to modify the map, although that is not recommended.

If you need to modify the map to store some intermediate processing results, then you should use the cache map, which is discussed in a later section.

Mapping One Request Field to Multiple SAS Fraud Management Fields

It is possible that one request field might need to be mapped to multiple SAS Fraud Management fields. SAS Business Orchestration Services permits a `srcField` with the same name to show up multiple times.

In the following example, the first 35 characters of FTSTOLIQ_DR_PARTY are assigned to the dua_40byte_string_004 field. The next 35 characters are assigned to the rua_30byte_string_001 field.

```
<-- Three SAS Fraud Management fields with values come from the
same source field have to be defined in 3 field blocks -->
         <field name="dua_40byte_string_004">
            <srcField name="FTSTOLIQ_DR_PARTY">
               <groovy>
                  return value.substring(0, 35)
               </groovy>
            <srcField>
         </field>

         <field name="rua_30byte_string_001">
            <srcField name="FTSTOLIQ_DR_PARTY">
               <groovy>
                  return value.substring(35, 65)
               </groovy>
            <srcField>
         </field>
```

Mapping User Variable Segments

You can define user variable segments in SAS Fraud Management. For more information about how to define and deploy user variable segments, see *SAS Fraud Management: SAS Rules Studio User's Guide*. SAS Business Orchestration Services supports user variable segments, and allows for the mapping of incoming message field or fields in user variable segments.

Once user variable segments have been defined and deployed in SAS Fraud Management, an addendum file named <*messageAPIVersionNumber*>-addendum.xml is generated. You need to put this addendum file in the ***<EOP_install_dir>/config/*** directory so that SAS Business Orchestration Services becomes aware of the custom segments and fields.

The following example shows what an entry in the addendum file looks like. In the SAS Business Orchestration Services mapping file, you can use either the field name or the alias. Most users prefer the alias, since it is more user friendly.

```
<field name="i00_str_00000_008" offset="32" format="$CHAR8." buildId="50106">
   <alias>i00_priority_code</alias>
   <description>description messages</description>
</field>
```

Data Validation in Mapping

SAS Business Orchestration Services enables you to insert validation in various stages of request processing. The mapping file is a convenient place to insert some validation rules.

In the following example, field FTSTOLIQ_TRN_DATE is expected to have a length of 6. When this is not true, the request processing throws an exception, and it is handled by the exception logic to send the proper response back to the caller.

```
52 <-- Incoming request value format: 160916 (YYMMDD), need transform to 20160916 -->
53 <-- And simple validation -->
54          <field name="rqo_tran_date">
55             <srcField name="FTSTOLIQ_TRN_DATE">
56                <groovy>
57                   if (value == null || value.length() != 6) {
58                      throw new Exception("Invalid input data for TRN_DATE! " + value)
59                    }
60                    return '20' + value
61                </groovy>
62             <srcField>
63          </field>
```

Be aware that this validation is for the run-time field value only. Do not confuse it with the validation of the mapping file itself. The XSD for the mapping file is available, but it is for a different purpose.

Using Mapping Files in a Route

Previous sections talked about mapping files for different request input formats. This section shows how those mapping files can be used.

Using RequestDispatcher

The `RequestDispatcher` bean is available in SAS Business Orchestration Services out of the box. It handles data formats that are discussed in previous sections, and can even be expanded to handle other formats by registering other handlers using the `setHandlers(Map<String, Consumer<Exchange> handlers)` method.

In the following example, line 79 exposes a REST URI, and lines 74 through 78 shows example usages. The request data is sent via an HTTP post. The `mapperId` in the URI tells the `requestDispatcher` which mapping should be used.

```
73 <-- An simple route using RequestDispatcher. You can invoke like: -->
74 <-- curl -X POST "localhost:9090/transaction/xml?mapperId=iso20022_01&startingTag=CstmrCdtTrfInitn" -->
75 <-- curl -X POST "localhost:9090/transaction/json?mapperId=iso20022_01&startingTag=CstmrCdtTrfInitn" -->
76 <-- curl -X POST "localhost:9090/transaction/csv?csvId=sample1&mapperId=ping&delimiter=," -->
77 <-- By default, delimiter is comma  -->
78 <-- curl -X POST "localhost:9090/transaction/bny?mapperId=bny2"  -->
79          <post uri="/transaction/{format}">
80             <route id="generalRoute" startupOrder="55">
81                <convertBodyTo type="java.lang.String"/>
82                <bean ref="odeUtils" method="decodeWebString"/>
83                <to uri="direct:requestDispatcher"/>
84                <bean ref="outboundMapper"/>
85             </route>
86          </post>
87       </rest>
88
89       <route id="Dispatcher" startupOrder="33">
90          <from uri="direct:requestDispatcher"/>
91          <doTry>
92             <-- create Txn object -->
93             <bean ref="requestDispatcher" method="process"/>
94             <bean id="StoreUnqKeyFor_format" ref="uniqueKeySaver"/>
95             <to uri="disruptor:sendTransactionToODE?timeout=100&waitForTaskToComplete=Always"/>
96             <doCatch id="Catch Txn Processing Error(format)">
97                <-- Request parsing error, validation error come here. -->
```

```
98  <exception>com.sas.finance.fraud.ol.exceptions.TransactionProcessingException</exception>
99              </doCatch>
100             <doCatch id="Catch Custom Timeout(format)">
101 <exception>com.sas.finance.fraud.ol.exceptions.EnrichTimeoutException</exception>
102 <exception>org.apache.camel.ExchangeTimedOutException</exception>
103                <to id="Stand-in-Response(format)" uri="direct:standInResponse"/>
104             </doCatch>
105             <doCatch id="Catch SendToODE Error(format)">
106                <exception>java.net.ConnectException</exception>
107                <exception>java.lang.RuntimeException</exception>
108                   <to id="SendErrorHandler(format)" uri="direct:handleSendError"/>
109             </doCatch>
110          </doTry>
111       </route>
```

Using Custom Processors

If the processors that are available in SAS Business Orchestration Services do not fit your needs, then you can create custom processors to handle your requests. In this case, you have more coding work to do, but it gives you more freedom in implementing your data handling logic.

Each mapping is represented by an XmlMapper object at run time, and XmlMapper can be retrieved using the following API based on the `mapperId` that you defined in your mapping XML file.

```
IBXmlMappingsRepository.getInstance.getXmlMapperByMapperId(String mapperId)
```

Data Enrichment

Data Enrichment Overview

This section discusses data enrichment by retrieving additional data from sources other than input requests. “Data Transformation” on page 13 covers different scenarios for converting request data fields to SAS Fraud Management fields. If your data handling involves only request data, not retrieving data from other sources, see “Data Transformation” on page 13.

Data enrichment can be done using one or both approaches:

- using a mapping file
- using a Camel processor or processors

The following sections detail both approaches.

Data Enrichment Using a Mapping File

Overview

The inbound mapping file is a convenient place to configure data enrichment. SAS Business Orchestration Services supports the following resources. It can also be extended to support more resource types.

- relational database management systems (RDBMS)
- Redis servers
- free text mining

Data Enrichment Using an RDBMS

You might store additional information that is retrieved to enrich a transaction request in a relational database. Enrichment data are typically stored in tables, and can be queried based on some key value or values from an input request.

The following configuration uses a database query result to assign a SAS Fraud Management field named rua_8byte_string_003.

```
225  <-- dbField to enrich from database -->
226  <-- a Map of request NV pairs and jdbcTemplate are accessible inside Groovy block. -->
227  <-- rscName is mandatory which identifies a database -->
228  <-- name and default attributes are optional. If the name is given, then its value can be
229  <-- conveniently accessed inside the Groovy block by a variable named "value" -->
230  <-- There are two ways to store query results for use in subsequent field mappings, to minimize the
number of database accesses
231      1. Using the passed in "records" Map<String,String>, which has NV pairs from transaction request.
232         a. to update existing fields
233         b. to insert new name-value
234      2. Using the passed in "cache" Map<String, Object>, which is for data sharing purpose.
235         For example,
236            cache.put('key1', 'v1')
237            cache.get('key3')
238  -->
239  <field name="rua_8byte_string_003">
240     <dbField rscName="derby" name="Request.Field.Name" default="defaultValue">
241        <groovy>
242           <![CDATA[
243              String sql = 'select value from enrichments where unqKey=\'00000000000000001038\'
                  and fieldname=\'rua_8byte_string_003\'';
244           ]]>
245        </groovy>
246     </dbField>
247  </field>
```

Line 240 says that the database named derby is being used, and it is configured in **`eop_version/config/spring/dataSource.xml`**.

Line 244 uses the passed in `jdbcTemplate` parameter to run the query. The `jdbcTemplate` parameter comes from Spring Framework.

The name attribute in line 240 is optional. When it is specified, it contains a run-time value for the request field identified by name.

The `dataSourceRepo` bean can take multiple entries for its constructor argument sources. This means that multiple databases can be configured and used at the same time for data enrichment purposes.

```
59  <-- This bean is used by the  mapping file, dbFields. rscName references the key value here. -->
60  <bean id="dataSourceRepo" class="com.sas.finance.fraud.ol.enrich.DatasourceRepository">
61     <constructor-arg name="sources">
62        <map>
63           <entry key="derby" value-ref="dataSource"/>
64        </map>
65     </constructor-arg>
66  </bean>
```

Here is the exact Groovy method signature for RDBMS data enrichment:

```
def getValue(JdbcTemplate jdbcTemplate, String value, Map<String,
String> records, Map<String, Object> cache)
```

Data Enrichment Using a Redis Server

You might store additional information that is retrieved to enrich a transaction request in a Redis server. Enrichment data is stored in a name-value cache, and it can be queried based on some key value or values from an input request.

Redis provides a NoSQL storage alternative to a classical RDBMS for horizontal scalability and speed. In terms of implementation, key-value stores represent one of the largest (and oldest) members in the NoSQL space.

The following configuration uses a Redis server query result to assign a SAS Fraud Management field named rua_8byte_string_002.

```
106  <-- redisField to show enrich from Redis -->
107  <-- a Map of request NV pairs and Spring RedisTemplate<?, ?> are accessible inside Groovy block. -->
108  <-- rscName is mandatory which identifies a Redis server to use. -->
109  <-- name and default attributes are optional. If name is given, then its value can be
110  conveniently accessed inside the Groovy block by a variable named "value" -->
111  <-- There are two ways to store query results for use in subsequent field mappings,
to minimize number of Redis accesses:
112      1. Using the passed in "records" Map<String,String>, which has NV pairs from transaction request.
113         a. to update existing fields
114         b. to insert new name-value
115      2. Using the passed in "cache" Map<String, Object>, which is for data sharing purpose.
116         For example,
117            cache.put('key1', 'v1')
118            cache.get('key3', new Double(100.001)))  -->
119  <field name="rua_8byte_string_002">
120     <redisField rscName="server1" name="FTSTOLIQ_ORIG_DATE" default="20170203">
121        <groovy>
122           return redisTemplate.opsForValue().get('myKey');
123        </groovy>
124     </redisField>
125  </field>
```

Line 120 shows that an `rscName` of server1 is being used.

The `redisTemplate` at line 122 is an instance of `RedisTemplate` from the Spring Data Redis package, which is an API for accessing Redis. Beware that the configuration of the Spring bean `stringRedisTemplate` can point to a Redis server cluster. This enables scalability and high availability. In addition, Spring Data Redis supports both Jedis and Lettuce as underlying libraries, which provide the same API for users.

server1 is configured in **`eop_version/config/spring/ol-redis.xml`**.

```
30  <-- Use Spring Data Redis -->
31  <--  For more information: http://docs.spring.io/spring-data/redis/docs/2.0.0.M1/reference/html/ -->
32   <bean id="redisTemplateRepo" class="com.sas.finance.fraud.ol.enrich.RedisTemplateRepository">
33      <constructor-arg name="sources">
34         <map>
35            <entry key="server1" value-ref="stringRedisTemplate"/>
36         </map>
37      </constructor-arg>
38   <>/bean
```

`RedisTemplateRepository` can be configured with multiple instances of `RedisTemplates`. Therefore, it is possible for a mapping file to enrich data using multiple Redis servers.

Here is the exact Groovy method signature for Redis server enrichment:

```
def getValue(RedisTemplate<?, ?> redisTemplate, String value, Map<String,
String> records, Map<String, Object> cache)
```

Data Enrichment Using Free-Form Text Mining

SAS Business Orchestration Services can extract named entities from free-form text using a third-party library. The free-form text can be part of the input data, retrieved from enrichment resources, or it can be received from other places.

The sample implementation uses Stanford Named Entity Recognizer (NER), although it is possible to use other third-party packages to do the name recognition. SAS Business Orchestration Services does not provide any of the NER components. The NER components need to be downloaded separately.

NER can recognize personal names, organization names, and locations from free-form text when a proper classifier library is provided.

In the following example, the free-form text is from a request field named FILLER. The recognized name is assigned to the dee_entity_id_4 field. The organization name is assigned to the dee_entity_id_5 field. The location is assigned to the dee_entity_id_6 field.

```
<-- A sample free-form text mining field -->
 <field name="dee_entity_id_4">
    <miningField rscName="miner01" name="FILLER">
       <groovy>
          <![CDATA[
             if (value==null) return null;

             List<Triple<String, Integer, Integer>> l = mineTemplate.classifyToCharacterOffsets(value)

             for (Triple<String, Integer, Integer> e : l) {
                cache.put("M_" + e.first(), value.substring(e.second(), e.third()))
             }

             String s = cache.get("M_PERSON")
             return (s != null && s.length()>60)? s.substring(0, 60) : s
          ]]>
       </groovy>
    </miningField>
 </field>

 <field name="dee_entity_id_5">
    <miningField rscName="miner" name="FILLER">
       <groovy>
          <![CDATA[
             String s = cache.get("M_ORGANIZATION");
             return (s != null && s.length()>60)? s.substring(0, 60) : s;
          ]]>
       </groovy>
    </miningField>
 </field>

 <field name="dee_entity_id_6">
    <miningField rscName="miner" name="FILLER">
       <groovy>
          <![CDATA[
             String s = cache.get("M_LOCATION");
             return (s != null && s.length()>80)? s.substring(0, 80) : s;
          ]]>
       </groovy>
    </miningField>
```

```
457   </field>
```

Line 430 shows `rscName=minor01`, which is configured in **`eop_version/config/spring/ol-mining.xml`**. The `MiningRepository` can take multiple instances of NER, which makes it possible to enrich data using different NERs.

```
11  <bean id="miningTemplateRepo" class="com.sas.finance.fraud.ol.enrich.MiningRepository">
12     <constructor-arg name="sources">
13        <map>
14           <entry key="miner01" value-ref="NER"/>
15        </map>
16     </constructor-arg>
17  </bean>
18
19  <-- This NER is for PERSON, ORGANIZATION, LOCATION. -->
20  <bean id="NER" class="edu.stanford.nlp.ie.crf.CRFClassifier" factory-method="getClassifier">
21     <constructor-arg value="classifiers/english.all.3class.distsim.crf.ser.gz"/>
22  </bean>
```

Data Enrichment Using Other Resources

The same patterns used in mapping for an RDBMS, Redis server, and free text mining can be used to implement enrichment using other resources. For more information, contact your SAS Fraud Management support personnel.

Data Enrichment Using Camel Processors

Data enrichment can be done at different stages of request processing, and it can be done using different resources. It depends on processing routes design. The mapping file is usually used after input data is converted to request name-value pairs, but before the SAS Fraud Management transaction object is created.

Data enrichment can happen at any step of request processing. In-bound data enrichment can happen at any time before the transaction is sent to SAS Fraud Management. This means that enrichment using a mapping file is limited as to when enrichment can be done.

Data enrichment via Camel processors is more flexible as to when it can be applied, although it means creating a custom bean or at least extending some SAS Business Orchestration Services classes from the com.sas.finance.frud.ol.enrich package.

The custom bean is inserted into the request processing flow as one step or node, and it can implement logic for accessing any other services.

The `com.sas.finance.fraud.ol.enrich` package provides a base interface, an abstract class, and the following sample enrichment examples using a relational database and a Redis server:

```
public interface IEnrich<T,R>;
public abstract class AbstractEnricher<T,R> implements IEnricher<T,R>;
public abstract class DbEnricher<T,R> extends AbstractEnricher<T,R>;
```

Data Enrichment Considerations

Data Enrichment Efficiency Using Service-Level Agreements

SAS Business Orchestration Services uses error-handling mechanisms such as time-outs to ensure data enrichment efficiency.

For each request processing, there is typically a service-level agreement (SLA) time constraint such as 100ms. If a normal response cannot be given in that amount of time, then a pre-agreed response needs to be sent to the

caller. For more information, see “Configuring Stand-In Response” on page 32. Each node in the processing routes is responsible for overall time.

Data Enrichment Efficiency Using a Mapping File

When data enrichment is done using a mapping file, SAS Business Orchestration Services has already optimized the related components by pre-loading, pooling, or both.

It is possible that `<dbField>`, `<redisField>`, and `<miningField>` might be used multiple times in the same mapping file. This means that the related component and API are invoked multiple times. For example, if `<dbField>` is used 10 times in a mapping file, and each time it calls `jdbcTemplate.queryForObject()`, then the database is accessed 10 times per request for data enrichment alone.

Reduce the number of queries as much as possible. Use the cache map to store query results for later use. “For an example of cache usage, see “Data Enrichment Using Free-Form Text Mining” on page 20.

The scope for the cache map can be per request or per mapping file. The performance improvement is significant when cache is used properly.

Data Enrichment Efficiency Using a Camel Processor

When custom processor beans are implemented and used for data enrichment, you must ensure their efficiency.

Data Enrichment Error Handling

You might encounter errors with data enrichment. For example, you might be unable to reach resources, data might be missing, an operation might time out, and so on. The logic needed to handle errors can be very different for each case or customer.

Errors can be divided into recoverable errors and irrecoverable errors. Irrecoverable errors remain as errors each time you retry. Recoverable errors are temporary and might not cause a problem on the next try.

From an application business logic point of view, data enrichment operations can be divided into two categories:

- Mandatory enrichment

  When mandatory enrichment fails to complete successfully, the request should be marked as failed. The normal request routing should stop.

- Optional enrichment

  When operational enrichment fails, the request processing should continue as usual.

Camel makes it simple for users to design request routing for handling both categories.

Configuring Outbound Data Mapping

The outbound data mapping generates response messages for client systems in expected formats. SAS Business Orchestration Services has a few built-in outbound data mappers, and you can also create custom bean processors if more customized behavior is preferred.

Outbound Mapper Using Mapping File

SAS Business Orchestration Services uses outbound mapping XML files to collect and convert response information from different sources. Those sources can be any of the following:

- input request data
- transaction responses from SAS Fraud Management

- constant values
- data enrichment resources

The mapping results can be returned as a map of name-value pairs, XML, or JSON. It can be returned to client systems directly, or transformed into other formats depending on your route design.

First, configure a Spring bean. Here is an example of a bean using a `mapperId` of `ctMasterResp`, which identifies a mapping.

```
<bean id="OBMCtMaster"
class="com.sas.finance.fraud.ol.demos.OutboundMapperUsingXmlMapping">
     <constructor-arg name="mapperId" value="ctMasterResp"/>
     <constructor-arg name="format" value="MAP"/>
</bean>
```

Note: The `mapperId` value should match with the value in your outbound mapping XML file.

Once the bean is defined, it can be used in a route. Depending on the route design and whether you want the message body to be updated with outbound mapping results, there are two different methods that you can invoke:

1 `<bean ref="OBMCtMaster" method="mapFromExchange"/>`

   This method uses the mapping information and creates an outbound result map (if it does not exist), and then puts the result map in a Camel exchange property. This step can be invoked more than once during message routing, so that information can be collected at different stages of message routing.

2 `<bean ref="OBMCtMaster" method="mapFromExchangeAndUpdateBody"/>`

   In addition to performing the Method 1 tasks, this method converts the result map into the requested format, and puts it into Camel exchange body. This is usually the last step of outbound mapping using this bean.

Mapping from an Input Request Data

Some applications might require that certain input field value or values be sent back in a response with or without a transformation.

The following example shows how this can be achieved:

```
37     <--  Use an incoming request source field -->
38     <field name="ResponseName">
39        <srcField name="Foo.Bar" default="y">
40           <groovy>
41              return (value == 'FOO')? 'OOF' : value
42           </groovy>
43        </srcField>
44     </field>
```

This mapping tells SAS Business Orchestration Services to look for the input field named Foo.Bar. If its value is FOO, then it returns OOF. Otherwise, it returns its value. The returned value is assigned to a field named ResponseName, and this is part of the response data.

Mapping from Constant Values

Mapping a constant string value to a response field can be configured two ways. You can use either a `<constant>` tag or a default. The following example shows both methods.

```
46     <-- Two ways to specify static value -->
47     <field name="Response.Fld01">
48        <constant>
```

```
49          <![CDATA[plain text here.]]>
50        </constant>
51     </field>
52
53     <field name="Response.Fld02" default="FooBar"/>
```

Mapping from Enrichment Resources

Although it is uncommon to use data enrichment for outbound mapping, the functionality is available. The outbound mapping shares the same mapping file syntax and parsing engine as that of inbound mapping.

For detailed syntax about data enrichment, see "Data Enrichment" on page 17. The main difference is the field name. For inbound mapping, the field name is a SAS Fraud Management field name. However, for outbound mapping, it is the response field name.

Outbound Mapping Using Template Files

Some client applications might expect that response messages are sent in a fixed format that replaces certain fields or tokens for each transaction. In this case, the template file-based mapping can be used.

The template file-based mapping depends on the outbound mapper using a mapping file with the format set to Map. You also need to create a template file and place it in the **<*eop_install_dir*>/config/mappings/outbound/templates/** directory.

Here is an example of a template file:

```
$ cat custom_outbound.template
<Response>
   <ServiceStatus>${Service.Status}</ServiceStatus>
   <ServiceError>${Service.ExceptionMsg}</ServiceError>
   <CardNumber>${CardNumber}</CardNumber>
   <Smh_rtn_code>${Response.Code}</Smh_rtn_code>
   <Request-Date>${Request Date}</Request-Date>
   <Ming-Fields>
      <Person>${FreeText Name}</Person>
      <Organization>${FreeText Organization}</Organization>
      <Location>${FreeText Location}</Location>
   </Ming-Fields>
</Response>
```

The key names that are enclosed in `${ . . . }` are replaced by values from the key-value map.

You also need to configure a Spring bean. Here is an example:

```
<bean id="templateMapper" class="com.sas.finance.fraud.ol.demos.OutboundTemplateMapper">
   <constructor-arg name="templateFilename" value="${demo_outbound_response_template}"/>
</bean>
```

In your route, you also need to add a bean processor after the mapper node that uses the mapping file. Here is an example:

```
<bean ref="OBMCtMaster" method="mapFromExchangeAndUpdateBody"/>
<bean ref="templateMapper"/>
```

3

Managing SAS Business Orchestration Services

Logging

Logging Overview

SAS Business Orchestration Services uses log4j for logging. Log4j is flexible and configurable via its configuration file. For SAS Business Orchestration Services, this file is located at **<*eop_version*>/config/log4j.xml**.

The default logging level is set to ERROR, and log files are written to the **<*eop_version*>/logs/** folder.

For most configurations, you need to change only the value for the root logger level.

```
<root>
   <level value="DEBUG"/>
   <appender-ref ref="file"/>
</root>
```

From the most verbose to least verbose, valid values for the logging level are the following:

- ALL
- DEBUG
- INFO
- WARN
- ERROR
- FATAL
- OFF

SAS Business Orchestration Services recognizes modifications to the log4j.xml file at run time. There is no need to restart SAS Business Orchestration Services.

Logging Throughput of a Route

You change only one line of your configuration to log the real-time throughput on a route.

```
<rest>
<get uri="/ping">
   <route id="Ping">
      <to uri="log:myRoute01?level=INFO&groupInterval=5000"/>
      <transform><constant>pong</constant></transform>
   </route>
</get>
</rest>
```

If the global logging level is INFO or lower, then in the log file that you see prints every 5000 milliseconds:

```
myRoute01:159 - Received: 500 new messages, with total 103000 so far. Last group took:
5000 millis which is: 100 messages per second. average: 100
```

Show Time Spent on Each Route Step

One or more routes might fulfill the processing of an incoming request, and each route might contain one or more steps. If you want to see how much time was spent on each step during the processing of a request, then you can turn on `MsgHistLogger`. Within the route of your choice, you can insert the following:

```
<onCompletion parallelProcessing="true">
   <to uri="direct:MsgHistLogger" />
</onCompletion>
```

If the global logging level is INFO or lower, then in the log file you see messages such as the following:

```
    generalRoute-onCompletion2-onCompletion    : 0 ms
    generalRoute-restBinding5-restBinding                  : 3 ms
    generalRoute-general_route_logger-to                   : 1 ms
    generalRoute-convertBodyTo2-convertBodyTo              : 0 ms
    generalRoute-bean14-bean                               : 1 ms
    generalRoute-to8-to                                    : 12 ms
        Dispatcher-doTry1-doTry                            : 12 ms
            Dispatcher-bean3-bean                          : 5 ms
            Dispatcher-StoreUnqKeyFor_format-bean          : 0 ms
            Dispatcher-to1-to                              : 6 ms
                coreHandler-onCompletion1-onCompletion     : 0 ms
                coreHandler-setTransactionId-bean          : 0 ms
                coreHandler-bean8-bean                     : 1 ms
                coreHandler-validation-bean                : 2 ms
                coreHandler-ODE Load Balancer-loadBalance  : 0 ms
                coreHandler-odeSimulator01-bean            : 0 ms
                coreHandler-bean9-bean                     : 0 ms
                coreHandler-cacheResponseForStandIn-bean   : 0 ms
    generalRoute-bean15-bean                               : 1 ms
    generalRoute-generalRoute-removeHeaders                : 0 ms
    generalRoute-to7-to                                    : 1 ms
        MsgHistLogger-messageHistoryLogger-bean            : 1 ms
```

The message is indented to separate child routes from parent routes.

Monitoring SAS Business Orchestration Services

Users can look at SAS Business Orchestration Services log files and the SAS Business Orchestration Services management console for status updates. The management console is a web application that can be deployed in a web container such as Tomcat. It is recommended that you use Tomcat 7 or later. It includes the management console and an agent.

Running the Management Console Agent

The management console talks to the agent that runs on the same box with SAS Business Orchestration Services. The agent is not started by default. You need to start the agent before you launch the SAS Business Orchestration Services management console, and you should stop it when you are not using it.

To start or stop the SAS Business Orchestration Services management console agent, invoke the following script:

```
<eop_version>/bin/eopmcagent.sh start
<eop_version>/bin/eopmcagent.sh stop
```

Using SAS Business Orchestration Services Management Console

Deploying SAS Business Orchestration Services Management Console

Tomcat has been tested as a web container for hosting the SAS Business Orchestration Services management console. It can be installed on a separate host or co-located with SAS Business Orchestration Services. For more information about a Tomcat installation, see Tomcat User Guide.

To deploy the management console, place the following WAR files into the **webapps** folder in Tomcat:

- hawtio-default-1.4.68.war
- sasol-branding.war
- olstat-plugin.war

Starting SAS Business Orchestration Services Management Console

The web application is started when Tomcat is started. From your web browser (preferably Chrome), use the following URL:

```
http://localhost:8080/hawtio-default-1.4.68/jvm/connect
```

In the **Connections Settings** form, provide the correct SAS Business Orchestration Services host name and port number (8778), and then click **Connect to remote server**.

4

Managing Data Flow

Store and Forward

SAS Business Orchestration Services moves transactions to store and forward (SAF) when SAS Fraud Management is not available, and forwards stored transactions when at least one SAS Fraud Management server becomes available.

Enabling Store and Forward

Overview

There are two parts to enabling SAF:

- enabling store
- enabling forward

Enabling Store

Catching exceptions due to SAS Fraud Management connection problems can trigger the SAF store operations.

```
104  <route id="customTransactionHandler" startupOrder="48">
105     <from uri="disruptor:customTransaction?concurrentConsumers=8"/>
106     <doTry id="Main route(custom)">
107        <convertBodyTo type="java.lang.String"/>
108        <bean id="decodeWebString" ref="odeUtils" method="decodeWebString"/>
109        <bean id="cust_to_Transaction" ref="inboundMapper" method="customToTransaction"/>
113        <-- steps omitted -->
114        <toD id="SendCustomTxnToODE" uri="disruptor:sendTransactionToODE?timeout=
${exchangeProperty.REMAIN_TIME}&waitForTaskToComplete=Always"/>
124        <doCatch id="Catch SendToODE Error(custom)">
125           <exception>java.net.ConnectException</exception>
126           <exception>java.lang.RuntimeException</exception>
127           <to id="SendErrorHandler(custom)" uri="direct:handleSendError"/>
128        </doCatch>
129     </doTry>
130  </route>
```

The child route invokes SAF store method, which stores the failed transactions to the SAF data store.

```
193  <route id="handleSendError" startupOrder="20">
194     <from uri="direct:handleSendError" />
195     <doTry>
196        <bean ref="saf" method="store"/>
197           <!-- <to uri="activemq:saf.failed.xtions?messageConverter=
198           #messageConverter&disableReplyTo=true" /> -->
199           <to uri="log:?level=ERROR" />
200           <doCatch>
201              <exception>java.lang.Throwable</exception>
202              <to uri="log:?level=ERROR" />
203           </doCatch>
204        </doTry>
205     <bean ref="saf" method="setLastSendErrorTime" />
206     <!-- Get stand-in response. -->
207     <bean ref="requestTimeoutHandler" method="handleTimeout" />
208  </route>
```

Near the end of the child route definition, a stand-in response is retrieved. For more information, see“Overview” on page 32.

Enabling Forward

There are different ways to trigger SAF forward operation. Out of the box, SAS Business Orchestration Services offers an implementation that uses a Camel timer to trigger it. The `SAF` bean has built-in logic to determine whether it is the right time to forward and how fast it can forward transaction. This logic is discussed in later sections.

```
222  <route id="SAF" startupOrder="54">
223     <from uri="timer://SAF?fixedRate=true&period={{saf_interval_milli}}"/>
225        <bean id="SAF Processor" ref="saf" method="forward"/>
226  </route>
```

The `saf_interval_milli` property determines how often SAF is asked to try forward operations. It is not the rate at which failed transactions are forwarded.

Configuring SAF Data Store

SAS Business Orchestration Services supports a few SAF data store solutions out of box. To support other data store solutions, see “Using Unsupported Data Store” on page 32.

In-Memory Data Store

The following example uses an in-memory queue to store failed transactions. It is used primarily for test and demonstration purposes. Do not use it in any production environment.

The following example shows an in-memory configuration. It can also be found in the ol-beans.xml file.

```
209  <bean id="saf" class="com.sas.finance.fraud.ol.saf.BasicSafImpl">
210     <constructor-arg name="waitSecondsAfterLastTimeout" value="30"/>
211     <constructor-arg name="waitSecondsAfterLastError" value="60"/>
212     <constructor-arg name="maxTxnsPerSecond" value="2000"/>
213     <property name="template" ref="producerTemplateForSaf"/>
214     <property name="maxQueueSize" value="1000000"/>
215     <property name="odeStatus" ref="ODEStatus"/>
216  </bean>
```

The Camel template is an object that is used for forwarding transactions to a route when it is time to forward.

The `waitSecondsAfterLastTimeout`, `waitSecondsAfterLastError`, and `odeStatus` are used for determining when to start forwarding transactions. `odeStatus` can tell whether there are any reachable SAS Fraud Management systems. `waitSecondsAfterLastTimeout` specifies the number of seconds to wait after the last time-out before the next forward attempt can be made.

`maxTxnsPerSecond` controls the transaction forwarding rate.

Active MQ

The configuration for using Active MQ as a data store is similar to the in-memory data store, except for a few arguments for specifying Active MQ parameters.

If you have an Active MQ server running already, use the `uri` to point to your server. `destination` configures the destination queue name for storing transactions.

```
218  <-- Use ActiveMq message queue based Store & Forward. -->
219  <bean id="saf" class="com.sas.finance.fraud.ol.saf.ActiveMqSafImpl">
220     <constructor-arg name="waitSecondsAfterLastTimeout" value="5"/>
221     <constructor-arg name="waitSecondsAfterLastError" value="10"/>
222     <constructor-arg name="maxTxnsPerSecond" value="1000"/>
223     <constructor-arg name="uri" value="tcp://${saf_activemq_host}:61616?wireFormat.maxInactivityDuration=0"/>
224     <constructor-arg name="destination" value="saf.failed.xtions"/>
225     <constructor-arg name="receiveWaitMs" value="10"/>
226     <property name="template" ref="producerTemplateForSaf"/>
227     <property name="odeStatus" ref="ODEStatus"/>
228  </bean>
```

IBM MQ

Depending on your configuration scenarios, there are a few overloaded constructors that you can use. SAS Business Orchestration Services expects that you have properly installed and configured an IBM MQ manager or managers.

The following example shows an IBM MQ configuration as a SAF data store. The bean takes `connectionFactory`, which points to an instance of `org.springframework.jms.connection.CachingConnectionFactory`.

```
244  <bean id="saf" class="com.sas.finance.fraud.ol.saf.IbmMqSafImpl">
245     <constructor-arg name="waitSecondsAfterLastTimeout" value="5"/>
246     <constructor-arg name="waitSecondsAfterLastError" value="10"/>
247     <constructor-arg name="maxTxnsPerSecond" value="1000"/>
```

```
248     <constructor-arg name="connectionFactory" ref="cachedConnectionFactory"/>
249     <constructor-arg name="queueName" value="saf.failed.xtions"/>
250     <constructor-arg name="receiveWaitMs" value="10"/>
251     <property name="template" ref="producerTemplateForSaf"/>
252     <property name="odeStatus" ref="ODEStatus"/>
253  </bean>
```

The bean configuration for `cachedConnectionFactory` can be found at ibmmq.xml.

Kafka

You can use a Kafka server or cluster for SAF data store. The following example shows a Kafka configuration using the `com.sas.finance.fraud.ol.saf.KafkaSafImpl` class.

```
231  <bean id="saf" class="com.sas.finance.fraud.ol.saf.KafkaSafImpl">
232     <constructor-arg name="waitSecondsAfterLastTimeout" value="30"/>
233     <constructor-arg name="waitSecondsAfterLastError" value="60"/>
234     <constructor-arg name="maxTxnsPerSecond" value="1000"/>
235     <constructor-arg name="uri" value="localhost:9092"/>
236     <constructor-arg name="topic" value="test01"/>
237     <constructor-arg name="groupId" value="OL2"/>
238     <property name="template" ref="producerTemplateForSaf"/>
239     <property name="odeStatus" ref="ODEStatus"/>
240  </bean>
```

When you deploy multiple instances of SAS Business Orchestration Services, make sure that there are multiple partitions on a topic so that each instance of SAS Business Orchestration Services can handle forwarding concurrently.

For more information about Kafka installation and configuration, see Kafka Documentation.

Using Unsupported Data Store

To use data stores that are not supported by SAF, you need to either implement an ISaf interface, or extend the `AbstractSafImpl` class. If you plan on implementing an unsupported data store, you should discuss this with your SAS Fraud Management support personnel.

Configuring Stand-In Response

Overview

Stand-in response is the last received response from SAS Fraud Management for a given account number. For a valid transaction request from a client system, a response is expected even in scenarios where there is no SAS Fraud Management server available or there is a time-out during the request processing. It is a common practice to respond with a stand-in response in those scenarios.

Here are the two things that you need to configure for a stand-in response:

1. Enable stand-in response.
2. Configure stand-in data store.

Enable Stand-In Response

You enable stand-in response through configuration. The following example shows a configuration to handle a transaction time-out. Within the route, `<doCatch>` catches the time-out exceptions that it cares about, and invokes the child route `direct:standInResponse`.

```
104  <route id="customTransactionHandler" startupOrder="48">
105     <from uri="disruptor:customTransaction?concurrentConsumers=8"/>
106     <doTry id="Main route(custom)">
107        <convertBodyTo type="java.lang.String"/>
108
109        <bean id="decodeWebString" ref="odeUtils" method="decodeWebString"/>
110        <bean id="cust_to_Transaction" ref="inboundMapper" method="customToTransaction"/>
111        <bean id="StoreUnqKeyForCustomTxn" ref="uniqueKeySaver"/>
112        <!—Steps omitted -->
119        <doCatch id="Catch Custom Timeout(custom)">
120           <exception>com.sas.finance.fraud.ol.exceptions.EnrichTimeoutException</exception>
121           <exception>org.apache.camel.ExchangeTimedOutException</exception>
122              <to id="Stand-in-Response(custom)" uri="direct:standInResponse"/>
123        </doCatch>
124     </doTry>
125  </route>
```

The following example defines the child route `direct:standInResponse`. It logs the exception, invokes some beans to track the last time-out, and calls the `handleTimeout` logic.

```
213  <route id="standInResponse" startupOrder="21">
214     <from uri="direct:standInResponse"/>
215     <to id="log warn" uri="log:?level=WARN" />
216     <bean id="safSetTimeout" ref="saf" method="setLastTimeout" />
217     <bean id="safSetTimeoutTS" ref="reporter" method="setAolLastTimeoutTimestamp"/>
218     <bean id="requestTimeoutHandler" ref="requestTimeoutHandler" method="handleTimeout"/>
219  </route>
```

The default implementation of `handleTimeout` gets the stand-in response from the stand-in data store based on the account number in the request. So, the processing of the request continues as if a transaction response was received from SAS Fraud Management. You can alter the flow and logic of the `handleTimeout` method by providing a custom implementation. However, it should not be necessary for the majority of cases.

Configuring Stand-In Data Store

SAS Business Orchestration Services updates the data store when a transaction response is received from SAS Fraud Management. It tries to retrieve a previously stored transaction per account number when it needs to. This is similar to a caching service for a data store.

Here are a few factors to consider about the data store:

- Size: The data store size is proportional to the number of unique account numbers. For each account number, there can be one data entry. A data entry contains information such as the last SAS Fraud Management transaction response, account number, and a timestamp.
- Persistence: Depending on your application requirements, the selected data store might need to persist data across SAS Business Orchestration Services restarts. SAS Business Orchestration Services offers a few choices out of the box, and they are covered in later sections.
- Speed: The choice of data store can affect the SAS Business Orchestration Services's performance while saving and retrieving to and from data store.

- High availability: The data store needs to be reliable for SAS Business Orchestration Services's stand-in response feature. When a data store becomes unavailable, SAS Business Orchestration Services operates as if the stand-in response feature is turned off.
- Accessibility among SAS Business Orchestration Services: In real deployments, multiple instances of SAS Business Orchestration Services might be used. Depending on your business needs, it might be necessary for multiple instances of SAS Business Orchestration Services to share the same data store.

Using Supported Data Stores

SAS Business Orchestration Services puts no restriction on what data stores you can use. However, it does offer a few data stores that are supported out of box.

In-Memory Data Store

The in-memory data store is the simplest and the most efficient stand-in data store that is offered by SAS Business Orchestration Services. It is also easy to configure. The sample configuration for stand-in data store can be found at **`eop_version/config/spring/ol-beans.xml`**.

```
134  <bean id="requestTimeoutHandler"
135   class="com.sas.finance.fraud.ol.standin.RequestTimeoutHandler">
136      <property name="standInDatastore">
137         <bean class="com.sas.finance.fraud.ol.standin.CachedResponses"/>
138      </property>
139  </bean>
```

However, the in-memory data store does not offer persistence, and it is not sharable among instances of SAS Business Orchestration Services. When there are too many unique account numbers, there could be a memory usage issue. Thus, it is not recommended for production use.

Redis Data Store

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. It supports various data structures and queries. Redis has built-in replication, Lua scripting, LRU eviction, transactions, and different levels of on-disk persistence, and provides high availability via Redis Sentinel, and automatic partitioning with Redis Cluster. For more information, see Redis.

The sample configuration for this stand-in data store can be found at **`eop_version/config/spring/ol-beans.xml`**.

```
139  <bean id="requestTimeoutHandler" class="com.sas.finance.fraud.ol.standin.RequestTimeoutHandler">
140     <property name="standInDatastore">
141        <bean class="com.sas.finance.fraud.ol.standin.RedisDatastore">
142           <constructor-arg name="template" ref="transactionRedisTemplate"/>
143           <constructor-arg name="stringTemplate" ref="stringRedisTemplate"/>
144           <property name="timestampProvider" ref="timestampProvider"/>
145        </bean>
146  </bean>
```

The configuration for the beans that are used by `standInDatastore` can be found in **`eop_version/config/spring/ol-redis.xml`**. If your Redis server is already installed, then you should open the camel-context.properties file and make sure `redis_host` and `redis_port` point to your server.

```
40  <!-- JedisConnectionFactory can be injected with RedisClusterConfiguration to handle clusters. -->
41  <bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"
42    p:host-name="${redis_host}" p:port="${redis_port}" p:use-pool="true" />
43
44  <bean id="stringRedisTemplate" class="org.springframework.data.redis.core.StringRedisTemplate"
45    p:connection-factory-ref="jedisConnectionFactory"/>
```

```
46
47 <bean id="transactionRedisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
48    <property name="connectionFactory" ref="jedisConnectionFactory"/>
49    <property name="keySerializer">
50       <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>
51    </property>
52    <property name="valueSerializer">
53       <bean class="com.sas.finance.fraud.ol.standin.TransactionRedisSerializer"/>
54    </property>
55 </bean>
56
57 <!-- By default, rqo_tran_date and rqo_tran_time are being used. It is configurable though. -->
58 <bean id="timestampProvider" class="com.sas.finance.fraud.ol.standin.ConfigurableDateTimeProvider"/>
```

The `timestampProvider` bean ensures that the transaction response that is stored in data store is the latest response. By default, the rqo_tran_date and rqo_tran_time fields are used to calculate a timestamp. As the class name `ConfigurableDateTimeProvider` implies, field names are configurable. Line 58 is equivalent to the following bean configuration. If you want to use different field names, then you can modify the values accordingly.

```
58 <bean id="timestampProvider" class="com.sas.finance.fraud.ol.standin.ConfigurableDateTimeProvider"/>
59    <constructor-arg name="dateFieldName" value="rqo_tran_date"/>
60    <constructor-arg name="timeFieldName" value="rqo_tran_time"/>
61    <constructor-arg name="dateTimeFormat" value="yyyyMMddHH:mm:ss.SS "/>
62 </bean>
```

Relational Data Store

SAS Business Orchestration Services also supports using a relational database as stand-in data store, although it might not perform as well as the previous two choices. Performance varies depending on your environment and system settings.

The configuration for using a relational database is similar to using a Redis server.

```
146 <bean id="requestTimeoutHandler" class="com.sas.finance.fraud.ol.standin.RequestTimeoutHandler">
147    <!-- Use a relational database as StandIn Dat48  a Store -->
148    <property name="standInDatastore">
149       <bean class="com.sas.finance.fraud.ol.standin.RelationalDbDatastore">
150          <constructor-arg name="jdbcTemplate" ref="jdbcTemplate"/>
151          <property name="timestampProvider" ref="timestampProvider"/>
152       </bean>
153    </property>
154 </bean>
```

Line 150 uses the `jdbcTemplate` bean, which is a Spring `JdbcTemplate` instance. Its exact configuration depends on which type of relational database you use, and additional options for connection pooling. Here is an example for a MySql server.

```
26 bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"
27    <property name="dataSource" ref="dataSource"/>
28 </bean>
29 <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">
30    <property name="driverClass" value="com.mysql.jdbc.Driver" />
31    <property name="jdbcUrl" value="${MySql_Server_Port}" />
32    <property name="user">
33       <bean class="com.sas.finance.fraud.ol.secure.OLSecureUsername">
34          <constructor-arg name="identity" value="${MYSQL_CREDENTIAL}"/>
35       </bean>
36    </property>
```

```
37     <property name="password">
38        <bean class="com.sas.finance.fraud.ol.secure.OLSecurePassword">
39           <constructor-arg name="identity" value="${MYSQL_CREDENTIAL}"/>
40        </bean>
41     </property>
42     <property name="maxPoolSize" value="25" />
43     <property name="minPoolSize" value="10" />
44     <property name="maxStatements" value="100" />
45     <property name="testConnectionOnCheckout" value="true" />
46  </bean>
```

The configuration for `user` and `password` might not at first glance be straightforward. For more information, see "Encrypting Credentials" on page 51.

You need to create a table in your relational database that uses the SQL equivalent to the following example, which is for a MySQL database.

```
CREATE TABLE StandIn (Id INTEGER AUTO_INCREMENT PRIMARY KEY,
AcctNum varchar(50), Response BLOB, Timestamp BIGINT, CONSTRAINT uc_key UNIQUE (AcctNum));
```

Using Unsupported Data Stores

SAS Business Orchestration Services can be extended to support other types of data stores, although only three data stores are supported out of the box.

To support new data store, implement the following SAS Business Orchestration Services interface and configure the Spring bean:

```
com.sas.finance.fraud.ol.standin.IStandInDatastore
```

Configuring Communication between SAS Business Orchestration Services and SAS Fraud Management

By default, communication between SAS Business Orchestration Services and SAS Fraud Management uses sockets. SAS Business Orchestration Services offers IBM MQ as an alternative.

Using Sockets for Communication

Camel offers several components for socket communication. SAS Business Orchestration Services uses the Camel:Netty4 component to talk to SAS Fraud Management. For example, the following line of a configuration uses a Netty4 socket to communicate with a server listening on port 5018.

```
<to id="rdcesx12073" uri="netty4:tcp://rdcesx12073.race.sas.com:5018
?sync=true&encoder=#encoder&decoder=#decoder"/>
```

The following options given in the uri:

- sync=true: A response is expected for each request sent.
- encoder=#encoder: Use the specified encoder bean when sending a request.
- decoder=#decoder: Use the specified decoder bean when receiving a response.

There are many other options available for the component.

In a real deployment, more than one socket might be configured to provide load balancing and failover support. For more information, see "SAS Business Orchestration Services Scalability and High Availability" on page 43.

Using IBM MQ for Communication

Using IBM MQ for Communication Overview

Although using sockets between SAS Business Orchestration Services and SAS Fraud Management is recommended, using IBM MQ to exchange request and response messages between SAS Business Orchestration Services and SAS Fraud Management works for systems that do not require very high throughput and have very low latency.

The **eop_version/config/spring/camel-context-simple-Q.xml** file provides a complete sample configuration. Here are a few key configuration differences:

- Instead of using the Camel:Netty4 component, use Camel:JmsComponent send messages to request the queue.

  ```
  <to id="sendMessageToMQ"
  uri="ibmmq:queue:{{ibm.mq.req.queue.name}}?messageConverter=
  #messageConverterMQ&disableReplyTo=true"/>
  ```

  The `ibmmq` is an instance of `JmsComponent` configured in **ibmmq.xml**.

  The `ibm.mq.req.queue.name` is the name of the SAS Fraud Management request queue, which is configured for SAS Fraud Management to listen for transaction requests.

- Use an asynchronous component to get a response from the SAS Fraud Management response queue.

  ```
  <to id="asyncMsgRetrieval" uri="mrp:messageReceiver"/>
  ```

  This component uses response messages received from the `ibm.mq.rsp.queue.name queue`. SAS Fraud Management sends response messages to this queue.

  ```
  199  <route id="ODE_RESP_Q_Handler" startupOrder="32">
  200        <from
  201         uri="ibmmq:queue:{{ibm.mq.rsp.queue.name}}?messageConverter=#
  202         messageConverterMQ&disableReplyTo=true"/>
  203        <bean id="storeResponseTxn" ref="txnMsgHandler" method="process"/>
  204        <bean id="cacheResponseForStandIn" ref="requestTimeoutHandler" method="handleSuccess"/>
  205  </route>
  ```

Multiple MQConnector versus Single MQConnector

MQConnector is configured on SAS Fraud Management for receiving transaction messages from a request queue and sending processed results to a response queue. SAS Business Orchestration Services sends transaction requests to the request queue. SAS Business Orchestration Services receives transaction responses from the response queue.

It is possible to have multiple SAS Fraud Management systems that point to the same pair of request and response queues, or to configure each SAS Fraud Management system to point to different pair of request and response queues.

Multiple instances of SAS Business Orchestration Services can use one pair of request and response queues, or multiple pairs of request and response queues.

Configuration for a Single MQConnector

If you use only one pair of queues for multiple SAS Business Orchestration Services, then you need to use a message selector for each instance of SAS Business Orchestration Services to receive the correct responses.

```
198  <route id="ODE_RESP_Q_Handler" startupOrder="32">
199     <from
```

```
uri="ibmmq:queue:{{ibm.mq.rsp.queue.name}}?messageConve
rter=#messageConverterMQ&disableReplyTo=true&selector={{ibm.mq.msg.selector}}"/>
203     <bean id="storeResponseTxn" ref="txnMsgHandler" method="process"/>
204     <bean id="cacheResponseForStandIn" ref="requestTimeoutHandler" method="handleSuccess"/>
205  </route>
```

Line 199 specifies the selector. The selector uses one of the message properties that are inserted in the original request message. For each instance of SAS Business Orchestration Services, you need to make sure a unique value is used for `olMsgSelectorIdx` within the SAS Business Orchestration Services group.

```
60  <bean id="messageConverterMQ" class="com.sas.finance.fraud.ol.converter.OLMessageConverterForOdeMQ">
61     <property name="olMsgSelectorIdx" value="${ol.msg.selector.idx}"/>
62  </bean>
```

Your use of selector could add processing overhead on the message consuming end, thus increasing the overall latency. However, it reduces the number of queues to manage. In an environment where very high throughput is expected, you should consider using multiple instances of MQConnector.

Configuration for Multiple MQConnectors

In this case, each instance of SAS Business Orchestration Services uses a different pair of request and response queues, so no selector is needed. Then, on the SAS Fraud Management side, it means that multiple instances of MQConnector need to be configured on each SAS Fraud Management system, so that all SAS Fraud Management systems can process requests from all request queues.

Here is the SAS Business Orchestration Services configuration for multiple instances of MQConnector:

```
199 <route id="ODE_RESP_Q_Handler" startupOrder="32">
200     <from
         uri="ibmmq:queue:{{ibm.mq.rsp.queue.name}}?messageConve
         rter=#messageConverterMQ&disableReplyTo=true"/>
203     <bean id="storeResponseTxn" ref="txnMsgHandler" method="process"/>
204        <bean id="cacheResponseForStandIn" ref="requestTimeoutHandler" method="handleSuccess"/>
205  </route>
206  <bean id="messageConverterMQ" class="com.sas.finance.fraud.ol.converter.OLMessageConverterForOdeMQ"/>
```

You can expect better performance from multiple instances of MQConnector. If they cannot provide sufficient performance, then you should use sockets.

Consuming SAS Fraud Management Messages on Downstream Queues

Previous sections focus primarily on configuring SAS Business Orchestration Services to receive and process transaction requests. SAS Business Orchestration Services can be easily configured to consume SAS Fraud Management messages (such as enterprise case management system (ECMI) messages and alerts) on downstream queues, and then process and deliver these messages according to your business needs.

The configuration is typically done by adding a route or routes, so that SAS Business Orchestration Services is able to use multiple transports.

Here is an example route that consumes SAS Fraud Management ECMI messages, processes them, and then sends them to different web services based on message content.

```
508  <route id="SAS Fraud Management_ECMI_CONSUMER">
509     <from
         uri="ibmmq:queue:{{ibm.mq.queue.name}}?mapJmsMessage=false"/>
511     <bean id="FulFill Message Processor"
         ref="NABOutboundPmtBean" method="handleFulFillMessages"/>
```

```
513     <setHeader headerName="Exchange.CONTENT_TYPE">
514        <constant>text/plain</constant>
515     </setHeader>
516
517     <choice id="ActionCodeBasedRouting">
518        <when>
519           <simple>${header.FFCode} == 'auto_action_1',<simple>
520           <bean id="XmlToJson" ref="jsonMapperUtil" method="xmlToJson"/>
521           <to id="SendToWebService1" uri="{{web.service.destination.1}}?bridgeEndpoint=true"/>
522        </when>
523        <when>
524           <simple>${header.FFCode} == 'auto_action_2'</simple>
525           <multicast parallelProcessing="true" stopOnException="false">
526              <to id="SendToWebService2" uri="{{web.service.destination.2}}?bridgeEndpoint=true"/>
527              <to id="SendToWebService3" uri="{{web.service.destination.3}}?bridgeEndpoint=true"/>
528           </multicast>
529         </when>
530         <otherwise id="NoMatchOnActionCode">
531            <to id="NoMatch_SendToQ" uri="activemq:ode.analyst.ffcode.other?disableReplyTo=true"/>
532         </otherwise>
533     </choice>
534  </route>
```

Line 509 specifies the endpoint from which messages are received. In this case, the endpoint is IBM MQ.

Line 511 invokes the bean processor method named `handleFulFillMessages` for parsing the incoming messages and sets them up for further routing. From Line 517 to 533, it uses content-based route EIP patterns and multicast EIP patterns to route messages to different destinations.

Data Stores

Data Store Overview

SAS Business Orchestration Services features, such as enrichment, store and forward, and stand-in response can be configured to use different types of external data stores. Properly selected data stores help improve overall system performance, throughput, and reliability. In addition to these features, a data store can also be used for other purposes, such as storing intermediate processing results.

External data stores are not limited to relational databases. You can also use a more high-performing in-memory data store, such as Redis or Pivotal GemFire Powered by Apache Geode. This section focuses on configuring SAS Business Orchestration Services to use either Redis or GemFire. Redis and GemFire provide similar functionality. SAS Business Orchestration Services does not favor one data store over the other. You choose which data store to use based on things like your existing infrastructure.

Note: The specifics of how you install and configure a Redis and GemFire data store are not covered in this document. For more information, see the documentation for that product.

Configuring a Data Store

SAS Business Orchestration Services supports Pivotal GemFire powered by Apache Geode and Redis.

Pivotal GemFire Powered by Apache Geode

The following example has one GemFire region named region01 acting in PROXY mode. The region is hosted in the server cluster and connects through two locators in myPool.

```
<gfe:client-cache id="eop-datastore" pool-name="myPool"/>
<gfe:pool id="myPool">
   <gfe:locator host="${datastore.gemfire.locator1.host}" port="${datastore.gemfire.locator1.port}"/>
   <gfe:locator host="${datastore.gemfire.locator2.host}" port="${datastore.gemfire.locator2.port}"/>
</gfe:pool>
<gfe:client-region id="region01" cache-ref="eop-datastore" pool-name="myPool" name="region01"
shortcut="PROXY"/>
```

The following example code uses the region, region01, that was created above to configure a data store for general use and a data store proxy for use by Apache Camel.

```
<bean id="datastore" class="com.sas.finance.fraud.ol.datastore.gemfire.GeodeFactory"
      factory-method="getGeodeStrMapCache">
   <constructor-arg name="region" ref="region01"/>
   <constructor-arg name="mappingsRepository" ref="extCsvMappingsRepository"/>
   <constructor-arg name="fieldsForValueId" value="value_selection"/>
</bean>
<bean id="datastoreProxy" class="com.sas.finance.fraud.ol.syf.datastore.DatastoreProxy">
   <constructor-arg name="ds" ref="datastore"/>
</bean>
```

Redis

The following code example configures a stand-alone Redis with Jedis connection factory.

```
<bean id="redisStandaloneConfiguration" class="org.springframework.data.redis.connection.
RedisStandaloneConfiguration">
   <constructor-arg name="hostName" value="${redis.hostname}"/>
   <constructor-arg name="port" value="${redis.port}"/>
</bean>
<bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"
init-method="afterPropertiesSet">
   <constructor-arg name="standaloneConfig" ref="redisStandaloneConfiguration"/>
   <property name="usePool" value="true"/>
</bean>
```

The following example code creates a Redis backed data store using the Jedis connection factory that was configured above. It also creates a data store proxy for use by Apache Camel.

```
<bean id="stringRedisSerializer" class="org.springframework.data.redis.serializer.StringRedisSerializer"/>
<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate"
init-method="afterPropertiesSet">
   <property name="connectionFactory" ref="jedisConnectionFactory"/>
   <property name="keySerializer" ref="stringRedisSerializer"/>
   <property name="valueSerializer" ref="stringRedisSerializer"/>
</bean>
<bean id="redisDataCache" class="com.sas.finance.fraud.ol.datastore.redis.RedisDataCache">
   <constructor-arg name="redisTemplate" ref="redisTemplate"/>
</bean>
<bean id="datastoreProxy" class="com.sas.finance.fraud.ol.syf.datastore.DatastoreProxy">
   <constructor-arg name="ds" ref="redisDataCache"/>
</bean>
```

Storing Data

The following route removes a transaction from an IBM message queue and converts it to a Java Map interface. The Map data is then stored in the data store with txnId as the key.

```
<route id="transcation_to_datastore">
   <from uri="Ibmmq:queue:{{ibm.mq.queue.name}}?disableReplyTo=true"/>
   <bean id="byte_to_txn" ref="messageConverter" method="byteArrayToTransaction"/>
   <bean id="create_txn_backed_map" beanType="com.sas.finance.fraud.ol.xmlmapping.utils.TransactionBackedMap"
            method="createTransactionMap"/>
   <bean ref="datastoreProxy" method="store(${body.get('txnId')}, ${body})"/>
</route>
```

Retrieving Data

The following route retrieves data with a key that equals the header.dataKey in a Camel exchange. The header.dataKey can be set before invoking the route. The retrieved data is then stored in the retrieved_data property in the exchange.

```
<route id="datastore_retrieve">
   <bean ref="datastoreProxy" method="retrieve(${exchange}, ${header.dataKey}, 'retrieved_data')"/>
</route>
```

5

SAS Business Orchestration Services Scalability and High Availability

Scalability and High Availability Overview

SAS Business Orchestration Services is scalable both vertically and horizontally to meet throughput and latency requirements. Systems are usually scaled up vertically by using more powerful machines. Horizontal scalability is typically achieved by deploying more physical or virtual nodes.

You can also deploy SAS Business Orchestration Services in different ways to eliminate a single point of failure, such that the resulting system is highly available.

Tuning Performance Vertically

SAS Business Orchestration Services is multi-threaded and can use multiple CPU cores.

SAS Business Orchestration Services can be deployed and configured to do different things, but processing incoming transaction requests is one of the most common uses. How each transaction is routed and processed affects overall performance.

You have seen route configuration examples using Camel components and many other library classes for JMS configuration and database connection pooling. During a transaction routing, many steps can happen, depending on your business needs. These steps can include (but are not limited to) in-bound mapping, data enrichment by accessing other resources such as a database, communication with SAS Fraud Management, and outbound mapping.

Many areas are tunable, and you should observe the impact after each change. Tunings are highly environment-dependent. There is no single configuration that fits all systems.

Horizontal Scale and High Availability

Horizontal Scale and High Availability Overview

In some cases, you might scale SAS Business Orchestration Services horizontally. Instead of deploying SAS Business Orchestration Services on a more powerful machine to meet increasing requirements, you deploy SAS Business Orchestration Services to multiple, less powerful machines or nodes to meet the same throughput and latency requirements. This deployment provides high availability naturally.

In front of those nodes, there is usually a hardware or software load balancer that distributes incoming requests. Those load balancers, such as F5 and HAProxy, can be deployed in high-availability mode. It does not matter to SAS Business Orchestration Services which upstream systems are used. It can be configured just like stand-alone mode.

Resource Sharing among SAS Business Orchestration Services Nodes

The configuration for each instance of SAS Business Orchestration Services is identical to its stand-alone mode, although the following resources can be shared among SAS Business Orchestration Services:

- Resources being used for enrichment: For more information, see “Data Enrichment” on page 17.
- These shared resources can be RDBMS, Redis, and others.
- Stand-in data store
- SAF data store

Number of SAS Business Orchestration Services Nodes to Use

The number of SAS Business Orchestration Services nodes to use depends on your business needs. For high availability support, at least two nodes are needed.

For more information, contact your SAS Fraud Management support personnel.

Number of SAS Fraud Management Nodes to Use

The number of SAS Fraud Management nodes to use depends on your business needs. For high availability support, at least two nodes are needed. A typical configuration has each SAS Business Orchestration Services load balancing with failover across all SAS Fraud Management nodes. So in each SAS Business Orchestration Services, you can use a configuration such as the following:

```
2  <loadBalance id="ODE Load Balancer" inheritErrorHandler="false">
3     <failover roundRobin="true" maximumFailoverAttempts="2">
4        <exception>java.net.ConnectException</exception>
5        <exception>java.lang.RuntimeException</exception>
6     </failover>
7
8     <to id="sfm1" uri="netty4:tcp://<host1>:<port>?sync=true&encoder=#encoder&decoder=#decoder"/>
9     <to id="sfm1" uri="netty4:tcp://<host1>:<port>?sync=true&encoder=#encoder&decoder=#decoder"/>
10    <to id="sfm1" uri="netty4:tcp://<host1>:<port>?sync=true&encoder=#encoder&decoder=#decoder"/>
11  </loadBalance>
```

In this configuration, SAS Business Orchestration Services routes transaction requests to each SAS Fraud Management system. If there is a SAS Fraud Management failure, then SAS Business Orchestration Services tries the other SAS Fraud Management systems two times at most. When one or more SAS Fraud Management systems fail, but at least one SAS Fraud Management system is available, the whole system continues to operate, although at a reduced capacity.

6

SAS Business Orchestration Services Security

Security Overview

SAS Business Orchestration Services offers a wide range of configuration options for security to fit different deployment scenarios. SAS Business Orchestration Services can be configured to offer secured services, and it can be configured to consume secured services from other third-party applications. SAS Business Orchestration Services also offers different ways to protect customer credential information.

Due to the flexibility of SAS Business Orchestration Services in its service offering and the wide range of third-party applications that it supports, it is impossible to show security configurations for all of them. Some typical security configurations are covered in this chapter.

If your security concerns are not addressed in this chapter, you should consult with your SAS Fraud Management implementation team.

Configuring HTTPs

To secure SAS Business Orchestration Services when it offers HTTP, REST services, or both, complete these steps:

1 Configure SSL context parameters. Open **<*EOP_dir*>/config/spring/ol-security.xml**. By default, only TLS protocols are included. The following **camel-context.properties** file also contains a few attributes that need to be configured. They are enclosed in {{ }}.

```
<sslContextParameters id="sslContextParameters"
xmlns="http://camel.apache.org/schema/spring">
   <secureSocketProtocols>
```

```
        <!-- Do NOT enable SSLv3 (POODLE vulnerability) -->
        <secureSocketProtocol>TLSv1secureSocketProtocol>TLSv1>
        <secureSocketProtocol>TLSv1.1secureSocketProtocol>TLSv1.1>
        <secureSocketProtocol>TLSv1.2secureSocketProtocol>TLSv1.2>
      </secureSocketProtocols>
      <keyManagers keyPassword="{{passPhraseForKeyRest}}">
        <keyStore resource="{{keyStoreForRest}}" password="{{passPhraseForKeyStoreRest}}"/>
      </keyManagers>
      <trustManagers>
        <keyStore resource="{{trustStoreForRest}}"
         password="{{passPhraseForTrustStoreRest}}"/>
      </trustManagers>
    </sslContextParameters>
```

Table 6.1 Attributes in the camel-context.properties File

Attribute Name	Description
keyStoreForRest	Full path to the keystore file.
passPhraseForKeyRest	Password for the key.
passPhraseForKeyStoreRest	Password to open the keystore file.
passPhraseForTrustStoreRest	Password to open the truststore file.
trustStoreForRest	Full path to truststore file.

If you are not sure about the attributes, then talk to your security administrator who created the keystore and truststore for you.

2 Use SSL context parameters. Open **<*EOP_dir*>/config/spring/camel-context-simple.xml** and make sure that the **ol-security.xml** file has been imported.

```
<import resource="ol-security.xml"/>
```

If the **<restConfiguration>** tag is being used, and you want to turn on SSL for all REST services configured under that tag, then you can include the following code:

```
<endpointProperty key="sslContextParametersRef" value="sslContextParameters"/>
```

If your HTTP was configured using a regular `<route>` tag, then you can use the sslContextParameters tag such as the following:

```
  <route id="jettyRouteSample">
    <from uri="jetty:{{https://0.0.0.0:9098}}/sampleservice?
     sslContextParametersRef=#sslContextParameters"/>
    <transform><constant>sample service using jetty httpssample service using jetty https>
     </constant></transform>
  </route>
```

3 Verify your HTTPS after you have made these changes, restart the SAS Business Orchestration Services, and test it using either a browser or the following command from the command line:

```
Curl -k -v https://localhost:9098/sampleservice; echo;
```

Configuring Basic Authentication

HTTPS is typically used in conjunction with basic authentication. SAS Business Orchestration Services uses Spring Security to support user authentication and role-based authorization.

If you are familiar with Spring Security, then the configuration of authentication and authorization in SAS Business Orchestration Services should be straightforward.

In the `<EOP_dir>/config/spring/ol-security.xml` file, the in-memory `userDtailsService` is configured by default. You should begin with this default configuration by adding or removing users. In a production environment, typically the user information and roles are stored in other data stores such as JDBC or LDAP servers. These data stores are not provided by SAS Business Orchestration Services, but SAS Business Orchestration Services can be configured to use them. The `ol-security.xml` file contains a sample configuration for these data stores.

To configure basic authentication, complete these steps:

1 Configure `userDetailsService` and `authorizationPolicy`. The default in-memory `userDetailsService` configuration looks like the following:

```
<!-- Use in-memory user info -->
<security:user-service id="userDetailsService">
   <!--All the password are: jimspassword-->
   <security:user name="name"
    password="enter a password"
    authorities="ROLE_USER"/>
   <security:user name="name1"
    password="enter a password"
    authorities="ROLE_USER"/>
</security:user-service>
```

By default, `BCryptPasswordEncoder` is used for encoding the password, but you can change it to some other encoder if you want to.

```
<bean id="passwordEncoder" class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" />
```

The authorization policy can be configured for a role like the following:

```
<authorizationPolicy id="user" access="ROLE_USER"
   xmlns="http://camel.apache.org/schema/spring-security"/>
```

2 Use `authorizationPolicy` in routes. If you are using the `<restConfiguration>` tag in your configuration and you want to enable basic authentication, then you can include the following lines:

```
<endpointProperty   key="filtersRef" value="springSecurityFilterChain"/>
```

Inside the `<route>` tag, insert the following lines:

```
<policy ref="user">
   <!—Your pre-existing routing steps -->
</policy>
```

With the `<policy>` tag, only an authenticated user with the ROLE_USER authority can access routes within the `<policy>` tag.

Configuring Secured Communication with IBM MQ

Depending on your deployment environment, it might be necessary for you to secure communication between a SAS Business Orchestration Services server or servers and an IBM MQ server or servers.

You should have the connectivity between MQ and SAS Business Orchestration Services working before turning on SSL. Once you have the connectivity working, you or your MQ administrator can configure SSL on the MQ server.

To enable SAS Business Orchestration Services to communicate with MQ securely, complete these steps:

1 Import the certificate from the MQ server into your truststore file.

2 Configure `sslContextParameters`. This step is the same as what is described in "Configuring HTTPs" on page 47. If the same truststore is being used for both HTTPS setup and MQ connectivity, then the same `sslContextParameters` can be used.

3 Use `sslContextParameters` on the MQ connection. Open **<*EOP_dir*>/config/spring/ibmmq.xml** and remove the comments around the `SSLCipherSuite` and `SSLSocketFactory` properties.

```
<!-- Turn it on for connection to MQ with SSL on.
<bean id="sslContext" factory-bean="sslContextParameters" factory-method="createSSLContext"/>
<bean id="socketFactory" factory-bean="sslContext" factory-method="getSocketFactory"/>
<bean id="sslStoreSetter" class="com.sas.finance.fraud.ol.utils.SystemPropertiesSetter">
   <constructor-arg name="nameValuePair">
   <map>
      <entry key="javax.net.debug" value="true"/>
      <entry key="com.ibm.mq.cfg.useIBMCipherMappings" value="false"/>
   </map>
</constructor-arg>
</bean>>
-->

<--
<property name="SSLCipherSuite" value="${ibm.mq.cipherSuite}"/>
<property name="SSLSocketFactory" ref="socketFactory"/>
-->
```

Make sure that `ibm.mq.cipherSuite` has the correct value in the `camel-context.properties` file. If you are not sure what value to use, consult your MQ administrator.

4 Restart SAS Business Orchestration Services.

Configuring SSL with SAS OnDemand Decision Engine

By default, a SAS Business Orchestration Services server communicates with SAS OnDemand Decision Engine using a socket. Between them, a SAS proprietary binary data stream is transmitted.

When secure socket is enabled for SAS Fraud Management, you need to configure secure socket in SAS Business Orchestration Services so that it communicates with SAS Fraud Management securely. For more

information about secure socket configuration in SAS Fraud Management, see *SAS Fraud Management: Installation and Configuration Guide*.

To configure secure socket in SAS Business Orchestration Services, complete these steps:

1 Obtain a certificate from the SAS Fraud Management system that SAS Business Orchestration Services is going to communicate with.

2 On the machine where your SAS Business Orchestration Services is installed, import the certificate into the truststore.

You can use keytool to do it, for example:

```
keytool -import <path to trust store file> -storepass changeit -alias MyCertificate -file <certificate file>
```

3 In the `camel-context.properties` file, point the `sslKeyStoreFile` property and `sslTrustStoreFile` property to the truststore file.

4 In the `camel-context.properties` file, set `sslEnabled` to **`true`**.

5 Make sure that your route configuration uses the correct URI.

```
<to id="sfm1" uri="netty4:tcp://<host><port>?sync=true&ssl={{sslEnabled}}&passphrase
={{sslPassPhrase}}&encoder=#encoder&decoder=#decoder&trustStoreFile=#sslTsf&keyStoreFile=#sslKsf"/
```

Encrypting Credentials

SAS Business Orchestration Services provides two mechanisms to hide sensitive configuration information, such as the user name and password. One method stores the sensitive information in a metadata server, and the second method uses Jasypt.

Configure Credentials Using the Metadata Server

Credentials are typically needed for SAS Business Orchestration Services to access other systems, such as a relational database or a queue manager. To avoid putting clear-text passwords in a configuration file or files, SAS Business Orchestration Services uses the system of record database to store credentials. The **`eop_version/config/spring/ol-beans.xml`** file defines a SAS Business Orchestration Services credential store bean.

```
13  <!-- Credential store can be either OMR or local text file. -->
14  <bean id="olCredentialStore" class="com.sas.finance.fraud.ol.secure.OLSecureCredentialStore">
15     <!-- <constructor-arg name="filePath" value="${credentialFilePath}"/> -->
16     <constructor-arg name="host" value="${OMR_HOST}"/>
17     <constructor-arg name="port" value="${OMR_PORT}"/>
18     <constructor-arg name="domain" value="${OMR_DOMAIN}"/>
19  </bean>
```

The `com.sas.finance.fraud.ol.secure.OLSecureCredentialStore` also supports the use of a local credential file for scenarios where the system of record database is not available or is not used.

Once the credential store has been configured, it can be used in other bean configurations.

"Relational Data Store" on page 35 has an example for configuring a user name and password for accessing a database. The following example shows part of that configuration. `user` and `password` are configured using the `OLSecureUsername` and `OLSecurePassword` beans. In this example, the previous configured credential store is used.

```
32  <property name="user">
```

```
33     <bean class="com.sas.finance.fraud.ol.secure.OLSecureUsername">
34        <constructor-arg name="identity" value="${MYSQL_CREDENTIAL}"/>
35     </bean>
36  </property>
37  <property name="password">
38     <bean class="com.sas.finance.fraud.ol.secure.OLSecurePassword">
39        <constructor-arg name="identity" value="${MYSQL_CREDENTIAL}"/>
40     </bean>
41  </property>
```

Configure Credentials Using Jasypt

To configure credentials using Jasypt, complete these steps:

1 Modify ***EOP_dir*/config/spring/camel-context-simple.xml**.

```
<bean class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
   <!-- Use either plain, JVM property, or env property. -->
   <!-- <property name="password" value="eopphrase"/> -->
   <!-- <property name="password" value="sys:EOPENCPHRASE"/> -->
   <property name="password" value="sysenv:EOPENCPHRASE"/>
</bean>
```

2 Encrypt plaintext passwords in the `camel-context.properties` file.

   a cd ***EOP_dir*/bin**

   b export EOPENCPHRASE=<*mypassphrase*>

   c For each password that you want to encrypt, issue the following command:

   ```
   ./encrypt.sh <mypassword>

   Source text: mypassword
   Encrypted text: 2R0KsB17xPlF1djT6JErqaoWntdJliVQ
   ```

   The output from this command provides encrypted text that you can use to replace the plaintext <*mypassword*> with the encrypted text from the output. For example:

   ```
   passPhraseForKeyStoreRest=ENC(2R0KsB17xPlF1djT6JErqaoWntdJliVQ)
   ```

3 Restart SAS Business Orchestration Services.

4 (Optional) Clear your command history and unset the exported environment variable to make sure that no one can see your EOPENCPHRASE.

   To clear your command history, use the `history -c` command.

   To unset an exported environment variable, use the `unset EOPENCPHRASE` command.