# SAS® Business Orchestration Services 10.2: User's Guide

SAS® Documentation
March 14, 2023

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2022. *SAS® Business Orchestration Services 10.2: User's Guide*. Cary, NC: SAS Institute Inc.

**SAS® Business Orchestration Services 10.2: User's Guide**



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

March 2023



10.2-P1:eopug

# Contents

# What's New in SAS Business Orchestration Services 10.2

## Overview

SAS Business Orchestration Services 10.2 key efforts center on the enhanced ease of usability as it relates to the configuration of SAS Business Orchestration Services routes. The newest features include the introduction of Spring Boot, which allows for application configuration through properties files instead of using XML. In addition, data type converters have been enhanced to tackle complex data type mappings and support for all Java data types as field values with reduced use of Groovy code.

The following lists the major enhancements for this release:

- Improved the ease of use as it relates to configuration
- Integration of Spring Boot
- Enhanced data type conversion support reducing the need to use Groovy scripts for data type conversions
- Support for standards-compliant XPath and JSON pointer
- Support for ISO 8583 as a data format
- Redesigned store and forward functionality as an error handler
- Redesigned SAS On Demand Decision Engine functionality as a component

# 1

# Overview of SAS Business Orchestration Services



## Overview

SAS Business Orchestration Services is a highly customizable orchestration framework. Its features enable you to implement enterprise integration patterns of varying complexity using a set of high-level abstractions that need minimal coding. This enables you to focus on what needs to be done, rather than the how. It has a rich set of built-in features and it allows for easy extension due to a plugable architecture.

SAS Business Orchestration Services enables you to achieve quick integration and a loose coupling of systems in your organization with minimal overhead and latency. It enables you to integrate newer tools, technologies, and systems into your organization easily so that the software systems that support your business can evolve at the pace of your business. The rich set of enterprise integration patterns that are supported by SAS Business Orchestration Services enable you to optimize and enhance the data flow between systems that might have been hard before.

## Key Features

Here are the key features of SAS Business Orchestration Services:

- Support for all popular network transport and application protocols.

- Support for securing data-in-motion and data-at-rest.
- Support for all popular distributed system architectures including RESTful microservices, SOA with Web Services, event-driven architectures, and message-driven architectures.
- Data-driven routing capabilities to optimize operating costs and throughput.
- Real-time message and event processing with low latency and high throughput.
- Support for processing batch workloads.
- Automated data type conversions and manually configurable data type conversions.
- Rich set of features for data enrichment.
- Support for a wide variety of data exchange formats.
- Robust and flexible error handling and recovery mechanisms. Includes a store and forward feature for enhanced reliability and error recovery. Also includes a stand-in response feature that guarantees a valid response to the callers based on the last known good response.
- Specialized integration components for integration with other SAS software products.
- Integration components that are provided by SAS for third-party data providers.
- Lightweight process with minimal resource requirements.
- Horizontally scalable for high volume and high availability configurations.

# 2

# Installing SAS Business Orchestration Services



## Installing SAS Business Orchestration Services

You can install SAS Business Orchestration Services using one of the following methods:

- manually using Linux commands to install RPMs
- using Ansible

For more information about installing SAS Business Orchestration Services, see SAS Business Orchestration Services 10.2: Deployment Guide.

# Key Folders and Files

## Overview

Once installation is complete, the following key folders and files are located on your server:

**/opt/sas/viya/home**

- The base deployment folder for SAS Viya products. You should not modify anything under this folder.
- Subfolders have a specific SAS Business Orchestration Services area (for example: /opt/sas/viya/home/bin/boss, /opt/sas/viya/home/libexec/boss, /opt/sas/viya/home/share/boss, and so on).
- **/opt/sas/viya/home/bin/boss**
  - Contains all executable scripts that come with SAS Business Orchestration Services. This includes the boss.sh script that is used to start and stop SAS Business Orchestration Services and to check its status.
  - Referred to as the `$BOSS_BIN` variable in this document.
  - **boss.sh:** The main executable file to start, stop, and check the status of the SAS Business Orchestration Services process.
- **/opt/sas/viya/home/libexec/boss**
  - Contains the core SAS Business Orchestration Services executable (spring-boot) JAR file.
- **/opt/sas/viya/home/lib/boss**
  - Contains additional JAR files for SAS Business Orchestration Services components (for example, third-party adapters, SAS OnDemand Scoring Engine component, and so on).
- **/opt/sas/viya/home/share/boss**
  - Contains files for your use like samples and JAR files that you can use while building your configurations.
  - **/opt/sas/viya/home/share/boss/lib**
    - Contains JAR files that are part of SAS Business Orchestration Services, but you can use them while you create your configurations. Primarily contains the `boss-api.jar` file.

**/opt/sas/viya/config**

- The base configuration folder for SAS Viya products. Contains files that you can edit.
- Subfolders have a specific SAS Business Orchestration Services area (for example: **`/opt/sas/viya/config/etc/boss`**,**`/opt/sas/viya/config/var/log/boss`**, and so on).

- **/opt/sas/viya/config/etc/boss**
  - Contains all configuration files related to SAS Business Orchestration Services.
  - This folder is referred to as the `$BOSS_CONFIG` variable in this document.
  - This folder is part of the application CLASSPATH. All files, directories, and JAR files that are placed in this folder are added to the CLASSPATH automatically and can be referenced in configurations.
  - The files at the root of this folder are accessible at the root of the CLASSPATH. Files in nested folders should be referred to with the path reflecting the nested structure.
  - The `setinit.txt` file (that supplies the product license information) is expected to be in this folder by default.
  - If you are using an `application.properties` file to override the default behavior of the application, then it should be placed in this folder.
  - If you are using a `logback.xml` file for advanced configurations for logging, then it should be placed in this folder
  - **/opt/sas/viya/config/etc/boss/lib**
    - Although you are allowed to put any additional JAR files that you want to load into the application anywhere in the configuration folder, the additional JAR files should be placed in this folder.
- **/opt/sas/viya/config/var/log/boss**
  - Contains all SAS Business Orchestration Services logs.
  - **application.log**
    - Contains log output that is generated by SAS Business Orchestration Services.
  - **boss.log**
    - Contains log output generated by SAS Business Orchestration Services, but before the application is loaded.
    - Any issues that are encountered before the application is launched (for example, issues with the start-up script, inability to write to log files, launch configuration information messages, and so on) are written to this log.
    - If SAS Business Orchestration Services fails to start and no useful output can be found in the application.log file, then check this log file.
  - **commandHistory.log**
    - The `boss.sh` script makes an entry into this log every time a life-cycle command is issued to SAS Business Orchestration Services (for example, start or stop).
    - This log file primarily exists for audit purposes, if it is secured properly.

# Summary

- Executable scripts are in the **`/opt/sas/viya/home/bin/boss`** folder.

- Configuration files are in the **/opt/sas/viya/config/etc/boss** folder.
- Log files are in the **/opt/sas/viya/config/var/log/boss** folder.
- Examples and shared libraries are in the **/opt/sas/viya/home/share/boss** folder.

# 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 JDK binaries to the 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
```

# Life-Cycle Commands

## Overview

A shell script at `$BOSS_BIN/boss.sh` is used to manage the life cycle of the SAS Business Orchestration Services process. The following options can be used with the `boss.sh` script:

- `help:` explains how to use the `boss.sh` script
- `start:` starts the process
- `status:` shows the status of the process and its process ID
- `stop:` stops the process
- `tail:` sends the main process application log to the console

## Typical Option Usages

- Start SAS Business Orchestration Services (this does nothing if it is already running):

```
$BOSS_BIN/boss.sh start
```

- Stop SAS Business Orchestration Services:

  ```
  $BOSS_BIN/boss.sh stop
  ```

- Restart (stop and then start) SAS Business Orchestration Services:

  ```
  $BOSS_BIN/boss.sh stop start
  ```

- Restart SAS Business Orchestration Services and tail the `application.log` file:

  ```
  $BOSS_BIN/boss.sh stop start tail
  ```

- Restart SAS Business Orchestration Services in legacy mode:

  ```
  $BOSS_BIN/boss.sh stop start --context <path to xml file containing
  Camel Context definition, for example, spring/camel-context.xml>
  ```

# Frequently Used JVM Run-Time Parameters

SAS Business Orchestration Services is a Java process. Inside the `boss.sh` script, some JVM options and parameters are supplied when the process is started. Those can be modified and others can be added if the options and parameters fit the use case.

The JVM heap size is specified as `HEAP_OPTS` in the `boss.sh` script with the following option as the default:

```
HEAP_OPTS="-Xms1G -Xmx4G"
```

For more information, see https://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html

Set the JVM default character encoding. The following example sets the encoding to UTF-8:

```
-Dfile.encoding=UTF-8
```

For more information about the supported encodings, see Supported Encodings.

The debugging options are specified as `DEBUG_OPTS` in the `boss.sh` script with the following setting:

```
DEBUG_OPTS="-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=logs -
XX:ErrorFile=logs/server_java_error%p.log"
```

DEBUG_OPTS performs a heap dump when an out-of-memory error occurs. It also creates a log file when a fatal error occurs. These options are not enabled by default. To enable the debugging options, add it to the JVM argument.

# Running with the Default Configuration

SAS Business Orchestration Services 10.2 can run without any configuration files. The only required file in the `$BOSS_CONFIG` folder is the product license file.

To start SAS Business Orchestration Services with the default configuration, use this command:

```
boss.sh start
```

The default configuration checks to see whether the product installation was successful. Look for a statement in the **/opt/sas/viya/config/var/log/boss/application.log** file that states the following:

```
SAS Business Orchestration Services started
```

This statement verifies that the product installation was successful and that the license for the core product is valid. If any additional modules are installed, license checks for those modules are also performed at this time.

# Running in Legacy Mode

SAS Business Orchestration Services 10.2 does not require defining a Camel Context as part of the configuration. Instead, it is created automatically. If you are upgrading from an older version of the product, which required Camel Context definition as part of the product configuration, you would need to use the legacy mode to start the application. In legacy mode, SAS Business Orchestration Services turns off the automatic configuration features that are new in release 10.2, and relies completely on the configuration files to set up the application state.

To run SAS Business Orchestration Services in legacy mode, use the following command:

```
boss.sh start --context <path to the XML file that contains camel
context definition>
```

For example: `boss.sh start --context spring/camel-context.xml`

Instead of using `--context` as the option, you could also use the shorter version, `-c`, making the start-up command in the example: `boss.sh start -c spring/camel-context.xml`.

# 3

# Configuring SAS Business Orchestration Services



## Overview

To configure and use SAS Business Orchestration Services, you need some understanding of the Apache Camel framework. For more information, see Apache Camel Getting Started and other documentation on this topic.

Familiarity with Spring Framework and Spring Boot is also useful to understand bean creation and auto-configuration. For more information about Spring Framework, see Spring Framework Documentation.

While you configure SAS Business Orchestration Services, some files are expected to be in a specific location. The files should be placed in specific locations because of the default values that are set in the application. This section provides information about the files that are used to configure SAS Business Orchestration Services, the default location for these files, and whether the default location can be changed.

# Recommended Folder Structure

You should use the following folder structure to organize all the files that make up your configuration. Using this structure makes it easier to locate and work with these files as your project evolves.

`$BOSS_CONFIG` (root of the configuration folder)

This directory contains the following files:

- setinit.txt (required): Contains the required product license file. SAS Business Orchestration Services will not start or it will shut down within 24 hours if this file is missing or invalid.
- application.properties (optional): If you use this file, it should reside in this folder. All externalized property values that are used in the configuration must be specified in this file unless the default values, if any, are acceptable.
  - For more information about the properties that are used to configure technology connectors (for example, JDBC data sources, JMS and AMQP queuing servers, and so on), see "Appendix A. Common application properties" in the *Spring Boot Reference Guide*.
  - For more information about the properties that are specific to a component, see "Spring Boot Auto-Configuration."
  - Any properties that are specific to SAS Business Orchestration Services are documented in a relevant section.
- logback.xml (optional): If you use this file, it should reside in this folder. This file is used to customize logging from SAS Business Orchestration Services. For more information, see Logback Project and Apache Camel Log component documentation.

`$BOSS_CONFIG/routes`

- This folder contains only XML files that define routes.
- Each file might contain one or more routes. You can have any number of files in this folder.
- All the routes that are defined in these files are automatically initialized and are added to the application run-time state.
- You should define closely related routes in a single file and separate unrelated routes into different files (similar to how you might structure code into different classes when using a programming language).
- For a template for this file, see "Route Definitions" on page 107.

`$BOSS_CONFIG/beans`

- This folder contains only XML files that define Spring beans.
- Each file might contain one or more bean definitions. You can have any number of files in this folder.
- All the beans that are defined in these files are automatically created and are added to the application bean registry.

- You should define closely related beans in a single file and separate unrelated beans into different files.
- For a template for this file, see "Bean Definitions" on page 108.

> **TIP** If there are any XML files with bean definitions in this folder when you are running in legacy mode, SAS Business Orchestration Services tries to create all of these beans, in addition to any beans that are defined in any other files. If there is any collision with bean identifiers or other errors while instantiating the beans, then SAS Business Orchestration Services might fail to start. Check the `application.log` file for error messages if SAS Business Orchestration Services fails to start. Resolve the issues to proceed.

**`$BOSS_CONFIG/rest`**

- This folder contains only XML files that define REST API endpoints.
- Each file might contain one or more endpoint definitions. You can have any number of files in this folder.
- All the REST endpoints that are defined in these files are automatically initialized and are added to the application run-time state.
- You should define closely related REST endpoints in a single file and separate unrelated REST endpoints into different files.
- For a template for this file, see "REST Endpoint Definitions" on page 107.

**`$BOSS_CONFIG/converters`**

- This folder contains only XML files that define field mapping for data type converters.
- Each file might contain field mappings for only one data type converter. You can have any number of files in this folder.
- All the data type converter mapping files placed in this folder are loaded, validated, and added to the application run-time state at start-up.
- When errors are encountered in these field mapping files, SAS Business Orchestration Services logs an error in the application.log file and continues. Errors in these files do not cause the application to fail at start-up. You would see the errors when you invoke data type converters. You would also see the errors if the corresponding mapping file had errors.
- Changes to these files are picked up automatically. The application does not need to be restarted.
- For a template for this file, see "Converter Mapping File" on page 108.

**`$BOSS_CONFIG/groovy`**

- All Groovy language code to be loaded as scripts in your configuration should be placed in this folder.
- When referencing Groovy code from this folder, you need to include `groovy` in the path (for example, `classpath:groovy/MyGroovyUtils.groovy`).
- These are not requirements, only recommended conventions.

**`$BOSS_CONFIG/libs`**

- This folder contains any additional JAR files that you have downloaded from other sources (for example, additional Apache Camel components, JDBC drivers

from a server vendor, and so on) or your custom business logic that you implemented in Java.

- The JAR files in this folder are automatically included in the application CLASSPATH.
- If there are any collisions between the JAR files that you added to this folder and the JAR files that are packaged with SAS Business Orchestration Services, the JAR files that are packaged with SAS Business Orchestration Services take precedence.

# Spring Boot Auto-Configuration

## Overview

When running in legacy mode, all Spring Boot auto-configuration features for Camel are turned off. Because of this, you are required to create a Camel Context through XML.

When running in native mode, although SAS Business Orchestration Services has built-in integrations for Redis and JMS communications, the auto-configuration of these two features is turned off by default. All the other auto-configuration features of Spring Boot are available in both legacy mode and native mode.

## Enabling Redis Auto-configuration

To enable Redis auto-configuration, set the following property in the **application.properties** file:

```
com.sas.integration.autoconfig.redis.enabled=true
```

## Enabling JMS Auto-configuration

To enable JMS auto-configuration, set the following property in the **application.properties** file:

```
com.sas.integration.autoconfig.jms.enabled=true
```

# 4

# Logging



## Configure Using Spring Boot Properties

In the application.properties file, use the following setting to direct logging to the log file:

```
logging.file=/opt/sas/viya/config/var/log/boss/application.log
```

Use the following options to change the logging level for ROOT and for a package:

```
logging.level.root=warn
logging.level.com.sas.integration=debug
```

By default, log files rotate when they reach 10 MB. Use the following options to change the file size and the maximum number of archive log files to keep:

```
logging.file.max-size=20MB
logging.file.max-history=5
```

For more information about how to configure logging with Spring Boot, see Log Format.

## Mapped Diagnostic Context Logging

Camel Mapped Diagnostic Context (MDC) logging works with Logback in Spring Boot. The following example enables MDC logging:

```
camel.springboot.use-mdc-logging=true
```

```
logging.pattern.level=%-10.10X{camel.exchangeId} -
%-10.10X{camel.routeId}
```

# Logback XML Configuration

If you want more control over the Logback behavior, then the Logback XML configuration can be supplied in `${BOSS_CONFIG}`. For more information about how to write a Logback XML configuration file, see Chapter 3: Logback Configuration.

In native mode in the Spring properties file, specify the following:

```
logging.config=file:logback-spring.xml
```

In legacy mode, specify the following code in the Java command line:

```
-Dlogback.configurationFile=${FULL_PATH_TO}/logback-spring.xml
```

# 5

# Working with Data Types

# Overview

Whenever you are tasked to integrate two or more systems from different vendors, from different times, or to exchange data there is a high likelihood that each system expects data in a different format or it uses different data types as the payload. Even when these systems use an industry standard protocol and message exchange format, you might still encounter differences on each side (for example, the data encoding scheme that is used or custom fields are added on top of the industry standard). It might be impractical or expensive to change the source or the target systems in order to accomplish the integration. Sometimes it might not even be possible to change because it might break other systems that already use the existing interfaces. One way to deal with this is to write custom software to act as the translator between these systems. SAS Business Orchestration Services provides capabilities to tackle this task, avoiding potentially time-consuming and expensive middleware component development.

By the time at which you attempt to convert data types, it is assumed that the data to be converted is already in memory as some structured object format. It assumes that the data is not as a stream that is waiting to be read (either a network stream or a stream opened to read content from a file) or strings that need to be parsed. You would have to use the capabilities that are built into the component that you are using to read the underlying streams and convert them to an object that you can process further. Sometimes you need to use a data format to convert the incoming data (that is in the form of a string) into a structured object.

> **TIP** The only exception is while you are using the random access capabilities of the data converter to read data directly from a string using XPath expressions. This feature accepts a string as the input rather than looking for a structured data type. For more information, see "Using XPath Expressions for Random Access in XML Strings" on page 46.

Data format conversions that can be done in SAS Business Orchestration Services are explained in Chapter 6, "Working with Data Formats," on page 61. This section explains the data type conversions that can be done using SAS Business Orchestration Services.

# Data Type Converters

## Overview

SAS Business Orchestration Services provides capabilities to convert data types ranging from simple scalar values like strings, numbers, and dates all the way to complex hierarchical structures. It inherits capabilities to convert simple types from Apache Camel and Spring Framework and builds on top of those capabilities to accomplish complex conversions involving hierarchical structures. All these capabilities are provided in a way that rarely needs you to write code. Instead, you configure these complex type conversions declaratively through XML files and use them in your data processing flow through the Apache Camel component invocation mechanism.

## Basics of Using Data Type Converters

### The Field Mapping File

You create the field mapping XML file to provide information about what the converter is expected to do. The following shows a sample mapping file:

```
<?xml version="1.0"?>
<fieldMapper xmlns="https://www.sas.com/boss/fieldMapper-10.2"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://www.sas.com/boss/
fieldMapper-10.2 fieldMapper-10.2.xsd"
             sourceType="Map" targetType="Map" id="addressMapper">
    <field name="line_1">
         <sourceField name="addr1"/>
    </field>
    <field name="line_2">
```

```
            <sourceField name="addr2"/>
        </field>
        <field name="city">
            <sourceField name="city"/>
        </field>
        <field name="province">
            <sourceField name="state"/>
        </field>
        <field name="postal code">
            <sourceField name="zip"/>
        </field>
    </fieldMapper>
```

By default, SAS Business Orchestration Services looks for these mapping files in the **${BOSS_CONFIG}/converters** directory. During start-up, SAS Business Orchestration Services loads all the mapping files, validates them for errors, and configures the type converters in the system.

# Invoking the Data Type Converter

SAS Business Orchestration Services delivers enhanced data type conversion functionality through the `SAS Convert Body To` component. `sas-convert-body-to` is the scheme that is associated with the `SAS Convert Body To` component. You can invoke the `SAS Convert Body To` component by creating the following endpoint:

```
<to uri="sas-convert-body-to:?mapperId=addressMapper"/>
```

Whenever this route gets called, the Apache Camel execution engine passes the message to the SAS Convert Body To component for processing. The component performs the data type conversion.

In the example above, the argument provided to the `sas-convert-body-to` endpoint is *mapperId=addressMapper*. This is an optional parameter. It indicates to the converter that the specified mapper should be used for the conversion. Whenever `mapperId` is provided, SAS Business Orchestration Services can perform additional validation of the incoming data type and raise an exception if the message body type does not match the `sourceType` in the mapping file.

The `sas-convert-body-to` endpoint accepts another optional parameter named `targetType`. When the `targetType` parameter is used alone, SAS Business Orchestration Services automatically searches an internal registry for a converter that is capable of converting the incoming data type to the target data type and use it for the conversion. It raises an exception if no such type converter is available. More information about how SAS Business Orchestration Services performs this search is explained in later sections. For now, either a `mapperId` or `targetType` parameter must be provided on a call to the `sas-convert-body-to` endpoint.

# Inside the Data Type Converter

When you invoke the `sas-convert-body-to` endpoint with the mapping file shown above, the producer that is associated with the endpoint gets the `Exchange` object routed. The processing that is built into the producer is responsible for transforming

the contents of the `Exchange` object according to the functionality that is provided by the endpoint. In this case, the functionality that is provided is to convert the type of the object in the message body to the `targetType` that was configured in the mapper with the ID that was provided on the endpoint, which is `addressMapper`.

The `addressMapper` ID has `Map` as the targetType and it is expected to handle input that is also a `Map`. But there are differences in the names of the keys that are used in the source `Map` and the target `Map`. At run time, the converter checks that the message body is a `Map` and creates another `Map` instance as the output. The converter fails with an error if the source object is not a `Map`.

> **TIP** The `sourceType` attribute on the `fieldMapper` element is optional.

After the source and target objects are in place, the converter processes each field in the order that they are defined in the mapping file. If there are more fields in the source or target object, then the converter ignores them. This enables you to transfer only a partial set of fields from the source object, if that is what you want to do.

The following discusses how fields are processed by walking through the steps for one field. For the target field `line_1`, the source value comes from the `addr1` property in the source object. There are no data types on the `field` or `sourceField` elements. This means that you want the converter to read whatever is the value for this property in the source object and automatically convert it to the data type that is expected for the `line_1` property in the target object. In this specific case it might be reasonable to assume that both `line_1` and `addr1` are strings. So in this case, the value is copied from the source `Map` to a property in the target `Map`, but the property names are different. Even if the value for the `addr1` property was, for example, a byte array, the mapping file could look exactly the same. When the source and target data types are different for any field, the converter automatically attempts to convert the value by finding additional converters as needed. If none of the available converters in the system are capable of doing the conversion that is needed, then the converter invocation fails with an exception.

SAS Business Orchestration Services comes with lot of built-in converters to handle commonly encountered data type conversions automatically. These converters are either available in SAS Business Orchestration Services or are inherited from the underlying Camel and Spring Frameworks. No additional configuration is needed to use these automatic conversions.

In this introductory example mapper, the other fields also have a similar function of copying a value from source object to the target object where the property names might be different and potentially some data type conversion is being done automatically. The full power of the `SAS Convert Body To` component is discussed in later sections.

**TIP** Although the converter can look at the data type's source and target objects and attempt to convert the value automatically, if there is any chance that other people are looking at the mapping file, they might find it helpful to know the data type of any field. You should add the `type` attribute to provide clarity.

In addition to providing clarity, having this additional information in the mapping file enables the converter to also perform some additional checks and help identify misconfigurations. For example, if the incoming type was configured as a date, but a string was received, the converter fails. It would not take whatever string was provided and convert it to a date and proceed silently, even if the string value could be parsed into a valid date value.

Keep your mapping file as simple as possible, but clarify the expectations and intent with additional information when it is helpful.

# Convert Body To Processor versus the SAS Convert Body To Component

Apache Camel provides a `Convert Body To` processor that converts the message body to a different data type. You invoke that functionality with the following syntax:

```
<convertBodyTo type="java.util.Map"/>
```

You can continue to use this functionality in your routes if there is a converter that can do what you need. If the message body contains a simple data value or special objects like `InputStream` or `ByteBuffer`, then using a `Convert Body To` processor is the only option. The `SAS Convert Body To` component does not handle the same use cases as the Camel `Convert Body To` processor. They are designed to be complimentary. The `SAS Convert Body To` component uses Camel type converters when necessary to provide the seamless integration between the two.

The key difference between the two mechanisms is that the `SAS Convert Body To` component deals with the inner structure and the properties of the object being converted. The differences can be shown using the earlier address mapper example.

In the address mapper example, a `Map` was being converted to another `Map` with different keys and potentially different value types. If you use `<convertBodyTo type="java.util.Map"/>`, then the processor sees that the source (message body) type and target type are the same and returns the object unchanged. The only way to perform the conversion that is needed while using the `Convert Body To` processor is by writing an implementation of `org.apache.camel.TypeConverter` interface and loading it into the system.

Even if the source and target types do not match exactly (for example, from a `java.util.LinkedHashMap` to a `java.util.HashMap`), and you try using `<convertBodyTo type="java.util.HashMap"/>`, it would still not accomplish the conversion that you need. The `Convert Body To` processor definitely gives an instance of `java.util.HashMap` as output, but it includes all the keys and values exactly like what they were in the source object. Compare this to the processing done by the `SAS Convert Body To` component that was explained in the previous section.

Another key restriction in Camel Type Converter architecture is that there cannot be more than one converter registered between a source type and target type. The `SAS Convert Body To` component removes this restriction and lets you register as many converters as you need and provides an unambiguous way for you to specify which converter to use in a given context.

# Invoking the sas-convert-body-to Endpoint with a Target Type

## Overview

Instead of invoking the `sas-convert-body-to` endpoint by providing the `mapperId` parameter, you can invoke it by specifying `targetType` as the parameter if your target type differs from source type.

> **TIP** If the `targetType` and `sourceType` are the same, then providing the `mapperId` is required.

Here is how you invoke the `sas-convert-body-to` endpoint by specifying a `targetType`.

```
<to uri="sas-convert-body-to:?targetType=Map"/>
```

Invoking the converter with a `mapperId` is less ambiguous, but is less flexible as well. Sometimes it becomes cumbersome to keep track of all the IDs that are used in the mapping files when there are lot of them in the project. Another disadvantage is that if you ever change the ID in any file, then you need to search all the places where it is used and change them accordingly.

In addition to the disadvantages mentioned above, there could be times where you cannot determine the mapper to use ahead of time, because the data could be coming to the converter from different execution paths and the mapper might need to be selected based on the type of the incoming data.

In this simple example, assume that you are working with a legacy order processing system and you have a route set up to convert data that is coming in as a `Map`. You need to transform this order data into an `Order` bean that is expected by your order fulfillment system. You created the necessary converter that takes the `Map` as input and transforms it into an `Order` bean and uses that converter in one place in your route.

Also, a mobile channel was added by the sales team that uses a third-party platform to capture order data, along with some additional fields that are specific to mobile orders. Assume that this data comes in the form of a representation bean that is provided by the mobile order platform API. Although you have the option to read this data as a `Map`, the fields are also different. So you would need to create another converter to perform this transformation to an `Order` bean from the mobile representation bean.

Assuming that you want to do the conversion to an `Order` bean at a single location in your route, and since you have two mappers to use for this conversion based on which channel the order came from, you cannot hardcode a single mapper ID while invoking the conversion functionality. Instead, you need to let the endpoint choose the appropriate converter to use by specifying the target type. The converter endpoint looks for the appropriate converter to use based on the target type and the type of the source object in the message body.

# Converter Lookup Rules

When the `sas-convert-body-to` endpoint is invoked with the `targetType` parameter, the endpoint searches for a matching converter in an internal registry.

The rules applied during the search are as follows:

**Note:** The following rules apply only when the target type is not the same as the source type.

1. Find converters with a target type that exactly matches the target type provided as the parameter.
2. If rule 1 returns no matches, then find converters with the target type that is assignment-compatible with the target type provided as the parameter.
3. Out of the matches in rules 1 or 2, find converters with the source type that exactly matches the message body type.
4. If rule 3 returns no matches, then find converters with the source type that is assignment-compatible with the message body type.

Exactly one converter should remain after applying all the rules.

If more than one converter matches the message body type and the given target type, then the endpoint fails with an exception. Under such circumstances, the best option might be to change the route logic to tackle the conversion at a different place so that you can use the `mapperId` parameter instead of the `targetType` parameter.

# Location of the Mapping Files

By default, SAS Business Orchestration Services loads, validates, and registers converters based on the XML mapping files that are placed in the **`${BOSS_CONFIG}/converters`** directory. The location for these mapping files can be changed by setting the following property in the **`application.properties`** file. Make sure to place the directory containing the mapping files in the **`${BOSS_CONFIG}`** directory.

```
sas.integration.converter.mapper-files-location=<path to mapping file
directory relative to ${BOSS_CONFIG} directory>
```

> **TIP** A restart is required before the changes to the mapping file location go into effect.

# A Deep Dive into the Field Mapping File

## Overview

A field mapping file explains the operations that are performed by the data type converter. A converter that is defined through these files always operates on a single instance of the given data type. Therefore, the source object and target object of a converter are always going to be either a `Map` object or a Java bean type that models some domain object.

Although SAS Business Orchestration Services handles collections of objects, that operation is performed at a higher level and is explained in later sections. Mapping files focus on the steps that are involved in transforming a single instance of the source type to a single instance of the target type.

## The fieldMapper Element

The `fieldMapper` element is the top-level element in the mapping file. The XML namespace declarations in this element associate the default namespace to the `fieldMapper-10.2.xsd` file. (For more information, see the `addressMapper` example in "The Field Mapping File" on page 17.)

The `fieldMapper` element accepts the following attributes:

- `id` (required): a unique identifier. It must start with a letter and can contain letters, digits, and underscores. More special characters are allowed, but it is safer to stick with letters, numbers. and underscore. For more information, see Declaring Namespaces.
- `sourceType` (optional): the data type that is expected for the source object (that is, the body of the message that is being routed). If a value for this attribute is provided, then additional type checks are performed to make sure that the source object is of this type. If this attribute is omitted, then the converter accepts any incoming object as source, and tries to read source fields from that object. You might want this behavior when different types of objects with the same set of fields come in as the source object. Any fully qualified type name is valid. Typically, the source is a Map file or a Java bean. For more information, see "Providing Type Information" on page 36.
- `targetType` (required): the data type that is produced by the converter as the output. This attribute accepts the following:
  - only the `Map` interface type

- any type that implements `Map` or its sub-interfaces
- only string keys are allowed while value can be of any type when using a `Map`
- any concrete Java Bean type
- no wildcards are allowed in the type (for example, Map<*String, Number*> is valid, but not Map<*String, ? extends Number*>)

Inside the `fieldMapper` element, you are required to have one or more field on page 24 elements. Character data is not allowed inside this element.

# The field Element

The `field` element declares a field in the output object that is produced by the converter. At least one `field` element is required inside the `fieldMapper` element.

The `field` element accepts the following attributes:

- `name` (required): when using a `Map` as the target type for the converter, it can be any string.

  When using a Java bean as the target type for the converter, the name must be a valid Java identifier (that is, it must start with a letter, and can contain letters, numbers, and underscores).

  It can be a nested field name, mapped field name, or indexed field name.

- `type` (optional): data type of the value that is expected to be in this field. Can be a scalar type (for example, a number, string, date/time, and so on), Map, bean, collection, or an array.

  If a Java bean is used as the target type and the type is not provided, then the converter uses the introspector on the bean to determine the field type and invokes any converters that are available.

  If a Map is used as the target type, then no conversion is attempted and the object that is returned by the source for this field is set as the value.

  For more information, see “Providing Type Information” on page 36.

- `pattern` (optional): the pattern to use to convert output values into a string.

  Can be used only when the type is number (or its subtypes in Java), LocalDate, LocalDateTime, and ZonedDateTime.

  For more information, see “Using Patterns to Parse and Format Values” on page 31.

- `default` (optional): If the value that comes from the source of this field is null, then the default value is used. Must be of the type specified for the field.

  `default` accepts property placeholders and bean references.

  For more information, see “Configuring Default Values for Fields” on page 34.

- `allowZeroLengthString` (optional): For string fields, determines whether a zero-length string is considered a valid value while deciding whether to return the default value, if a value has been set for the default attribute.

  The default value for this attribute is **`False`** (that is, a zero-length string is not considered equivalent to a NULL value).

Set this attribute to `True` if a zero-length string value must be considered a valid value and returned as is.

This attribute has no effect when the field type is not a string or when a default value has not been specified.

- `mapperId` (optional): if the field value is a bean or a `Map` and another mapper needs to be invoked to convert the source value to the value type that is accepted by this field, then the converter that is associated with this ID is invoked.

  When the `mapperId` attribute is not provided and the field value needs conversion, the converter lookup process explained above tries to find an appropriate converter automatically.

Inside the `field` element, one `sourceField`, `constant`, or `custom` elements must be nested. These elements provide the source value that will be transformed and set it in this field.

Inside the `field` element, one `sourceField`, `constant`, or `custom` elements must be nested. These elements provide the source value that will be transformed and sets it in this field. Character data is not allowed inside the `field` element.

# The sourceField Element

The `sourceField` element declares which field in the input object should be read as the source value for the `field` element in which it is nested.

The `sourceField` element accepts the following attributes:

- `name` (required): when using a `Map` as the source type for the converter, this value can be any string. When using a Java bean as the source, the name must be a valid Java identifier (that is, starts with a letter, and can contain letters, numbers, and underscores). Can be a nested field name, mapped field name, or indexed field name
- `type` (optional): expected data type of the value in this source field. Can be a scalar type (a number, string, date/time, and so on), `Map`, bean, iterable, collection, or an array. For more information, see "Providing Type Information" on page 36.
- `pattern` (optional): pattern to be used to parse input string values to the type specified. Can be used only when the type is a number (or its subtypes in Java), LocalDate, LocalDateTime, or ZonedDateTime. For more information, see "Using Patterns to Parse and Format Values" on page 31.

Character data is not allowed inside the `sourceField` element.

# The xmlQueryNamespace Element

The XPath expressions that are used in the xmlQuery on page 27 elements need to have prefixes associated with all namespaces that are used in the XML input data - even for the default namespace, if any. Associating a prefix for namespaces is necessary under the following circumstances:

- When a default namespace is used and you want to access data from those elements and attributes in your XPath expressions.
- When more than one default namespace is used for different element tags in hierarchical data.
- When the prefixes that are defined in the XML file have duplicates (similar to using a different default namespace for elements).
- When the namespace prefixes that are defined in the XML data are verbose or confusing, and you want to associate alternate prefixes to the namespaces.

The `xmlQueryNamespace` element enables you to associate a prefix to the namespace URI. Once a prefix is defined, you can use that prefix in any `xmlQuery` elements in scope. You can define as many `xmlQueryNamespace` elements as necessary.

The `xmlQueryNamespace` elements can be children of the `fieldMapper` element or any `sourceField` element if XML string content is the value of the `sourceField` and an `xmlQuery` element is nested inside the `sourceField` element. Defining an `xmlQueryNamespace` element nested in a `sourceField` element without an accompanying `xmlQuery` element would be reported as an error.

> **TIP** All `xmlQueryNamespace` elements that are children of a `fieldMapper` element must be placed before any `initializer` and `field` elements. If `xmlQueryNamespace` elements are nested inside a `sourceField` element, then they must appear before the `xmlQuery` element. You can define any number of `xmlQueryNamespace` elements in the `fieldMapper` element, `sourceField` element, or both.

The order of precedence for XML namespace prefixes defined is as follows (with the highest precedence first):

- Namespace prefixes defined by `xmlQueryNamespace` elements as direct children of a `sourceField` element (considered local scope).
- Namespace prefixes defined by `xmlQueryNamespace` elements as direct children of the `fieldMapper` element (considered global scope).
- Namespace prefixes defined inside the XML content itself (considered default scope).

The `xmlQueryNamespace` element accepts the following attributes:

- `prefix` (required): prefix to associate with a given namespace URI. It cannot be an empty string. It must start with a letter and can contain letters, digits, and underscores. There are more special characters allowed, but it is safer to just use letters, numbers, and underscores. For more information, see Declaring Namespaces.
- `uri` (required): The URI to associate with the given prefix. It cannot be an empty string. It must be a syntactically valid URI.

# The xmlQuery Element

The `xmlQuery` element declares the XPath expression to use to extract atomic values from a string.

The `xmlQuery` element can act as the source of a field value when it is nested directly inside a `field` element, or it can be used to process string data of a `sourceField` element. When it is nested inside a `field` element, the input data for the data type converter is expected to be a string with XML content in it. When it is nested inside a `sourceField` element, the value returned by the `sourceField` element is passed as input to the `xmlQuery` element nested inside it (that is, the `sourceField` value is the XML content to be processed).

The `xmlQuery` element accepts the following attributes:

- `expression` (required): a valid XPath expression.
- type (optional): the data type that is expected as the output of the XPath expression validation. String is the default type if no value is specified for this attribute. Only the following data types are accepted:
  - String: String is the default type if no value is specified for this attribute.
  - List<String>: If a list of values is expected as the result of evaluating the expression, then the type must be specified as a List<String>.
  - A numeric type or a List of numeric type, when using a `pattern`.
  - A date or time type or a List of date/time types, when using a `pattern`.
- `pattern` (optional): a pattern to use to parse the string values that are returned by the XPath expression evaluation to the type specified. This attribute can be used only when the type is number (or its subtypes in Java), LocalDate, LocalDateTime, or ZonedDateTime. For more information, see "Using Patterns to Parse and Format Values" on page 31.

# The jsonQuery Element

The `jsonQuery` element declares the JSON Pointer expression to use to extract atomic values from a string. The usage is similar to the `xmlQuery` element, but in the context of JSON content.

The `jsonQuery` element can act as the source of a field value when it is nested directly inside a `field` element, or it can be used to process string data of a `sourceField` element. When it is nested inside a `field` element, the input data for the data type converter is expected to be a string with JSON content in it. When it is nested inside a `sourceField` element, the value returned by the `sourceField` element is passed as input to the `jsonQuery` element nested inside it (that is, the `sourceField` value is the JSON content to be processed).

The `jsonQuery` element accepts the following attributes:

- `expression` (required): a valid JSON pointer expression.

- type (optional): the data type that is expected as the output of the XPath expression validation. String is the default type if no value is specified for this attribute. Only the following data types are accepted:
  - String: String is the default type if no value is specified for this attribute.
  - List<String>: If a list of values is expected as the result of evaluating the expression, then the type must be specified as a List<String>.
  - A numeric type or a List of numeric type, when using a `pattern`.
  - A date or time type or a List of date/time types, when using a `pattern`.
- `pattern` (optional): a pattern to use to parse the string values that are returned by the JSON Pointer expression evaluation to the type specified. This attribute can be used only when the type is number (or its subtypes in Java), LocalDate, LocalDateTime, or ZonedDateTime. For more information, see "Using Patterns to Parse and Format Values" on page 31.

# The constant Element

The `constant` element declares a constant value to be used as the source value for the `field` element in which it is nested.

The `constant` element accepts the following attributes:

- `type` (optional): expected data type of the value in this constant. Can be a scalar type (a number, string, date/time, and so on), `Map`, bean, iterable, collection, or an array. For more information, see "Providing Type Information" on page 36.
- `pattern` (optional): a pattern to be used to convert output values to a string. This attribute can be used only when the type is number (or its subtypes in Java), LocalDate, LocalDateTime, or ZonedDateTime. For more information, see "Using Patterns to Parse and Format Values" on page 31.

The value for the constant must be provided as a nested value within the element. When using property placeholders or bean references, the same mechanism that is used to supply default values are also used for constant values. For more information, see "Configuring Default Values for Fields" on page 34.

# The custom Element

While transforming data from one type to another, there might be situations where you need to perform more complex operations. The `custom` element of the mapping file provides this capability. Using the `custom` element, you can process any available data elements to calculate the value for a target field. You can use this element to split values, combine values, or do any sort of complex calculations that are impossible to express using just XML. The `custom` element provides the capability to write the processing logic using the Groovy language, which implicitly allows the ability to use any Java library. The `custom` element enables you to access most bean, property, source, or target fields that are available in the converter.

The following sections explain the XML structures that enable you to do this.

The `custom` element accepts the following attributes:

- `name` (required): The name for the `custom` element. This name is used as the unique name for the code block in the execution environment.
- `type` (optional): The data type of the value that is expected as the output of the `custom` code. For more information, see "Providing Type Information" on page 36.

  If the type of the result that is produced by the `custom` code does not match this type, then automatic conversion is attempted.

  If no type is specified, then the result that comes out of the code is passed to the target field as is.

The `custom` element can have two nested elements, the `param` element and the `groovy` element, The `param` element is optional, and can be repeated as many times as necessary. The `groovy` element is required and there can be only one nested inside a `custom` element.

For examples of how to use the `custom` element, see "Using the custom Element" on page 53.

# The param Element

The `param` element defines a parameter that needs to be passed into the Groovy code block that is defined inside the `custom` element. The `param` element must be nested within a `custom` element.

The `param` element accepts the following attributes:

- `name` (required): name for the parameter. The parameter is accessible within the code using this name.
- `type` (optional): data type of the parameter value.

  If the actual value type does not match this type, then automatic conversion is attempted.

  This is the type that is used for the parameter in the code.

  If this attribute value is not specified, then the type assigned to the parameter is `java.lang.Object`.
- `valueSourceType` (optional): Indicates the source of the value to be passed for this parameter. Acceptable values for this attribute include the following:
  - SOURCE_FIELD (default): indicates that the value must be read from the source object that is passed into the converter.
  - TARGET_FIELD: indicates that the value must be read from the target object that is being populated.

    Provides the capability to read target fields that have been processed already.

    Fields are processed in the order that they are defined in the mapping file. Any field that is defined before this field that contains this `custom` element can be accessed.
  - PROPERTY: indicates that the value of a property defined in the system must be passed as the parameter value.

- BEAN: indicates that a bean that is defined in the registry must be provided as the parameter value.

If no value is provided for this attribute, then SOURCE_FIELD is used.

- `valueSource` (required): The actual source of the parameter value.

  If the `valueSourceType` attribute is set to either SOURCE_FIELD or TARGET_FIELD, then this value is treated as the field name in the source or target object, respectively.

  If the `valueSourceType` attribute is set to PROPERTY, then this value is treated as the full name of the property. The {{ and }} around the property name are optional.

  If the `valueSourceType` attribute is set to BEAN, then this value is treated as the ID that is assigned to the bean in the registry. The # character in front of the bean reference is optional.

The `param` element is optional. There can be `custom` elements without any `param` elements defined. You can define more than one `param` element inside a `custom` element if you need to pass more than one data item as a parameter into the Groovy code block. For examples on how to use the `param` element, see "Using the custom Element" on page 53.

# The groovy Element

SAS Business Orchestration Services provides the capability to write any arbitrary code inside the `custom` element using the Groovy language. The `groovy` element acts as the container for the Groovy code that is nested within the `custom` element.

The `groovy` element does not accept any attributes. The Groovy code must be nested within the `groovy` element as character data.

Use of a CDATA block within the `groovy` element is not required but highly recommended. You can avoid using it when the code is simple and does not contain any characters that are treated as special characters while parsing XML content. When in doubt, use a CDATA block as it avoids misinterpretation of code as XML content by the parser. Use of a CDATA block indicates to the XML parser that the content within that block should not be processed as XML content, and instead passes it to the program using the XML file as is. This prevents the XML parser from treating special characters like the ampersand (&), braces ({ and }), angle brackets ([ and ]), and so on like special XML characters and processing them in a way that breaks the Groovy code syntax.

At run time, the script that is nested inside the `groovy` element, except import statements, is considered as a single method and the value that is returned by the method is used for downstream processing by the converter. If there is no explicit return statement, the Groovy code returns the value of the last expression that was executed as the return value for the method.

In addition to the parameters declared through the `param` elements, there are two built-in parameters that are provided automatically. The first one is the Camel Exchange instance, accessible through the parameter name `exchange`. The second is a parameter named `cache` of type Map<String, Object>. You can use the `exchange` parameter to read any of the properties that are stored in the Camel Exchange instance. The `cache` parameter enables sharing data between different Groovy script elements.

**TIP** As a result of having `exchange` and `cache` as two automatic parameters available inside the Groovy code, you cannot define any local variables with the same name. A Groovy compilation error is reported when the mapping files are loaded if local variables with either of these names are declared inside the `groovy` element.

**TIP** Any import statements that are needed to avoid specifying fully qualified class names multiple times in the code can be placed at the beginning of the code block. Lines at the beginning of the code block starting with the keyword `import` are processed as import declarations. The first non-import statement terminates this processing (that is, if you place any other lines before or nested within the import declarations, even comment lines, lines starting from that non-import statement are considered part of the executable code). Empty lines between `import` statements are ignored.

The `groovy` element is required and there can be only one occurrence of this element inside of a `custom` element. For examples on how to use this element, see "Using the custom Element" on page 53.

# Using Patterns to Parse and Format Values

In the course of transforming objects of one data type to another, there might be a few occasions where data is a string on one side and a typed object on the other side. SAS Business Orchestration Services provides the capability to apply patterns to use while parsing incoming data from a string to a typed object, and to use while formatting output as a string from a typed object.

Here are various conversions that can be done from string type using patterns:

***Table 5.1*** *String Conversions*

| Conversion | Description | Valid Pattern to Use | Examples |
|---|---|---|---|
| String → LocalTime | Used to convert a string input into a `java.time.LocalTime` object.<br>String input should be time values with no time zone information. | java.time.format.DateTimeFormatter | '14:15:30' (for HH:MM:SS in 24–hour format)<br>Fractional seconds are allowed up to nanoseconds, like '14:15:29.999999999' |
| String → LocalDate | Used to convert string input into a `java.time.LocalDate` object.<br>String input should be date values | java.time.format.DateTimeFormatter | '2020-12-24'<br>'12242020'<br>24th December 2020' |

| Conversion | Description | Valid Pattern to Use | Examples |
|---|---|---|---|
| | with no time or time zone information. | | |
| String → LocalDateTime | Used to convert string input into a java.time.LocalDateTime object.<br>String input should be date values with no time or time zone information. | java.time.format.DateTimeFormatter | '2020-12-24T14:15:30' |
| String → ZonedDateTime | Used to convert string input into a `java.time.ZonedDateTime` object.<br>String input should be date and time values with time zone information. | java.time.format.DateTimeFormatter | '2020-12-24T14:15:30+04:00 America/New_York' |
| String → Date | Used to convert xtring input into a `java.util.Date` object. | java.text.SimpleDateFormat | '2020-12-24'<br>'2020-12-24T14:15:30+04:00' |
| String → Short, Integer, Long, or BigInteger | Used to convert string input into `Short`, `Integer`, `Long`, or `BigInteger`.<br>Attempting to parse decimal values into these integer types results in an error because it causes information loss. If information loss is acceptable, then use a more generic number as the target type while parsing, and then let the system do narrowing conversion to target type. | java.text.DecimalFormat | |

| Conversion | Description | Valid Pattern to Use | Examples |
|---|---|---|---|
| String → Float, Double, or BigDecimal | Used to convert string input into a `Float`, `Double`, or `BigDecimal`.<br><br>While parsing as `BigDecimal`, special decimal values NaN and Infinity (positive and negative) result in a `TypeConversionException`. | java.text.DecimalFormat | |
| String → Number | Used to convert string input into a Number.<br><br>The actual returned type would be a subclass of number based on the type of input. | java.text.DecimalFormat | |

> **TIP** When converting from a string type to a numeric type using patterns, SAS Business Orchestration Services uses BigDecimal while parsing, and then converts it to the target type. If data loss is detected when converting a given string value into a target numeric type, then a `TypeConversionException` is raised.

A corresponding set of formatted output operations are available to convert a given data type to a string using a pattern. The same rules as above apply to the patterns in each case.

When a `pattern` attribute is used on a `sourceField` element or a `constant` element, it assumes that the actual value is a string. Hence, the type specified on the elements should indicate the type to which the string has to be parsed. The type also drives the validation checks on the pattern.

Similarly, when a `pattern` attribute is used on a `field` element, it assumes that the output value would be a string. The type specified on the `field` element should indicate the type of the value before formatting. Here also, the type drives the validation checks on the pattern.

Using `String` as the type when using patterns is invalid because the operation to parse or format a string would be invalid.

# Configuring Default Values for Fields

## Overview

Sometimes data that is processed by SAS Business Orchestration Services might have gaps in it. These gaps manifest as NULL (a special concept in programming languages that means that no value is available) values or empty string values. To avoid errors downstream and incorrect results, these missing values should be handled appropriately. The default attribute on the `field` element of the mapping file enables you to specify a value to be used whenever missing values are encountered.

Source data for the `field` element might come from a `sourceField`, `constant`, or `custom` elements. When the default attribute is used on the `field` element, regardless of the source, if the value is NULL, the default value is used as the value for the (target) field.

```
<field name="line_2" default="NA">
    <sourceField name="addr2"/>
</field>
```

Setting `default="NA"` specifies that whenever the source value for the `line_2` field (which in this case is the `addr2` field on the source object) is NULL, `NA` should be used as the value for the field.

## The Difference between a NULL and an Empty String

When dealing with string values, for all practical purposes, a NULL value might seem to be equivalent to an empty string (a string value with no characters in it and has zero-length). But they are not the same when a program is processing the data. In some instances, a NULL value might not be acceptable but an empty string is, while in other cases, they must be treated the same way. Under these circumstances, you could use the `allowZeroLengthString` attribute to fine-tune whether empty string values are treated differently than NULLs. The `allowZeroLengthString` attribute defaults to False, meaning that an empty string is considered equivalent to NULL and the default value gets used when the value is NULL or an empty string. If you want to allow an empty string as a valid value, then set this attribute to True.

## Effect of Pattern on Default Value

A pattern can be used to format the output value on a field as a string according to the given pattern. If the default value is provided along with pattern, then the default

value should also follow the given pattern. The system validates the default value using the pattern and the type specified for the field and it results in an error at run time if the default value is not formatted as expected.

The following example shows how you can provide a default value for a field whose type is `java.time.LocalDate` and whose values are being converted into formatted strings in the output. In this example, a settled date shows up as 20201231 in the output if the source field value is NULL. The additional validation performed by the system makes sure that 20201231 is a valid `LocalDate` value when parsed using the given pattern. SAS Business Orchestration Services parses this string once into a `LocalDate` instance and uses that value as the actual default value instead of the string that is provided in the mapping file.

```
<field name="settledDate" type="LocalDate" pattern="yyyyMMdd"
default="20201231">
    <sourceField name="transaction_settled_date"/>
</field>
```

# Referencing a Property Value in Default Value

When the same default value needs to be used for multiple fields in a mapping file or even across multiple mapping files in the system, it might be error prone to specify the default values on each field separately and keep them in sync throughout the system life cycle. In these situations, you should define a property in the `application.properties` file and reference the property in the default value.

In the `application.properties` file, if you have a property named `empty.address.line` defined as `empty.address.line=NA`, then you could reference this property in the default attribute value as shown below, using the standard Camel property reference notation.

```
<field name="line_2" default="{{empty.address.line}}">
    <sourceField name="addr2"/>
</field>
```

# Passing a Bean Instance as the Default Value

All the information that is passed through the mapping file has to work within the confines of the XML specification, and XML files allow only string values for attributes. But the type of the value that you need to use as the default might not always be a string or something that could be parsed into the type that you need. In order to handle these situations, the default attribute value can accept a bean reference, using Camel bean reference notation of '#' in front of the bean identifier.

The following example uses a custom bean that was defined in the system and registered in the Spring bean registry (typically done through bean configuration files or Java code), named `com.mybank.transaction.Party` (a bean of type `com.mybank.transaction.Party`) with the necessary data attributes. While

processing transactions, if there is no intermediary provided on the transaction, you could use this pre-defined Party as the default.

```
<field name="intermediary" type="com.mybank.transaction.Party"
default="#defaultIntermediaryParty">
    <sourceField name="party_3"/>
</field>
```

# Providing Type Information

## Overview

The data type converter in SAS Business Orchestration Services tries to analyze the input and output field values to determine the types automatically. Sometimes it might be ambiguous given the information that is available. In these instances, the converter reports an error and fails. To resolve ambiguity in these situations, the converter needs additional input in the mapping files via the type attributes that are available on certain elements.

If you are not familiar with the types of type names that are allowed in the Java programming language, see Java Language Specification.

Generally, the `type` attribute accepts any fully qualified Java type name. When it comes to output objects that are created by the converters, either as the final output or as output of an intermediate step, abstract type names are not allowed, including interface types and abstract classes.

Here are a few examples of fully qualified type names:

- Simple type names like java.lang.Integer.
- Type names with generics like java.util.Map<java.lang.String, java.lang.Object>.
- Type names with wildcards in generics like java.util.Map<java.lang.String, ?>, java.util.Map<java.lang.String, ? extends Number>.
- Type names for arrays like java.util.String[].

## Short Class Names

Though fully qualified names are required as the `type` attribute values in the mapping file, for convenience and brevity, the following commonly used types have a shorter version that can be used instead of the fully qualified names.

- BigDecimal for java.lang.BigDecimal
- BigInteger for java.math.BigInteger
- boolean or Boolean for java.lang.Boolean
- Date for java.util.Date
- double or Double for java.lang.Double

- float or Float for java.lang.Float
- int or Integer for java.lang.Integer
- Iterable for java.util.Iterable
- List for java.util.List
- LocalDate for java.time.LocalDate
- LocalDateTime for java.time.LocalDateTime
- LocalTime for java.time.LocalTime
- long or Long for java.lang.Long
- Map for java.util.Map
- Number for java.lang.Number
- Object for java.lang.Object
- Set for java.util.Set
- String for java.lang.String
- short or Short for java.lang.Short
- ZonedDateTime for java.time.ZonedDateTime

These shorter names can be used anywhere that a fully qualified name is valid (for example, even in nested generic declarations like Map<String, List<? extends Number>>). If you are using fully qualified names, that same declaration would be java.util.Map<java.lang.String, java.util.List<? extends java.lang.Number>>.

## Type Restrictions for Output Values

Whenever the converter needs to produce output values, it needs to create brand new instances of the output types programmatically. The following are a few restrictions when specifying types for output values:

- They cannot be abstract types, including interfaces like Iterable and abstract classes like `java.util.AbstractList`.
- They cannot be types with wildcards like `Map<String, ?>`.
- There must be a no-argument constructor for the class to be instantiated.

These rules apply to types from the JDK or custom classes.

# Field Mapper XML Schema

While you are working with XML files, you might want to have the context-sensitive help and use validation of the content that comes when the XSD schema that the XML file needs to adhere. You need to work in an XML editor or IDE that provides such context-sensitive help and validation, along with the XSD schema for the mapping file.

The XSD file for the mapping files, `fieldMapper-10.2.xsd`, is included in the `boss-api.jar` file. This JAR file is included with your SAS Business Orchestration

Services installation. The JAR file is available in the **${BOSS_SHARE}/lib** folder. Placing this JAR file in the classpath of your project, along with the necessary namespace declarations in the XML, as shown below, should enable the tool to provide context-sensitive help.

```
<fieldMapper xmlns="https://www.sas.com/boss/fieldMapper-10.2"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://www.sas.com/boss/
fieldMapper-10.2 fieldMapper-10.2.xsd"
             ... >
```

In case your editor needs just the XSD file and not a JAR file, unzip the `boss-api.JAR` file. The `fieldMapper-10.2.xsd` file is in the **converters/schema** folder.

# Data Type Conversions

## Scalar Data Types

Scalar data types like Strings, Numbers, and Boolean values are converted to the target types automatically where necessary by the mapping converter when these values are encountered either as a message body or as part of a complex data structure. This applies to individual scalar values, as well as collections and arrays of these values.

## Date Types

Date values are like scalar values, but they come with a few more rules as they are dependent on the time zone that is used to perform the calculations. As discussed in “Using Patterns to Parse and Format Values” on page 31 , the system's default time zone ID is used by default while performing all conversions between various types of date and time types, and also to and from string values.

## Reading Source Object Properties

### Overview

Java Beans and Maps are probably the most common data structures that you encounter as part of data processing that needs to be done while integrating two or more systems. Each `sourceField` that is defined in the mapping file can point to a scalar value, a collection, an array, another bean, or a Map.

Mapping converters support four types of properties:

- Simple
- Mapped
- Indexed
- Nested

# Accessing Simple Properties

Simple properties are properties that you find at the top level for a Map or a Bean. Consider the following class structure:

```
class Order {
  String orderNumber;
  ZonedDateTime orderDate;
  List<Product> products;
  SalesChannel salesChannel;
  Map<String, Address> addresses;  // Keys 'BILLING', 'SHIPPING'
}

  class Product {
    long id;
    String sku;
    String name;
    String category;
}

  class SalesChannel {
    int id;
    String name;
    ...
}

class Address {
  String line1;
  String line2;
  ...
}
```

The properties named `orderNumber`, `orderDate`, `orderLines`, `salesChannel`, and `addresses` are all considered simple properties. In a mapping file, these fields can be referenced just by their simple name, the value given for the `name` attribute.

```
<field ...>
    <sourceField name="orderNumber" type="String"/>
</field>
<field ...>
    <sourceField name="orderDate" type="ZonedDateTime/>
</field>
```

Similarly, if the data is a Map, all the keys in the Map are considered simple properties. You can access them in the mapping file by the key value in the Map.

# Accessing Nested Properties

Nested properties are the properties of a bean which itself is the value of a property of its containing object.

In the class structure given above, all properties of the salesChannel property of the Order bean are considered nested properties. Nested properties can be accessed using <parentProperty><childProperty> notation in the name attribute.

```
<field ...>
    <sourceField name="salesChannel.name" type="String"/>
</field>
```

Nested properties can be from objects that are also values in a Map. For example, if Order had been a Map with its properties as keys, and its values the same as they are in the Order class above, the sales channel name is accessed in the same way as above using salesChannel.name.

# Accessing Mapped Properties

Consider the nested Map property addresses in the Order class above. Each key in the nested map would be considered a mapped property. Mapped properties can be accessed using <parentProperty>[mappedPropertyKey] notation.

```
<field ...>
    <sourceField name="addresses[BILLING]"
type="com.mycompany.model.Address"/>
</field>
```

You should use characters that are acceptable in Java variable names as properties in a Map when the Map instances are nested inside other beans or Maps.

# Accessing Indexed Properties

In the class structure above, the property named products is considered a simple property with a List as value, where as each element of the list is considered an indexed property. Indexed properties can be accessed using <parentProperty>[index] notation (same as a mapped property). The index value is zero-based (that is, the first element of the list has index 0, the second element has index 1, and so on).

In addition to elements of a List, elements of an array can also be accessed in the same manner.

```
<field ...>
    <sourceField name="products[1]"
type="com.mycompany.model.Product"/>
</field>
```

# Writing Target Object Properties

## Overview

The same four types of properties that were explained in the previous section are supported when creating target objects and writing property values in those objects.

## Writing Simple Properties

Writing a value to a simple property is as easy as reading it. You specify the `name` attribute on the `field` element.

```
<field name="orderNumber" type="String">
    <sourceField ... />
</field>
```

In order to write to any other type of property, you need to supply a bit more information to the converter through the mapping file.

Converters create output objects at run time. Therefore, types that are used on a `field` element or as a `targetType` on the `fieldMapper` element cannot be abstract types (except for a Map, List, and Set, which are taken to mean a HashMap, ArrayList, and HashSet, respectively). When dealing with the nested properties, mapped properties, or indexed properties on the target object, you need to first specify the type of the intermediate object and initialize it.

## Writing Nested Properties

For example, when you specify the nested property as `salesChannel.name`, the converter might not have a way to know that `salesChannel` is an object of type `SalesChannel`. The `SalesChannel` object needs to be created by the converter before it can set its name property.

Initialization of the intermediate objects can be done in the mapping file using the `initializer` element.

```
<initializer name="salesChannel"
type="com.mycompany.model.SalesChannel"/>
<field name="salesChannel.name" type="String">
    <sourceField ... />
</field>
```

The initialization of the `salesChannel` property must happen before the field mapping that is trying to write to a nested property, as shown above.

## Writing Mapped Properties

Similar to the nested properties, the Map that contains the mapped properties must be initialized before a value can be set to a mapped property.

```
<initializer name="addresses" type="Map<String,
com.mycompany.model.Address>"/>
<field name="addresses[BILLING]" type="com.mycompany.model.Address">
    <sourceField ... />
</field>
```

## Writing Indexed Properties

While writing to indexed properties, the container object, list, or array needs to be initialized.

```
<initializer name="products" type="List<Product>"/>
<field name="products[1]" type="com.mycompany.model.Product">
    <sourceField ... />
</field>
```

The following example shows how to initialize the `products` property if it is an array of `Product` objects.

```
<initializer name="products" type="Product[]" size="4"/>
<field name="products[1]" type="com.mycompany.model.Product">
    <sourceField ... />
</field>
```

While initializing arrays, you must also provide the size of the array. The array must be large enough to accommodate all the indexed properties that are being set in the mapping file to avoid a `NotWritablePropertyException` at run time.

# Converting Nested Beans or Maps

Accessing individual properties and elements of a nested object or collection might be useful in situations where the target type structure is different from the source type structure. In other situations, you might need to convert the complete nested bean or Map into another type.

For example, the Product details are coming in as a Map in the source type and it needs to be converted into a Product instance and assigned to a property in the target object.

In these situations, the system searches the converter registry to find a unique converter that can convert a Map to a Product instance. It would do so without any additional information.

```
<field name="product" type="com.mycompany.model.Product">
    <sourceField name="productAsMap" type="Map"/>
</field>
```

If the system cannot identify a unique converter, then the ID of the mapper that can do the conversion needs to be specified using the `mapperId` attribute of the `field` element.

```
<field name="product" type="com.mycompany.model.Product"
mapperId="productMapper">
    <sourceField name="productAsMap" type="Map"/>
</field>
```

# Converting a Collection and Arrays

## Overview

If the source object has a collection of objects as a property and it is necessary to convert all the instances of the collection to a different type and assign to another collection property in the target object, the operation would be similar to how individual bean or Map conversion is done as shown in the previous section.

For example, extending the previous example of converting a Map that contains properties of a product to a property of the target object as Product instance, here you are looking at a List of Maps that hold information about products in the source object. You need to convert that to a List of Product instances and assign to the `products` property on the target object.

Just as in the case of a single Map instance, the system searches the converter registry to find a unique converter that can convert a Map to a Product instance. It does so without any additional information. It applies this conversion repetitively to each element of the source collection and adds the converted objects to the collection on the target object.

```
<field name="products" type="List<com.mycompany.model.Product>">
    <sourceField name="productsMaps" type="List<Map>"/>
</field>
```

When the system cannot identify a unique converter, the ID of the mapper that can do the conversion needs to be specified using the `mapperId` attribute on the `field` element.

```
<field name="products" type="List<com.mycompany.model.Product>"
mapperId="productMapper">
    <sourceField name="productsMaps" type="List<Map>"/>
</field>
```

When converting a collection of objects as shown above, it is not required that the collection type used on the source object and target object be the same. For example, a List can be converted to an array or a Set, or to any other Collection implementation class like `java.util.LinkedHashSet`, `java.util.TreeSet`, and so on as long as it is not an abstract type.

## Collections and Arrays with Scalar Values

The following example shows that when the collections contain scalar values, the converter automatically handles the conversion to a collection of target element type.

```
<field name="inningsScores" type="List<Integer>">
    <sourceField name="scores" type="List<String>"/>
</field>
```

# Circular Object References

Sometimes data structures that are used in applications might include circular references. You should avoid using circular references in data transmission formats. Nevertheless, they might appear in some instances. SAS Business Orchestration Services is capable of handling circular references during data type conversions.

The following example illustrates a circular reference. This example uses the following data types:

```
class Order {
  String id
  String customerId
  List<OrderLine> lines
}

class OrderLine {
  String id
  Order order
  String productId
  int quantity
}
```

This example uses the following instances:

```
Order order101 = new Order(id: 'order-101', customerId: 'customer-101')

OrderLine line10001 = new OrderLine(id: 'line-10001', order: order101,
productId: 'item-01', quantity: 2)
OrderLine line10002 = new OrderLine(id: 'line-10002', order: order101,
productId: 'item-02', quantity: 5)

order101.lines = [line10001, line10002]
```

`OrderLine` instances are part of a `List` property that are inside of an `Order` instance. An `Order` instance is a property of each of the `OrderLine` instances. This circular reference is troublesome because it starts the conversion operation at an `Order` instance, goes to the `OrderLine` instances, and each of those `OrderLine` instances leads back to the `Order` instance at the start.

If you attempt to use this data structure, the program goes into an infinite loop resulting in a `StackOverflowError` or `OutOfMemoryError` unless the program is written to tackle this scenario and short-circuit the reference chain.

SAS Business Orchestration Services keeps track of each bean and Map instance that is converted at run time, and if the same instance appears again, it uses the reference to the result of the earlier conversion instead of attempting to convert it again.

In the example above, when the type conversion starts with the `Order` instance `order101` (that is, a Map), it remembers the Map that is the output of that step. When it encounters `order101` again as part of converting `OrderLine` instances `line10001` and `line10002`, it uses the result (a Map) that it is keeping track of already. This avoids the program going into infinite recursion.

# Using the constant Element

All `field` elements in a mapping file must have a nested element that acts as the source of the data for that field. In some situations, the source object might not have any input field that could be used as the source for a required field in the target object. Under these circumstances, by using the `constant` element, you could send in a constant value as the source for that field.

```
<field name="division_name">
    <constant>FSI</constant>
</field>
```

The `constant` element takes its value as character data (that is, it is nested within the `constant` element tags and unless a `type` attribute value is specified). It is always considered a String value.

The `constant` element provides functionality that is very similar to the default value functionality on a `field` element.

- If a type is provided for the `constant` element, the system attempts to convert the String value to that data type.
- If a pattern is provided for the `constant` element, the system attempts to use the pattern while converting to the type provided. When a pattern is provided, a type must also be provided.
- Property placeholders, bean references, and so on work the same way in both cases.

There are three differences between the default attribute of `field` element and the `constant` element.

- The value of the `constant` element is used all the time. This enables you to set a value in the output when there is no corresponding source field.
- The `constant` element can take multi-line text as the value. The default attribute can accommodate only a single line of text.
- Being an element, you can use CDATA tags to enclose strings with special characters that normally need escaping in XML data (for example, characters like an ampersand, <, >, and so on).

For more information, see “Configuring Default Values for Fields” on page 34.

# Using XPath Expressions for Random Access in XML Strings

## Overview

When structured data needed to be transferred between systems, initial methods like comma-separated values (CSV), fixed-width records, proprietary binary formats, and so on, were used. In order to make the data easily readable by humans, XML became a popular option and was used heavily with Web Services and SOA architectures involving SOAP (Simple Object Access Protocol). Next came JavaScript Object Notation (JSON) that became popular with REST services and Microservice architectures. You can find all of these data transmission formats in active use when you try to integrate systems built at different times. SAS Business Orchestration Services provides multiple mechanisms to deal with these message or file contents. If you need to marshal or unmarshal these messages or files in their entirety, you can use the appropriate data format that is included in SAS Business Orchestration Services. For more information, see Chapter 6, “Working with Data Formats,” on page 61.

Sometimes you might not need all the data that is included in an XML or JSON document, but need only a few specific elements of it. You can use XPath expressions or JSON Pointer expressions to extract just the information that you need.

This section explains how to use XPath expressions to extract information from XML documents. The section after explains the use of JSON Pointers to achieve similar effect when dealing with JSON documents.

> **TIP** This feature enables you to read-only values from an XML document that is presented to the data type converter as a String. XML content cannot be generated using this feature.

## XPath

### Overview

XPath is an expression language that is used to query specific parts of an XML file. When an XPath expression gets evaluated against a given XML file, the result of the

evaluation could be one or more atomic data values (like text, numeric, or Boolean), or it could be one or more complex XML document constructs (elements, document fragments, processing instructions, and so on). In the context of mapping one data type into another in SAS Business Orchestration Services, the focus is on extracting and transforming atomic data values.

# Using an xmlQuery Element in a field Element

The following sample XML file demonstrates the use of various XPath expressions:

```
<Order id="order-101" totalAmount="1,201.00" orderDate="12242020">
   <Customer id="cust-101">
      <Name>Jane Doe</Name>
   </Customer>
   <Items>
      <Item id="item-101">Milk - 2% Organic</Item>
      <Item id="item-102">Oreo Cookies - Family Pack</Item>
      <Item id="item-103">Breyers Ice Cream - Vanilla - Gallon</Item>
   </Items>
</Order>
```

The following sample shows how to use XPath expressions to read various parts of the XML file:

```
<!-- Reads id attribute of the Customer element nested inside the
Order element -->
<field name="customerId">
   <xmlQuery expression="/Order/Customer/@id"/>
</field>

<!-- Reads the text content of the Name element nested inside Order
and then in Customer elements -->
<field name="customerName">
   <xmlQuery expression="/Order/Customer/Name/text()"/>
</field>

<!-- Reads totalAmount attribute of the Order element and parses it as
a Number using the given pattern -->
<field name="orderTotal">
   <xmlQuery type="Number" pattern="###,##0.00" expression="/Order/
@totalAmount"/></field>

<!-- Reads the text content of all Item elements that are nested
inside an Order element -->
<field name="itemDescriptions">
   <xmlQuery type="List<String>" expression="/Order/Items/Item/
text()"/>
</field>

<!-- Reads the contents of id attribute of all Item elements that are
nested inside an Order element -->
<field name="itemIds">
   <xmlQuery type="List<String>" expression="/Order/Items/Item/@id"/>
```

```
</field>
```

XPath queries are not limited to reading values based on a strict nested structure. It allows for much more complex queries to precisely extract the data that you are looking for. A full discussion of XPath syntax and capabilities is beyond the scope of this document. The following is a sample of a conditional query expression:

```
<!-- Read the id attribute of an Item element nested inside Order
element, where the text content of the Item element contains the
string 'Oreo' -->
<field name="oreoItemId">
   <xmlQuery expression="/Order/Items/Item[contains(text(), 'Oreo')]/
@id"/>
</field>
```

# Dealing with XML Namespaces

None of the examples in the previous section use namespaces in the XML content. The XPath expressions used in those examples are appropriate only when no namespaces are associated with the elements in the XML content.

When namespaces are used in the XML content, the XPath Standard requires you to use the prefix that is associated with the namespace to which the element or attribute is defined under. XPath considers the element tags with no prefix to have no namespace associated with them. Even when a namespace is defined as the default namespace in the XML content, XPath expressions still need a prefix associated with the namespace URI in order to be able to refer to that element tags properly. The following XML content is identical to what was shown in the previous section, except that there is a default namespace associated at the root of the document, the `Order` element. Since this namespace declaration is placed at the root element of the document, it applies to all the elements and attributes in the XML content.

```
<Order xmlns="http://www.sas.com/order" id="order-101"
totalAmount="1,201.00" orderDate="12242020">
   <Customer id="cust-101">
       <Name>Jane Doe</Name>
   </Customer>
   <Items>
      <Item id="item-101">Milk - 2% Organic</Item>
      <Item id="item-102">Oreo Cookies - Family Pack</Item>
      <Item id="item-103">Breyers Ice Cream - Vanilla - Gallon</Item>
   </Items>
</Order>
```

Because of this namespace declaration, if you use the XPath expression `<xmlQuery expression="/Order/Customer/@id"/>`, a NULL value is returned instead of the customer id 'cust-101' in the XML file. This is because the `Order` element in the expression does not contain a prefix. The XPath processor considers this as an instruction to look for an `Order` element that is not associated to any namespace.

In order to access elements and attributes that are associated with a namespace in the XML content through XPath expressions, you need to associate a prefix with the namespace using the following syntax:

```
<order:Order xmlns:order="http://www.sas.com/order" ...>
 ....
```

```
</order:Order>
```

**Note:** All other element tags in the XML file would also contain an `order:` prefix. For brevity, the complete XML was not repeated

With the prefix defined for the namespace in the XML content, you can refer to these elements by using that prefix:

```
<xmlQuery expression="/order:Order/order:Customer/@id"/>
```

# Dealing with the Default Namespaces

Although the XML standard enables you to associate a namespace to any element as the default (that is, without specifying the prefix), XPath cannot process the default namespaces without some additional configuration. The XPath processor needs all namespaces of interest to be associated with a prefix, and then use those prefixes in your XPath expressions. If you defined prefixes for all namespaces at the root element of the XML document, those prefixes could be used in the XPath expressions without any additional configuration. This is shown in the example above with order as the prefix for the namespace that is used. When a default namespace is used, there will not be a prefix associated with that namespace declaration. (It is then considered the default namespace.) To refer to any element in this default namespace, you need to associate the default namespace with a prefix. The following sections explain how to define or modify prefixes for namespaces.

# Defining Global Prefixes for Namespaces

By default, the XPath processor in SAS Business Orchestration Services is aware of all prefixes that were defined for namespaces at the root element of the XML document that is being processed. If a default namespace is used and it does not have a prefix associated with it, then you need to configure it in the mapping file. Prefixes need to be defined in the mapping file if namespaces are associated to an element other than the root element, including any default namespaces that are defined for child elements.

```
<fieldMapper ...>
  <xmlQueryNamespace prefix="ord" uri="http://www.sas.com/order"/>
  <field name="customerId">
    <xmlQuery expression="/ord:Order/ord:Customer/@id"/>
  </field>
</fieldMapper>
```

The mapping file sample above shows associating the prefix `ord` with the namespace **`URI http://www.sas.com/order`**. If the namespace URI is the default namespace, then the given prefix is associated with that namespace. If the namespace has a prefix already defined in the XML content, the new prefix would be added in addition to the one in the XML content.

> **TIP** If the prefix defined in the `xmlQueryNamespace` element was already defined in the XML content, it is replaced with the new association. Make sure that prefixes that you define in the mapping file are not duplicates of the prefixes that are defined in the XML content, especially if you are associating the prefix with a namespace that is not the same as the association in the XML content. The namespace prefixes defined in the mapping file take precedence over the prefixes provided in the XML content.

Global prefixes that are defined using the `xmlQueryNamespace` elements nested inside the `fieldMapper` element are effective for all `xmlQuery` elements that appear in that mapping file. If there is a need to override these associations for a specific `xmlQuery` element that is nested inside a `sourceField` element, then a local namespace can be declared before the `xmlQuery` element that is nested inside the `sourceField` element. The local namespace declarations are applicable only to the `xmlQuery` element that is nested in the same `sourceField` element.

# Defining Local Prefixes for Namespaces

You can override namespace prefixes for a single instance of an `xmlQuery` element that is nested inside a `sourceField` element by nesting `xmlQueryNamespace` elements within the same `sourceField` element. These local definitions take precedence over the global prefixes that are defined as direct children of the `fieldMapper` element or the default namespaces that you define in the XML content itself.

> **TIP** There is no way to define a local namespace prefix for `xmlQuery` elements that are nested directly under a `field` element.

```
<fieldMapper ...>
  <field name="customerId">
    <sourceField name="orderData">
      <xmlQueryNamespace prefix="ord" uri="http://www.sas.com/order"/>
      <xmlQuery expression="/ord:Order/ord:Customer/@id"/>
    </sourceField>
  </field>
</fieldMapper>
```

There is no limit on the number of prefixes that you can define in the mapping file.

# Handling Nested XML Content

If the XML content that you are trying to process is contained within another data structure as a property value, then it is possible to invoke the data converter that handles the XML content from a data converter that deals with the enclosing object. You can use the `mapperId` attribute of the `field` element, similar to handling any other nested object. You might encounter this sort of nested XML or JSON content when encountering a system that wants to use a relational database table as the

data store, but want to have flexibility around the structure of the data in certain aspects.

In the following example, the report structure, formatting, and data contained within in it might be too generic to be distilled into a single set of tables and columns. It might have been designed to store the data needed to create a printable report in XML with some key columns to search and filter the reports.

```
Report {
  String id;
  String accountId;
  String countryCode;
  String activityType;
  String reportData;       // This property holds the data needed for
the system to generate a printable report in XML format
}
```

If the Report data type needs to be converted into another structure, along with certain data elements of the reportDate property, it can be using two data converters.

```
<!-- 'reportDataConverter' handles the extraction of necessary
elements of data from XML into a Map,
which gets invoked by the converter that handles the Report object -->

<fieldMapper id="reportConverter" ...>
  <field name="account">
    <sourceField name="accountId"/>
  <field>
    ...
  <field name="reportDetails" type="Map"
mapperId="reportDataConverter">
    <sourceField name="reportData"/>
  </field>
</fieldMapper>
```

# A Shortcut to Handle Nested XML Content

The nested invocation of another mapper shown above would be appropriate if you need to extract multiple data elements from the XML content. In situations where only a few data elements are needed from the nested XML content, you can nest an `xmlQuery` element inside a `sourceField` element.

```
<!-- The XML content from the reportData field from the Report object
gets passed on to the
xmlQuery element to extract the date value. The resulting value is set
as the value of reportDate
in the output generated by the converter -->

<field name="reportDate">
  <sourceField name="reportData">
    <xmlQuery type="Date" pattern="MMddyyyy" expression="/ReportView/
Header/ReportDate/text()"/>
  </sourceField>
</field>
```

It is possible to use the same technique to extract multiple pieces of data from the XML content, but it might affect performance when a large number of fields are processed in this manner. In such cases, you should use a separate converter to handle data extraction from the XML content.

The local namespace prefixes can be defined even when the `xmlQuery` element is nested inside a `sourceField` element.

# Using JSON Pointers for Random Access in JSON Strings

The functionality that is available in the `jsonQuery` element is very similar to the functionality that is available in the `xmlQuery` element, and deals with JSON formatted text content.

Just like XPath expressions that are used in the `xmlQuery` element, the expression attribute of the `jsonQuery` element needs to follow JSON Pointer syntax. Although JSON Pointers can be used to extract information out of JSON formatted text, JSON Pointers offer limited functionality as opposed to XPath for XML. JSON Pointers cannot be used to combine data elements from multiple instances of a property. They always identify one data element of a JSON document. JSON Pointers also lack the conditional expressions that XPath provides. Full comparison and discussion of JSON Pointers is beyond the scope of this document.

Here are a few samples to illustrate the use of the `jsonQuery` element and JSON Pointer syntax. Since `jsonQuery` usage is similar to `xmlQuery` usage, follow the instructions in the section above about the `xmlQuery` element for other use cases. Make sure to replace `xmlQuery` with `jsonQuery` and use JSON Pointer syntax for your expressions.

> **TIP** This feature enables you to read-only values from a JSON document that is presented to the data type converter as a String. JSON content cannot be generated using this feature.

The following sample converter configurations assume the JSON structure:

```
{
  "id": "order-101",
  "totalAmount": "1,201.00",
  "orderDate": "12242020",
  "customer": {
    "id": "cust-101",
    "name": "Jane Doe"
  },
  "items": [
    {
      "id": "item-101",
      "description": "Milk - 2% Organic"
    },
    {
```

```
      "id": "item-102",
      "description": "Oreo Cookies - Family Pack"
    },
    {
      "id": "item-103",
      "description": "Breyers Ice Cream - Vanialla - Gallon"
    }
  ],
  "trackingNumbers": [
    1028929,
    324324,
    3423434
  ]
}
```

```
<!-- Reading a nested property -->
<field name="customerName">
  <jsonQuery expression="/customer/name"/>
</field>

<!-- Reading a property and parsing it -->
<field name="orderTotal">
  <jsonQuery type="Number" pattern="###,##0.00" expression="/
totalAmount"/>
</field>

<!-- Reading properties of a specific object in a nested array -->
<field name="secondItemId">
  <jsonQuery expression="/items/1/id"/>
</field>

<!-- Reading an array of values -->
<field name="trackingNumbers">
  <jsonQuery type="List<String>" expression="/trackingNumbers"/>
</field>
```

# Using the custom Element

## Overview

The previous sections have discussed how various built-in features of field mapping data converters work. This section explains how to use the `custom` element, along with its nested `param` and `groovy` elements, to extend the capabilities of the converters with any custom processing that might be necessary in situations where there is no built-in feature of the data converters that is capable of doing what you need.

This capability would be useful in situations like the following:

- Need to calculate the value for a target field dynamically. For example, to inject today's date or current time, split the string value in a source field into multiple fields in the target, or combine multiple fields in the source object into a single value in the target object, and so on.
- Need a custom calculation for a target field based on field values in the source object.
- Need a custom calculation for a target field based on other target field values that have been already processed.
- Need to access a system property as part of calculating the value for a target field.
- Need to use an external bean that contains the logic to calculate the value for a target field.
- Need to access an external resource to fetch additional data. (See the tip at the end of this section regarding this usage.)

# Simplest custom Example

The following example is one of the simplest usages of the `custom` element:

```
<field name="orderDate">
  <custom name="today">
    <groovy>java.time.LocalDate.now()</groovy>
  </custom>
</field>
```

When the converter executes the mapping file that contains this field declaration, the `orderDate` field in the target object that was created by the converter is set to current (local) date. Since no types are provided anywhere in the field declaration, the object that is returned by the Groovy code (which is an instance of `java.time.LocalDate`) is passed on as the value from the `custom` element, which is acting as the source for the field `orderDate`. The field also does not have a type declared. So, the local date instance gets set as the value for the `orderDate` field on the target object.

The structure of the actual Groovy code that gets executed at run time provides a clearer picture of how the Groovy code is linked to various elements at run time.

```
def groovy_today(org.apache.camel.Exchange exchange) {
    java.time.LocalDate.now()
}
```

The processing that happens at run time is equivalent to calling this method had you defined it in a custom Groovy class.

Even though there was no mention of the Camel exchange in the XML, it is available for the code inside the method to access by default.

**TIP** Theoretically, a simpler example could have been to return a static value like True or False or a static string value from the Groovy code block, but the `custom` element would have been overkill for that behavior. Instead, using the `constant` element is more appropriate for those situations, which is simpler to use than the `custom` element. If a static value needs to be injected when the source value is null or empty, you can use the `defaultValue` attribute on the `field` element for that purpose.

# A More Complex custom Example

The following example provides a more complex usage of the `custom` element:

```
<field name="estimatedShipDate" type="Date">
  <custom name="calculateShipDate">
    <param name="shipMethod" type="my.company.order.ShipMethod"
valueSource="shipMethod"/>
    <param name="orderLines" type="List<my.company.order.OrderLine>"
valueSource="orderLines"/>
    <param name="inventoryService"
type="my.company.order.InventoryService" valueSourceType="BEAN"
valueSource="#invSvcBean"/>
    <param name="economyLeadTime" type="int"
valueSourceType="PROPERTY"
valueSource="{{order.processing.economyLeadTime}}"/>
    <groovy>
      <![CDATA[
        import java.time.LocalDate
        import my.company.order.OrderLine

        import static my.company.order.ShipMethod.*

        int leadTime = 0
        for (OrderLine line : orderLines) {
            int itemLeadTime = inventoryService.getLeadTime(line.sku)
            leadTime = Math.max(leadTime, itemLeadTime)
        }

        LocalDate shipDate = LocalDate.now()
        shipDate = shipDate.plusDays(leadTime)

        if (shipMethod == ECONOMY) {
            shipDate = shipDate.plusDays(economyLeadTime)
        } else if (shipMethod == PRIORITY) {
            shipDate = shipDate.plusDays(1)
        }
        return shipDate
      ]]>
    </groovy>
  </custom>
</field>
```

The equivalent Groovy code executed at run time would be the following:

```
import java.time.LocalDate
import my.company.order.OrderLine
import static my.company.order.ShipMethod.*

def groovy_calculateShipDate(org.apache.camel.Exchange exchange,
java.util.Map<java.lang.String, java.lang.Object> cache,
                             my.company.order.ShipMethod shipMethod,
java.util.List<my.company.order.OrderLine> orderLines,
                             my.company.order.InventoryService
inventoryService, java.lang.Integer economyLeadTime) {
    int leadTime = 0
    for (OrderLine line : orderLines) {
        int itemLeadTime = inventoryService.getLeadTime(line.sku)
        leadTime = Math.max(leadTime, itemLeadTime)
    }

    LocalDate shipDate = LocalDate.now()
    shipDate = shipDate.plusDays(leadTime)

    if (shipMethod == ECONOMY) {
        shipDate = shipDate.plusDays(economyLeadTime)
    } else if (shipMethod == PRIORITY) {
        shipDate = shipDate.plusDays(1)
    }
    return shipDate
}
```

A few points to note in the above example:

- The `param` elements that are nested inside the `custom` element are exposed as method parameters in the Groovy code.
- `economyLeadTime` is read from a property value.
- The `inventoryService` bean is accessed from the bean registry.
- The fully qualified names are used for method parameter declarations. If no additional references are made in the code, then there is no need to add import statements for these classes.
- The import statements are added for classes referenced inside the code.
- The static import is used for `enum` and `ShipMethod`.
- The `exchange` parameter is always included, but you can ignore it if you do not need to access it.
- Although the `exchange` and `cache` parameters are shown in the method signature to illustrate their existence, they are not used in the example code.

When this converter executes, the return value from the `custom` element is a `LocalDate` instance, even through the field type is declared as `Date`. Since these do not match, the converter attempts to convert the `LocalDate` result into a `Date` instance before assigning the value to `estimatedShipDate` field in the target.

# Using a Bean Reference in a custom Element

The ability to access any bean from the registry is a powerful feature. It can be used to access any functionality that is available through those beans. This includes the capability to call external services and resources. For example, you can call an arbitrary REST API endpoint, or fetch data from a data store, and so on. But, beware of the risks of this usage.

Type converters are used in numerous places throughout the message processing routes. So, it is advisable to keep the logic inside the converters to a minimum, so that they can execute quickly. Because of the performance implications of calling other beans from a converter, you should avoid time-consuming calls to external services and resources.

To perform data enrichment, especially when accessing an external resource, the Enrich EIP is more appropriate in a lot of situations. Using Enrich EIP keeps the processing logic more modular, compared to embedding it as part of the code in the `custom` element, and it enables you to precisely tune when a time-consuming access to the external resource is executed.

When time-consuming calls are made from the `custom` element, it is because the calls are made every time that the converter gets executed. If the resource access is part of code written for more than one field in a converter, the effect on performance is multiplied accordingly.

Depending on the structure of the data types that you are dealing with in your application, sometimes a call to one data type converter can invoke other data type converters to convert nested data types inside the top-level object. This sort of setup can lead to an explosion of the number of time-consuming calls, which leads to drastic performance implications.

To illustrate the effects, consider the example in the previous section. That example uses a bean reference to `InventoryService`, which is a hypothetical service that has a `getLeadTime()` method that returns the lead time for any item given the item's stock-keeping unit (SKU). Ideally, this service method would look up the lead time in some in-memory structure, which gets refreshed periodically using some other process. Structuring the logic in this way would not be appropriate if this service executes a query every time you request the lead time for an item. Every converter invocation would end up executing `n` number of queries against the database in such scenario for `n` number of items in the collection. Instead, a better approach is to create a different service method in the service bean to fetch the lead time for all the items in the `orderLines` collection with a single call to the database to retrieve the maximum lead time out of all the items in the collection.

Typically, whenever a collection of objects needs to be processed, it is better to process the collection as a whole when making calls to external resources.

# Using the cache Element in a custom Element

In all Groovy scripts that are nested inside `custom` elements in a mapping, there is a shared cache available through the parameter named `cache`. The data type for the `cache` parameter is a `Map` with `Strings` as keys and `Objects` as values. This is a

temporary storage area that your code can use to share data between different Groovy script elements within the same mapping file. The Groovy script elements that are defined in the mapping file get executed in the order that they are declared. So a value stored in the cache by Groovy code defined earlier in the mapping file can be accessed by the Groovy code that is defined later.

Each invocation of the converter that contains `custom` elements gets a new instance of `Map`. You cannot share data between different invocations of the converter, or between invocations of different converters. The primary purpose of the cache is to avoid expensive operations that are performed more than once within the same converter. Cache can also be used to keep track of summary data that is relevant within the same execution of the converter.

The code in the following example fetches additional data elements from an external service and adds them to the target object that holds customer information (given only the customerId as part of the source object):

```
<field name="customerName">
  <custom name="fetchCustomerName">
    <param name="customerId" valueSource="customerId"/>
    <param name="customerDataService"
type="my.company.CustomerDataService" valueSourceType="BEAN"
valueSource="#custDataSvc"/>
    <groovy>
      <![CDATA[
        // This is the first block where external data is needed. So,
instead of fetching just the data we need for this field,
        // we could fetch all the data needed for fields further down
in the mapping file and store it in cache so that
        // other fields could fetch the data they need from cache
avoiding multiple calls to the service.

        cache.put('custDetails',
customerDataService.getCustomerData(customerId))
        return cache.get('custDetails').customerName
      ]]>
    </groovy>
  </custom>
</field>
<field name="customerSince">
  <custom name="fetchCustomerSince">
    <groovy>
      <![CDATA[
        // Since the prior block already fetched the data from the
external service and stored it in the cache, we can fetch the relevant
data from cache
        return cache.get('custDetails').customerSince
      ]]>
    </groovy>
  </custom>
</field>
```

In the above example, even though multiple fields in the target object need data from the external service, multiple calls were not made to the external service by fetching all the data that was needed from that service in a single call.

In the first code block, the external service was called and stored the result in the cache. The cache object that was provided to the code would be of type `java.util.Map<String, Object>` (that is, the key must be a string and the value

can be any object). The subsequent code blocks in the same mapper can access the cache and read the necessary data elements.

## Accessing Database Tables

Apache Camel provides `enrich` and `pollEnrich` features, along with customizable aggregators, as a mechanism to bring and merge data from sources like databases and external REST services. This is the preferred approach to perform data enrichment.

The ability to define parameters with bean references as values, and the cache to store data, provides another way to fetch data from databases and incorporate that into the output created by the data type converter.

When JDBC datasource is configured using `spring.datasource.* properties`, a Spring JdbcTemplate bean also gets automatically configured by Spring Boot. This bean can be accessed using the bean reference `#jdbcTemplate`. With access to the JdbcTemplate instance, SQL code can be executed inside the `<groovy>` section to store the return data in the cache for access in other Groovy script sections of the mapping file.

> **TIP** The SQL scripts that are embedded inside the mapping files get executed once per invocation of the data converter. Consider the performance implications of this mechanism. Using the enrich feature would provide more control on the number of calls that are made to external data sources.

# Limitations of the sas-convert-body-to Endpoint

Multidimensional arrays are not supported.

# Hot Updates to Mapping Files

You can modify or delete a mapping file that has already been deployed in the application. These changes are reflected and loaded into the application online without needing a restart of the application. If you create a new file in the application mapping files location, the new data type converters are loaded into the application. This is the default behavior. To disable live updates to mapping files, the following property must be added in the application properties file:

```
sas.integration.converters.live-update.enabled=false
```

The default processing time for the changes to mapping files to update and load the modified or new data type converters is 60 seconds. You can configure this time in the following property:

```
sas.integration.converter.mappers-refresh-time=<Time in milli-seconds>
```

**CAUTION**

Although it is possible to update, create, and delete mapping files on a live system, you should not do so in production without first testing in a development or QA environment before uploading your changes to production.

# 6

# Working with Data Formats



## Overview

SAS Business Orchestration Services supports generic formats like JSON, XML, delimited, fixed width, and so on. It also supports specialized formats like ISO 8583, SWIFT, and so on.

The following sections discuss how to use each of these formats.

## ISO 8583 Data Format

### Overview

SAS Business Orchestration Services provides support for the ISO 8583 format. SAS Business Orchestration Services supports three versions of ISO 8583 out of the box: 1987, 1993, and 2003 in ASCII encoding. For more information, see ISO 8583.

The version is dynamically determined by the message type indicator (MTI) in the message. A bitmap can be either represented as hexadecimal characters in ASCII encoding or plain binary format. For more information about how to use bitmap in binary format, see “Customizing the ISO 8583 Data Format” on page 62.

The message would be marshaled from and unmarshaled to a Java Map. The field numbers (string) in an ISO 8583 message are used as the keys in the map. When marshaling, the map object with a field number (string) as the key is expected from the input.

If you use a streaming interface like sockets, then the incoming stream must be broken down into individual ISO 8583 messages before the data format APIs call (marshal or unmarshal). For example, if messages are sent to the socket using netty, then the decoder must break the stream down to the individual message.

The metadata to configure three ISO 8583 format versions (1987, 1993 and 2003) in ASCII encoding are included in the `boss-api.jar` file.

# Customizing the ISO 8583 Data Format

By default, a bitmap is represented as hexadecimal characters and encoded with ASCII. To handle a bitmap in plain binary format, some configuration customization is needed.

There are two ways to customize the ISO 8583 data format to set bitmap in binary format:

1. Change the configuration of the default sas-iso8583 data format. Add the following entries in the properties that point to the binary bitmap codecs for the three versions of ISO 8583 on the classpath:

```
sas.integration.dataformat.iso8583.config-file-
location.0=classpath:formats/iso8583/bitmap-binary_codec8583_87.xml
sas.integration.dataformat.iso8583.config-file-
location.1=classpath:formats/iso8583/bitmap-binary_codec8583_93.xml
sas.integration.dataformat.iso8583.config-file-
location.2=classpath:formats/iso8583/bitmap-binary_codec8583_03.xml
```

2. Create a new ISO 8583 data format. The following example creates a new data format with bean ID named custom-iso8583. In this case, the default sas-iso8583 data format still exists with bitmap in ASCII encoding. You can use both the sas-iso8583 and custom-iso8583 data formats independently in the routes.

```
<bean id="custom-iso8583"
class="com.sas.integration.dataformat.iso8583.Iso8583DataFormat">
   <property name="configFileLocation">
      <map>
         <entry key="0" value="classpath:formats/iso8583/bitmap-
binary_codec8583_87.xml"/>
         <entry key="1" value="classpath:formats/iso8583/bitmap-
binary_codec8583_93.xml"/>
         <entry key="2" value="classpath:formats/iso8583/bitmap-
binary_codec8583_03.xml"/>
      </map>
   </property>
</bean>
```

## Using the ISO 8583 Data Format

The following code snippet shows how the ISO 8583 data format is used in routes.

```
<!--Unmarshal ISO8583 message to a Map object-->
<unmarshal><custom ref="sas-iso8583"/></unmarshal>

<!--Marshal a Map Object to ISO8583 message-->
<marshal><custom ref="sas-iso8583"/></marshal>

<!--Unmarshal using a customized ISO8583 data format. ie. custom
binary bitmap-->
<unmarshal><custom ref="custom-iso8583"/></unmarshal>
```

The following message is a sample message in ISO 8583 format with ASCII encoding.

```
02003220000000808000000010000000001500120604120000000112340001840
```

The message can be broken down into three parts:

1. Message type indicator: 0200
2. Bitmap: 3220000000808000
3. Data fields: 000010000000001500120604120000000112340001840

   The data fields that are translated to JSON format are as follows: `{"11":"000001","3":"000010","4":"000000001500","49":"840","7":"1206041200","41":"12340001"}`

# JSON Data Format

## Jackson Library

Jackson libraries are included in SAS Business Orchestration Services to enable marshaling and unmarshaling of JSON formatted data. For more information about how to configure and use Camel Jackson for the JSON data format, see Camel JSON and JSON Jackson.

The Jackson ObjectMapper provides the functionality for reading and writing JSON either to and from basic plain old Java objects (POJO), or to and from a general-purpose JSON tree model. For more information about how to configure and use the Jackson ObjectMapper in Spring Boot, see Customize the Jackson ObjectMapper.

If an ObjectMapper configuration is not supplied in the application configuration, then Spring Boot provides an instance automatically. This instance, whether custom configured or provided by Spring Boot, is considered the default. You should

customize this instance to apply marshaling and unmarshaling settings consistently across the application.

In places where additional customization is needed to the ObjectMapper, additional properties can be provided as needed. These additional properties are applied on top of the settings in the default instance. This additional customization is applied only in that context while keeping the default instance unchanged.

The example below uses the following SAS Business Orchestration Services application-level properties:

```
spring.jackson.deserialization.USE_BIG_DECIMAL_FOR_FLOATS = true
camel.dataformat.json-jackson.enable-features =
USE_BIG_INTEGER_FOR_INTS
```

In route A, you have the following:

```
<marshal><json library="Jackson" prettyPrint="true"/></marshal>
```

In route B, you have the following:

```
<marshal><json library="Jackson" prettyPrint="false"/></marshal>
```

As a result, route A and route B each have their own ObjectMapper instance in JacksonDataFormat. Both routes have USE_BIG_DECIMAL_FOR_FLOATS (inherited from Spring Boot) and the USE_BIG_INTEGER_FOR_INTS features enabled. However, the `ObjectMapper` instance in route A has prettyPrint enabled while route B does not.

# 7

# Store and Forward



## Overview

Store and forward is a common transport process in which messages that are sent to services are stored in an intermediary medium in case these services are not available due to a connectivity or an unavailability issue. These messages that are stored in the intermediary medium are then processed and resent or forwarded to their destination services when this service becomes available.

There are many options available for the intermediary medium such as databases, Java message service products like ActiveMQ, streaming frameworks like Kafka, cache systems like Redis, or other messaging platforms like RabbitMQ. The store and forward feature in SAS Business Orchestration Services is not activated for a service unless it is configured in the route that is accessing this service.

This chapter explains what you need to do to configure the intermediate medium to store messages that were not sent to a service due to service unavailability. In addition to configuring the storage medium, you need to configure the route to

access a service and to trigger store and forward in case of an exception due to service unavailability.

You might need to use store and forward when you need to guarantee the delivery of a message to a service. When sending a request to this service, the route must be configured to store the message in case of an error or exception when accessing this service. You then resend this message using store and forward when the service becomes available.

# Configuration Using Error Handler

## Overview

The store and forward feature is not enabled by default in SAS Business Orchestration Services. To enable store and forward, you need to set only the `sas.integration.saf.enabled` property to **`true`**:

```
sas.integration.saf.enabled=true
```

To set the health check frequency for the destination service, the `sas.integration.saf.health-check-frequency` property should be configured as shown below. This is optional. If it is not set, then it defaults to 5000 milliseconds. The value can be any value that is in a valid Java Time Duration format.

```
sas.integration.saf.health-check-frequency=<integer value in
milliseconds>
```

## Limitations

If you are using the in-memory store and forward configuration, then the application's memory is used for store and forward. If the SAS Business Orchestration Services application restarts, then messages that are stored are lost.

If you are using the in-memory store and forward configuration, you should not disable forwarding.

## Route Configuration

Store and forward is triggered when the store and forward feature is enabled and in the case of an error or an exception in a route, if this route is configured to use the store and forward feature. This section describes how to configure a Camel route to use the SAS Business Orchestration Services store and forward feature.

To enable store and forward, complete these steps:

1 Configure the error handlers for a route by setting the `sas.integration.saf.handlers.<error handler name>` properties in the `application.properties` file:

```
sas.integration.saf.handlers.<handler name>.store-type=<memory or jms>
sas.integration.saf.handlers.<handler name>.store-id=<jms connection
factory bean>
sas.integration.saf.handlers.<handler name>.on-exception=<comma
separated fully qualified exceptions to be handled>
sas.integration.saf.handlers.<handler name>.disable-forward=<true or
false>
sas.integration.saf.handlers.<handler name>.health-check-uri=<service
health endpoint>
sas.integration.saf.handlers.<handler name>.forward-uri=<route
endpoint URI to forward>
sas.integration.saf.handlers.<handler name>.memory-queue-size=<Maximum
number that can be stored in a memory store. Default is 1000.>
```

The `health-check-uri` property is optional. If this property is not configured, then the application uses the `forward-uri` property to check the service health.

The `health-check-uri` property specifies a Camel route that checks the service status. If this health check route does not return an exception, then store and forward assumes that the service is up and forwards the message on to the route that is specified in the `forward-uri` property. If the health check route returns an exception, then store and forward assumes that the service is still down and continues to check the health of the service periodically as defined in the `health-check-frequency` property.

The `store-id` property is used when `store-type=jms`. It is optional if you auto-configure one JMS connection factory in Spring boot. It is mandatory if you configure multiple connection factories to connect to multiple JMS providers.

The `forward-uri` property is required. The value in this property is where the failed message is forwarded to in the store and forward process.

The following is the only required property:

```
sas.integration.saf.handlers.<handler name>.forward-uri=<route
endpoint URI to forward>
```

The `sas.integration.saf.handlers.<handler name>.on-exception` property is optional. If it is not set, then it defaults to the list of exceptions below:

```
java.net.ConnectionException
java.net.SocketTimeoutException
java.net.NoRouteToHostException
java.net.PortUnreachableException
java.net.SocketException
```

The `sas.integration.saf.handlers.<handler name>.disable-forward` property is optional. If it is not set, then the default value is False.

Here is a configuration example with these properties:

```
sas.integration.saf.handlers.handler1.store-id=mq1
sas.integration.saf.handlers.handler1.store-type=jms
sas.integration.saf.handlers.handler1.on-
exception=java.net.ConnectException
sas.integration.saf.handlers.handler1.disable-forward=false
sas.integration.saf.handlers.handler1.queue=orderService
```

```
sas.integration.saf.handlers.handler1.message-
converter=com.company.MyMessageConverter
sas.integration.saf.handlers.handler1.forward-uri=direct:myForwardPoint
```

The `message-converter` property is optional. The default value is `org.springframework.jms.support.converter.SimpleMessageConverter`.

Depending on the store type for the handler, extra variables might need to be configured.

For the JMS store type, the `sas.integration.saf.handlers.handler1.queue=myqueue` property must be configured:

For the JMS store type, if you have stored more than one JMS connection factory, or if you need to use a specific JMS connection factory, then you must define the `sas.integration.saf.handlers.handler1.store-id=mq1` property.

For the memory store type (the default), you must specify the `sas.integration.saf.handlers.<handler name>.queue` property.

2 Create an XML route configuration file.

Reference the dead letter error handler that was created in Step 1 by using the `errorHandlerRef` attribute as shown in the following XML:

```
<route id="myRouteId" errorHandlerRef="handler1">
   <to uri="http://localhost:8080?bridgeEndpoint=true"/>
</route>
..
<route id="myForwardRoute">
   <from uri="direct:myForwardPoint">
</route>
```

The above XML configuration triggers the store and forward feature if any exception is thrown while trying to access the service endpoint, if the error handler was configured to handle such an exception. The route with the route ID `myForwardRoute` refers to the `sas.integration.saf.handlers.handler1.forward-uri` property in the handler configuration. If you configure the store and forward routes using the `errorHandlerRef` attributes, then the route terminates if an exception occurs. To have more control when handling an exception that triggers store and forward, you can use the following SAS `saf` component in an `onException` clause:

```
<route id="saf">
  <from uri="direct:safPost"/>
  <convertBodyTo type="java.lang.String"/>
  <to uri="http://localhost:8080?bridgeEndpoint=true"/>
  <onException>
   <exception>java.net.ConnectException</exception>
   <to uri="sas-saf:/?handlerId=handler1"/>
  </onException>
</route>
..
<route id="myForwardRoute">
 <from uri="direct:myForwardPoint>
 ..
</route>
```

**Note:** You must configure one handler per service endpoint. If you use the same error handler for multiple routes and these routes have different endpoints, then the store and forward feature does not work properly and it is indeterministic.

## Message Forwarding Order

Messages are pulled from the store and are forwarded to the destination service using the first-in first-out order (FIFO). It tries FIFO as a best effort and there is no guarantee that the messages are processed in that order.

# Supported Data Stores Using Error Handler

## Overview

SAS Business Orchestration Services uses one of the following data stores to store a message as part of the store and forward process:

- cache services
  - heap memory
- Java Message Service
  - ActiveMQ

## Memory

If the store and forward feature is enabled and no store is configured in the application, then the application uses the memory as a store for store and forward. This is not recommended and a warning is logged when the application starts. You do not need to configure this memory store.

## Java Messaging Service

SAS Business Orchestration Services supports ActiveMQ as a data store for store and forward.

# Store and Forward JMS Queues

You must create a persistent queue for the store and forward JMS type store. Creating and configuring queues is different across vendors. For more information about on how to create and configure a persistent queue, see the JMS vendor documentation.

# ActiveMQ

To auto-configure an ActiveMQ data store, complete these steps:

1. Create or edit the `${BOSS_CONFIG}/application.properties` file.
2. Copy the following properties to the `${BOSS_CONFIG}/application.properties` file and configure them.

   ```
   spring.activemq.broker-url: ActiveMQ broker url
   spring.activemq.user: ActiveMQ username
   spring.activemq.password: ActiveMQ password
   ```

   For more information about the auto-configuration properties of ActiveMQ, see Spring Boot Common Application Properties.
3. Restart SAS Business Orchestration Services.

Here is an example of auto-configuring the properties for ActiveMQ:

```
spring.activemq.broker-url=tcp://0.0.0.0:61613
spring.activemq.user=admin
spring.activemq.password=password
```

For more information about the auto-configuration properties that are available for ActiveMQ, see Spring Boot Integration Properties.

If you want to configure other ActiveMQ connections, the following properties must be defined:

```
sas.integration.activemq.connections.<id>.broker-url: ActiveMQ broker
url.
sas.integration.activemq.connections.<id>.user: username.
sas.integration.activemq.connections.<id>.password: password.
```

In the properties, ID is a unique identifier of this Active MQ connection configuration.

Here is an example that shows the above properties configured:

```
sas.integration.activemq.connections.mq1.broker-url=tcp://0.0.0.0:61613
sas.integration.activemq.connections.mq1.user=admin
sas.integration.activemq.connections.mq1.password=password
```

Once the properties are configured, SAS Business Orchestration Services auto-configures ActiveMQ and uses it to configure store and forward if the feature is enabled.

If you choose to configure multiple ActiveMQ connections, then you must disable auto-configuration for ActiveMQ by setting the following property:

```
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jms
.activemq.ActiveMQAutoConfiguration
```

# Other JMS Implementations

Other JMS-compliant implementations are supported. If you need to use a JMS provider other than ActiveMQ, then you must define a connection factory as a bean in an XML Spring mapping file. This is because different JMS implementations do not have the standard configuration parameters. XML is the recommended way to define a bean. You can reference the bean for the error handler by setting the `sas.integration.saf.handlers.handler2.storeId` property to the ID of the connection factory bean.

**Note:** This feature has not yet been fully tested and verified for running in production. It is highly recommended that you use ActiveMQ as the JMS store and forward store.

# JMS Auto-configuration

JMS auto-configuration can be used for one vendor at a time. For example, you cannot enable auto-configuration for both Active MQ and IBM MQ at the same time in the same application. If you want to auto-configure ActiveMQ or if ActiveMQ is auto-configured, then IBM MQ auto-configuration must be disabled. To disable IBM MQ, set the following property:

```
spring.autoconfigure.exclude=com.ibm.mq.spring.boot.MQAutoConfiguration
```

If IBM MQ is auto-configured or if you need to auto-configure IBM MQ, set the following property to disable auto-configuration for ActiveMQ:

```
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jms
.activemq.ActiveMQAutoConfiguration
```

To disable auto-configuration for both ActiveMQ and IBM MQ, set the following property:

```
spring.autoconfigure.exclude=com.ibm.mq.spring.boot.MQAutoConfiguration
,org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfig
uration
```

# Message Converter

A JMS store uses the default JMS message converter named `SimpleMessageConverter`. This converter supports string, bytes, map, and serializable objects. If the message contains an object that is not serializable, then you must create a custom message converter that implements `org.springframework.jms.support.converter.MessageConverter`. The store

and forward handler can use this message converter by setting the `message-converter` property. This property is explained in “Route Configuration” on page 66.

# Legacy Configuration

Store and forward can be configured using the legacy configuration as in previous releases. For more information see, https://documentation.sas.com/?docsetId=eopug&docsetVersion=10.1&docsetTarget=p1nosvekakhl5rn12ro53ns20yde.htm in *SAS Business Orchestration Services 10.1: User's Guide*.

Follow this document to configure routes, beans, and rest endpoints.

# Supported Data Stores

- ActiveMQ
- IBM MQ
- In-Memory data store
- Kafka
- Oracle
- Postgres
- RabbitMQ
- Redis

# 8

# Using a Stand-in Response



## Overview

A 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.

## Enable Stand-In Response

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

```
<route id="customTransactionHandler">
     <from uri="disruptor:customTransaction?concurrentConsumers=8"/>
     <doTry id="Main route(custom)">
        <convertBodyTo type="java.lang.String"/>
```

```
        <bean id="decodeWebString" ref="odeUtils"
method="decodeWebString"/>
        <bean id="cust_to_Transaction" ref="inboundMapper"
method="customToTransaction"/>
        <bean id="StoreUnqKeyForCustomTxn" ref="uniqueKeySaver"/>
        <!—Steps omitted -->
        <doCatch id="Catch Custom Timeout(custom)">

<exception>com.sas.finance.fraud.ol.exceptions.EnrichTimeoutException</
exception>
           <exception>org.apache.camel.ExchangeTimedOutException</
exception>
              <to id="Stand-in-Response(custom)"
uri="direct:standInResponse"/>
        </doCatch>
     </doTry>
  </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.

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

The default implementation for the `handleTimeout` method gets the stand-in response from the stand-in data store based on the account number in the request. The request continues processing 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 the Stand-In Data Store

## 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 following bean definition creates an in-memory data store for stand-in:

```
<bean id="requestTimeoutHandler"
   class="com.sas.finance.fraud.ol.standin.RequestTimeoutHandler">
      <property name="standInDatastore">
```

```
        <bean
class="com.sas.finance.fraud.ol.standin.CachedResponses"/>
      </property>
  </bean>
```

The in-memory data store does not offer persistence, and it is not shared 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 following Redis client JAR files and their transitive dependencies must be installed in the **$BOSS_LIB** directory:

- commons-pool2-2.6.2.jar
- jedis-2.9.0.jar

The following is a sample configuration for the Redis stand-in data store. It is assumed that Springboot auto-configures the `redisConnectionFactory` and `stringRedisTemplate` beans, which are used and injected into other beans as defined below:

```
<bean id="requestTimeoutHandler"
class="com.sas.finance.fraud.ol.standin.RequestTimeoutHandler">
     <property name="standInDatastore">
        <bean class="com.sas.finance.fraud.ol.standin.RedisDatastore">
           <constructor-arg name="template"
ref="transactionRedisTemplate"/>
           <constructor-arg name="stringTemplate"
ref="stringRedisTemplate"/>
           <property name="timestampProvider" ref="timestampProvider"/>
        </bean>
  </bean>

  <bean id="transactionRedisTemplate"
class="org.springframework.data.redis.core.RedisTemplate">
     <property name="connectionFactory" ref="redisConnectionFactory"/>
     <property name="keySerializer">
        <bean
class="org.springframework.data.redis.serializer.StringRedisSerializer"
/>
     </property>
     <property name="valueSerializer">
        <bean
class="com.sas.finance.fraud.ol.standin.TransactionRedisSerializer"/>
     </property>
  </bean>
```

```
  <!-- By default, rqo_tran_date and rqo_tran_time are being used. It
is configurable though. -->
  <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. -->

  <bean id="timestampProvider"
class="com.sas.finance.fraud.ol.standin.ConfigurableDateTimeProvider"/>
     <constructor-arg name="dateFieldName" value="rqo_tran_date"/>
     <constructor-arg name="timeFieldName" value="rqo_tran_time"/>
     <constructor-arg name="dateTimeFormat" value="yyyyMMddHH:mm:ss.SS
"/>
  </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 in-memory data store and the Redis data store. Performance varies depending on your environment and system settings. You need to put an additional JDBC JAR file for the database that you choose in the **$BOSS_LIB** directory.

The configuration for using a relational database is similar to using a Redis server. It assumes that Springboot autoconfigures the `jdbcTemplate` bean and that there is a data source that connects to the database.

```
<bean id="requestTimeoutHandler"
class="com.sas.finance.fraud.ol.standin.RequestTimeoutHandler">
     <!-- Use a relational database as StandIn Dat48  a Store -->
     <property name="standInDatastore">
        <bean
class="com.sas.finance.fraud.ol.standin.RelationalDbDatastore">
           <constructor-arg name="jdbcTemplate" ref="jdbcTemplate"/>
           <property name="timestampProvider" ref="timestampProvider"/>
        </bean>
     </property>
  </bean>
  <bean id="timestampProvider"
class="com.sas.finance.fraud.ol.standin.ConfigurableDateTimeProvider"/>
     <constructor-arg name="dateFieldName" value="rqo_tran_date"/>
     <constructor-arg name="timeFieldName" value="rqo_tran_time"/>
     <constructor-arg name="dateTimeFormat" value="yyyyMMddHH:mm:ss.SS
"/>
  </bean>
```

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));
```

# 9

# Offline Files Processing



## Overview

Offline files processing is an enhanced feature that uses the Camel File component as part of its implementation. The Camel File component processes the entire file, whereas SAS Business Orchestration Services processes a single line in the file. Each line represents one data entry. Offline files processing auto-resumes from the last failure point after a crash. It is semi-transactional, meaning that one data entry only can be marked as processed after a response has been received. The position of all the failed data entries in the file and content (optional) are saved in a separate file for inspection.

# File Flow

You must place the offline files in the **landing** directory. Files that are complete and ready for processing must be accompanied by marker files with `.done` appended after the name.

Within each SAS Business Orchestration Services instance, there is one Camel route that uses the file component to pick up files one-by-one from the **landing** directory and then writes them to the **inProgress_${*nodeId*}** directory. The **inProgress_${*nodeId*}** directory is unique to each SAS Business Orchestration Services instance. The SAS Business Orchestration Services ID can be `FQDN`, an IP address, or a human-assigned unique ID.

After a file has been processed successfully, SAS Business Orchestration Services moves it to the **backup** directory.

SAS Business Orchestration Services uses a single thread to process a file line-by-line. The order of processing follows the order of the data entries in the files.

# Error Handling

## Crash

If SAS Business Orchestration Services crashes while processing a file, you should restart SAS Business Orchestration Services on the same server to resume processing from the last failure data entry.

## File Errors

If an exception occurs during file processing (for example, if there is an IOException or the file contains the wrong encoding), then Camel ends the processing of the entire file and puts the file in the **error_files** directory along with a file that contains the processed line number so that you can review the file.

## Data Entry or Record Errors

If an error occurs for a record during file processing (for example, if the CSV format of a line cannot be parsed), then SAS Business Orchestration Services creates a file

named `file_name.record_error` in the **`error_record`** directory and appends the position of the data entry and the content (optional), so that it can be reviewed.

## Manual Correction

In rare cases (timing-dependent), you might need to perform a manual correction to handle the errors. For example, if `boss01` crashed and is not going to recover, then you can go to the **`inprogress_boss01`** directory and copy the **`${file.name}`** and **`${file.name}.processed`** files to another directory (for example, **`inprogress_boss02`**) and create a **`${file.name}.done`** file. That way, `boss02` can resume processing that file from the failure point.

Another even more rare case for manual correction occurs when SAS Business Orchestration Services crashes while trying to pre-move a file into the **`inprogress_${nodeId}/temp`** directory and while writing to the **`inProgress_${nodeId}`** directory. You can delete the partially written file in the **`inProgress_${nodeId}`** directory and move the file in the **`inprogress_${nodeId}/temp`** to the **`landing`** directory. Then you can create the **`.done`** file so that SAS Business Orchestration Services can process the file again. If the corresponding **`${file.name}.camelLock`** file exists in the **`landing`** directory, you should remove it.

# Configuration

## Beans

Here is an example beans definition that is required by the offline files processing:

```
<bean id="producerRecordOffline"
class="org.apache.camel.impl.engine.DefaultProducerTemplate" init-
method="start">
    <constructor-arg name="camelContext" ref="camelContext"/>
    <property name="defaultEndpointUri" value="direct:recordOffline"/>
</bean>
<bean id="offlineFilesId" class="java.lang.String">
    <constructor-arg type="java.lang.String" value="boss01"/>
</bean>
<bean id="persistFilerFactory"
class="com.sas.finance.fraud.ol.offlinefiles.SerialLineFilePersistFacto
ry">
    <constructor-arg name="template" ref="producerRecordOffline"/>
    <!--can be set to ignore first n lines-->
    <property name="failoverStartingLineNumber" value="1"/>
    <property name="logicOrRegexExcludeList"><list><value>^#.*$</
value></list></property>
    <property name="persistLineErrorContent" value="true"/>
</bean>
```

```
<bean id="lineRecordFileProcessor"
class="com.sas.finance.fraud.ol.offlinefiles.LineRecordFileProcessor">
    <constructor-arg name="fileProcessorFactory"
ref="persistFilerFactory"/>
</bean>
<bean id="offlinefilesMove"
class="com.sas.finance.fraud.ol.commons.SimpleRouter" init-
method="init">
    <constructor-arg name="context" ref="camelContext"/>
    <constructor-arg name="fromStr" value="file://$
{offlinefiles.landing}?scheduler=spring&scheduler.cron=
${offlinefiles.move.schedule}&delete=true&preMove=../inprogress_$
{offlinefiles.id}/temp/$simple{file:name}
&readLock=markerFile&moveFailed=../../error_files&doneFileName=
$simple{file:name}.done
&readLockDeleteOrphanLockFiles=false"/>
    <constructor-arg name="toStr" value="file://$
{offlinefiles.rootdir}/inprogress_${offlinefiles.id}?doneFileName=
$simple{file:name}.done"/>
    <property name="errorHandlerFactory" ref="noErrorHandler"/>
    <property name="startOrder" value="88"/>
    <property name="routeId" value="offlinefilesMove"/>
</bean>
<bean id="idempotentRepository"
class="org.apache.camel.support.processor.idempotent.MemoryIdempotentRe
pository"/>
<bean id="offlinefilesProcess"
class="com.sas.finance.fraud.ol.commons.SimpleRouter" init-
method="init">
    <constructor-arg name="context" ref="camelContext"/>
    <constructor-arg name="fromStr" value="file://$
{offlinefiles.rootdir}/inprogress_${offlinefiles.id}
?initialDelay=${offlinefiles.process.initialdelay}&delay=$
{offlinefiles.process.period}&move=
../backup&moveFailed=../error_files&doneFileName=
$simple{file:name}.done&idempotentRepository=
#idempotentRepository&readLock=idempotent"/>
    <constructor-arg name="toStr" value="direct:offlinefileTo"/>
    <property name="errorHandlerFactory" ref="noErrorHandler"/>
    <property name="routeId" value="offlinefilesProcess"/>
</bean>
```

# Camel Routes

Here are some example Camel routes:

```
<route id="offlinefileTo">
    <from uri="direct:offlinefileTo"/>
    <ean ref="lineRecordFileProcessor"/>
</route>
<route id="recordOffline">
    <from uri="direct:recordOffline"/>
    .......steps to process data entry
</route>
```

# Directory Structure

Here is the directory structure for offline file processing:

```
/workingDir           # read/write permission is required this directory and down
    /backup           # backup successfully processed files
    /error_files      # exception occurs during file processing (that is, IOException).
"processed" file (if any) indicates how many lines processed before exception. The
directory is for human inspection only.
          example_error_file.csv
          example_error_file.csv.processed
    /error_records    # exception occurs during processing single data entry. They are
appended to a file for human inspection.
          example_file_with_error_records.csv.error
    /inprogress_boss01.sas.com   # file in progress by BOSS 01. "processed" file indicates
how many lines processed at the moment. In case of SAS Business Orchestration Services
crash, it will start from the next line.
         /temp                  # file preMoved here
          example_file_in_progress.csv
          example_file_in_progress.csv.camelLock
          example_file_in_progress.csv.done
          example_file_in_progress.csv.processed
    /inprogress_boss02.sas.com
    /landing          # landing directory for files. Separate "done" file is a marker
indicate the file is complete and ready for processing.
          example_file_to_be_processed_next.csv
          example_file_to_be_processed_next.csv.done
```

# 10

# Datastore Backed by Apache Geode (Pivotal Gemfire)



## Data Store Overview

SAS Business Orchestration Services can store and retrieve data from an Apache Geode or a Pivotal GemFire cluster as a data store or as a cache.

## Configuration

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*. Then, a Geode creates a data store bean that takes *String* as a key and a value.

```
<gfe:client-cache id="boss-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"/>

<bean id="geode"
class="com.sas.finance.fraud.ol.datastore.gemfire.GeodeSimpleDataCache"
>
   <constructor-arg name="region" ref="region01"/>
</bean>
```

For more information about how to configure Geode with Spring XML, see Spring Data for Apache Geode Reference Guide.

# Usage in the Routes

In the following example, the Camel exchange's header contains the key as a string. The body is the value, which is also a string. The following code stores the key value pair in the Geode data store:

```
<bean ref="geode" method="store(${header.key}, ${body})"/>
```

The following code retrieves the value as the body in the Camel exchange:

```
<bean ref="geode" method="retrieve(${header.key})"/>
```

# Working with Other Data Stores and Cache Implementations

Camel out-of-the-box provides support for many data stores and cache implementations in the form of Camel Components (for example, Ehcache, Hazelcast, Redis, and so on). For a complete list, see the Camel Components web page.

# 11

# Integrating with SAS OnDemand Decision Engine



## Overview

SAS OnDemand Decision Engine is a component that sends a transaction to SAS OnDemand Decision Engine and receives a response transaction from SAS OnDemand Decision Engine using a socket. For more information about OnDemand Decision Engine, see *SAS OnDemand Decision Engine: System Administrator's Guide*.

# Convert a Message to a SAS OnDemand Decision Engine Transaction

## Overview

To convert a message to a SAS OnDemand Decision Engine transaction, complete these steps:

1. If you are deploying custom rules, then import the generated addendum file to `${BOSS_CONFIG}`. For more information, see "Import an Addendum File".
2. Add any required beans to the `${BOSS_CONFIG}/beans` directory as an XML file. For more information, see "Required Beans for SAS OnDemand Decision Engine Component Integration".
3. Update the `${BOSS_CONFIG}/application.properties` file with connection information. For more information, see "Update the ${BOSS_CONFIG}/application.properties File".
4. Create a SAS OnDemand Decision Engine transaction. For more information, see "Creating a SAS Detection Transaction".
5. Send the transaction to the SAS OnDemand Decision Engine. For more information, see "Send a Transaction to SAS OnDemand Decision Engine".
6. Convert the returned transaction back into a Map (optional). For more information, see "Convert a Returned Transaction Back to a Map".

## Import an Addendum File

Whenever a new addendum file is generated by a SAS OnDemand Decision Engine, copy it to the `${BOSS_CONFIG}` directory and restart SAS Business Orchestration Services to pick it up. A new addendum file is generated whenever the rules authoring component deploys rules and there are changes to the variables configured within the web application. The addendum file should have a name like `06.01.00-addendum.xml`. The addendum file maps variables in SAS Fraud Management to those used in SAS Business Orchestration Services.

SAS Business Orchestration Services provides native support for multiple organizations in SAS OnDemand Decision Engine transactions.

In the following example, the addendum file has two tenants, identified by the organizations named ACME and EMCA. It is defined with the following segments. For more information about addendum files and tenants in SAS OnDemand

Decision Engine, see *SAS OnDemand Decision Engine: System Administrator's Guide*.

**Note:** Some fields and content were omitted for brevity.

```
<tenant multiOrg="ACME">
  <segment name="I00" id="14000000" buildId="50158">
    ...
    ...
    <field name="i00_num_00000_008" offset="32" format="RB8.0"
buildId="50155">
      <alias>_i_num</alias>
      <initialValue>0</initialValue>
      <description>Variable in Ixx_Seg1</description>
    </field>
  </segment>
  <segment name="I01" id="14000100" buildId="50157">
    ...
    ...
    <field name="i01_str_00000_008" offset="32" format="$CHAR8."
buildId="50157">
      <alias>_q_counter</alias>
      <initialValue>ABACAP</initialValue>
    </field>
  </segment>
<segment name="V04" id="10070400" buildId="50152">
    ...
    ...
    <field name="v04_str_00000_016" offset="125" format="$CHAR16."
buildId="50152">
      <alias>_k_counterr</alias>
      <initialValue>a</initialValue>
    </field>
  </segment>
</tenant>
<tenant multiOrg="EMCA">
  <segment name="I00" id="14000000" buildId="50158">
    ...
    ...
    <field name="i00_num_00000_008" offset="32" format="RB8.0"
buildId="50156">
      <alias>_i_num</alias>
      <initialValue>0</initialValue>
    </field>
  </segment>
  <segment name="I02" id="14000200" buildId="50158">
    ...
    ...
    <field name="i02_str_00000_008" offset="32" format="$CHAR8."
buildId="50158">
      <alias>_q_counter</alias>
      <initialValue>ABBFT</initialValue>
    </field>
  </segment>
<segment name="V03" id="10070300" buildId="50147">
    ...
```

```
    ...
    <field name="v03_str_00000_016" offset="125" format="$CHAR16."
buildId="50147">
      <alias>_k_counterr</alias>
      <initialValue>a</initialValue>
    </field>
  </segment>
</tenant>
```

# Required Beans for SAS OnDemand Decision Engine Component Integration

The beans should be set up according to the addendum file to match the segment name with an alias prefix. The alias coincides with the variable keys seen in both the addendum file and the SAS OnDemand Decision Engine.

```
<bean
class="org.springframework.beans.factory.config.MethodInvokingFactoryBe
an">
    <property name="staticMethod"
value="com.sas.finance.fraud.ol.utils.OdeUtils.setMultiOrgCustSegNameMa
p"/>
    <property name="arguments">
        <list>
            <value>ACME</value>
            <map>
                <entry key="_i_" value="I00"/>
                <entry key="_q_" value="I01"/>
                <entry key="_k_" value="V04"/>
            </map>
        </list>
    </property>
</bean>
<bean
class="org.springframework.beans.factory.config.MethodInvokingFactoryBe
an">
    <property name="staticMethod"
value="com.sas.finance.fraud.ol.utils.OdeUtils.setMultiOrgCustSegNameMa
p"/>
    <property name="arguments">
        <list>
            <value>ECMA</value>
            <map>
                <entry key="_i_" value="I00"/>
                <entry key="_q_" value="I02"/>
                <entry key="_k_" value="V03"/>
            </map>
        </list>
    </property>
</bean>
```

Similarly, if the smh_multi_org_name field is used with a single tenant named GLOBAL, then a single bean should be set in the addendum file to match the segment name with an alias prefix.

Single Tenant "GLOBAL"

```
<bean
class="org.springframework.beans.factory.config.MethodInvokingFactoryBe
an">
    <property name="staticMethod"
value="com.sas.finance.fraud.ol.utils.OdeUtils.setMultiOrgCustSegNameMa
p"/>
    <property name="arguments">
        <list>
            <value>GLOBAL</value>
            <map>
                <entry key="_i_" value="I00"/>
                ......
                ......
            </map>
        </list>
    </property>
</bean>
```

# Update the ${BOSS_CONFIG}/ application.properties File

The SAS OnDemand Decision Engine works with Spring Boot auto-configuration. All Camel Netty Spring Boot auto-configuration options that are documented in Camel Compnents are supported with SAS Business Orchestration Services namespace. This means that the prefix `camel.component.netty` is replaced with `sas.integration.component.ode`. For example, if you want to auto-configure to use a different value for a connection time-out, you can set the following properties:

```
sas.integration.component.ode.netty-config.connect-timeout= 5000
```

Due to a bug that exists in the Camel Netty component auto-configuration, the encoders and decoders cannot be configured using properties. You can configure multiple SAS OnDemand Decision Engines using the following properties. The following example configures two SAS OnDemand Decision Engines:

- `sas.integration.component.ode.instances.<instanceId>.hostname`
- `sas.integration.component.ode.instances.<instanceId>.port`

```
# SAS OnDemand Decision Engine   1
sas.integration.component.ode.instances.ode1.hostname=ode1.mycompany.co
m
# port is optional;  default   to  5018
sas.integration.component.ode.instances.ode1.port= 5018

# SAS OnDemand Decision Engine   2
sas.integration.component.ode.instances.ode2.hostname=ode2.mycompany.co
m
# port is optional;  default   to  5018
sas.integration.component.ode.instances.ode2.port= 5019
```

# Creating a SAS Detection Transaction

## Create a SAS Detection Transaction

Use the bean in the `OdeUtils` class to convert a Map to a SAS OnDemand Decision Engine transaction. The input map's keys must have the same name as the transaction field's name or an alias that you defined in an addendum file. The data type converter can be used to generate a Map with the correct keys. The following code can be used in the route for conversion:

```
< bean   beanType = "com.sas.finance.fraud.ol.utils.OdeUtils"   method
= "toTransaction2" />
```

## Use an Older smh_msg_version

In SAS Business Orchestration Services 10.2, the default smh_msg_version is 06.01.00. If you need an older version before converting a Map to a transaction, then change the smh_msg_version field in the Map. The following example uses the 04.04.03 version in the route:

```
<!-- change 'smh_msg_version' in Map and then convert to Transaction --
>
<script><groovy>exchange.getIn().getBody().put('smh_msg_version',
'04.04.03')</script></groovy>
<bean beanType="com.sas.finance.fraud.ol.utils.OdeUtils"
method="toTransaction2"/>
```

# Send a Transaction to SAS OnDemand Decision Engine

A TCP socket is the preferred SAS OnDemand Decision Engine communication protocol. The SAS OnDemand Decision Engine component extends from the Camel-Netty component and fully supports all Camel-Netty options. For information about the Camel Netty component, see the Camel documentation about the Netty component.

The input to SAS OnDemand Decision Engine must be a transaction object. The output from SAS OnDemand Scoring Engine is also a transaction object in request-response mode.

Default encoders and decoders are configured out of the box, so there is no need for you to provide them in the route. Sync is enabled by default, which means that it is in request-response mode. They can be overwritten in the route step.

The route sends a SAS OnDemand Decision Engine transaction to a SAS OnDemand Decision Engine and waits for a response with all the default settings or with Spring Boot auto-configured:

```
# target is SAS OnDemand Decision Engine 1 as defined in the previous
properties
# input must be Transaction object; output is Transaction Object
< to   uri = "sas-ode://ode1" />
```

If you want to run it in fire-and-forget mode (do not forget to set SAS OnDemand Decision Engine to no response and change the addendum file accordingly), then configure the following:

```
# target is SAS OnDemand Decision Engine 2 as defined in the previous
properties
< to   uri = "sas-ode://ode2?sync=false" />
```

## Convert a Returned Transaction Back to a Map

After receiving a SAS OnDemand Decision Engine transaction, you can use the bean in the OdeUtils class to convert the transaction back to a Map (optional). This enables you to more easily parse and modify the returned values.

```
< bean   beanType =
"com.sas.finance.fraud.ol.xmlmapping.utils.TransactionBackedMap"
method = "createMapFromTransaction" />
```

# 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 a secure socket is enabled for SAS Fraud Management, you need to configure a 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 OnDemand Decision Engine: System Administrator's Guide*.

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

1. Obtain a certificate from SAS Fraud Management that SAS Business Orchestration Services is going to communicate with.
2. On the machine where SAS Business Orchestration Services is installed, import the SAS Fraud Management certificate into the truststore. The following shows how you can use keytool to import it:

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

3 Configure an `SslContextParameters` bean. For more information, see the Camel documentation about the Netty component.

4 In the `application.properties` file, set `sslEnabled` to **`true`**.

5 Make sure that your route configuration uses the correct URI and the `SslContextParameters` bean ID that was created in Step 3.

```
<to uri="sas-ode://ode1?
ssl={{sslEnabled}}&sslContextParameters=#sslContextParametersBeanId"/>
```

# Using Store and Forward

## JMS-Compliant Message Queue Data Store

To configure a store and forward handler and a JMS-compliant message queue connection such as ActiveMQ, see Chapter 7, “Store and Forward,” on page 65.

The following example assumes that you defined a store and forward handler named handler1. The only additional property that you need to define is the `MessageConverter` property that converts a SAS OnDemand Decision Engine transaction object to a JMS message, and vice versa.

```
sas.integration.saf.handlers.handler1.message-
converter=com.sas.integration.ode.TransactionMessageConverter
```

## Using a Store and Forward Handler in the Route

The following sample route stores SAS OnDemand Decision Engine transactions when SAS OnDemand Decision Engine is offline.

```
<route id="myOdeStoreRoute" errorHandlerRef="handler">
    <from uri="direct:myOdeStore"/>
    <to uri="sas-ode://ode1"/>
</route>
```

The following sample route can be used to forward a transaction to SAS OnDemand Decision Engine. The forward route URI must be configured in the store and forward configuration.

```
<route id="myForwardRoute">
    <from uri="direct:forwardOde"/>
    <to uri="sas-ode://ode1"/>
</route>
```

# 12

# 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

## 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.

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:

- The 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 instance, you can use a configuration such as the following:

```
<loadBalance id="ODE Load Balancer" inheritErrorHandler="false">
     <custom ref="odeLoadBalancer"/>
           <to id="ode_host_1" uri="netty4:tcp://{{ode_host_1}}:
{{ode_port_1}}?sync=true&encoder=#encoder&decoder=#decoder"/>
           <to id="ode_host_2" uri="netty4:tcp://{{ode_host_2}}:
{{ode_port_2}}?sync=true&encoder=#encoder&decoder=#decoder"/>
           <to id="ode_host_3" uri="netty4:tcp://{{ode_host_3}}:
{{ode_port_3}}?sync=true&encoder=#encoder&decoder=#decoder"/>
</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.

In addition to failover capability, SAS Business Orchestration Services customized load balancer also remembers which external services are offline so that it skips them if they are offline. SAS Business Orchestration Services also provides a service health checker that periodically checks the health of external services. It detects the recovery of services.

Here is a typical configuration for a custom load balancer for SAS OnDemand Scoring Engine:

```
<bean id="serviceUnitChainGenerator"
class="com.sas.finance.fraud.ol.monitor.LbServiceUnitChainGenerator">
     <constructor-arg name="generators">
         <list>
              <bean
class="com.sas.finance.fraud.ol.monitor.SendProcessorServiceUnitGenerat
or"/>
              <bean
class="com.sas.finance.fraud.ol.monitor.MockOdeServiceUnitGenerator"/>
```

```
        </list>
    </constructor-arg>
</bean>

<bean id="ODEStatus" class="com.sas.finance.fraud.ol.mq.ODEStatus"/>

<bean id="odeLoadbalancerFactory"
class="com.sas.finance.fraud.ol.monitor.OdesDynamicFailoverLoadbalancer
Factory">
    <constructor-arg name="serviceUnitGenerator"
ref="serviceUnitChainGenerator"/>
    <constructor-arg name="healthCheckServiceName" value="ODE"/>
    <constructor-arg name="camelContext" ref="camelContext"/>
    <constructor-arg name="serviceStatus" ref="ODEStatus"/>
    <constructor-arg name="periodeInSec" value="30"/>
    <constructor-arg name="id" value="odeLoadBalancerFromBean"/>
    <constructor-arg name="exceptions">
        <list>
            <value>java.net.ConnectException</value>
            <value>java.lang.RuntimeException</value>
        </list>
    </constructor-arg>
    <property name="startListeners">
        <list>
            <ref bean="odeCamelClientsManager"/>
        </list>
    </property>
</bean>
<bean id="odeLoadBalancer"
class="com.sas.finance.fraud.ol.monitor.OdesDynamicFailoverLoadbalancer
Factory"
         factory-bean="odeLoadbalancerFactory" factory-
method="build"/>
<bean id="odeCamelClientsManager"
class="com.sas.finance.fraud.ol.monitor.OdeCamelClientsManager">
    <constructor-arg name="serviceUnitGenerator"
ref="serviceUnitChainGenerator"/>
</bean>
```

# 13

# 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 offerings 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 Camel SSLContextParameters

Java `SSLContext` is the key component in configuring SSL/TLS through the JSSE API. Camel provides a simple utility called `SSLContextParameters` to configure `SSLContext`. Once configured, `SSLContextParameters` can be used with many other Camel components such as Jetty, Netty, Mail, FTP, and so on, to enable their secure communications. For a full list of supported components and detailed instructions, see Camel JSSE utility.

# Configuring Jetty HTTPs

SAS Business Orchestration Services uses Camel-Jetty as the default HTTP service component.

For information about how to configure Jetty, see the documentation about the Camel Jetty component.

```
camel.component.jetty.use-global-ssl-context-parameters=true
```

# Configuring Netty over SSL

For information about how to configure Netty over SSL, see the documentation about the Camel Netty component.

# Encoding Transactions Using Beans

The encoding and decoding of incoming and outgoing transactions should be done using beans. The default character sets that are supported are ASCII, UTF8, UTF-16, Big5, and Cp1047.

Use the following numbers in the first byte to specify the character set:

- ASCII: 0
- UTF8: 1

- UTF-16: 2
- Big5: 3
- Cp1047: 4

In addition, you can use other character sets by specifying them in an optional **`charset_definition.properties`** file.

All character sets that are supported by Java are also supported for encoding the byte stream. For a complete list, see Oracle Java documentation.

The following example contains test sockets:

```
<rest>
   <post uri="/testSocket">
      <route id="testSocket">
         <convertBodyTo type="java.lang.String"/>
         <to uri="netty:tcp://localhost:5020?
sync=true&encoders=#stringEncoder&decoders=#stringDecoder"/>
         <setBody>
            <groovy>
               <'Caller received: ' +
exchange.getIn().getBody(String.class)>
            </groovy>
         </setBody>
      </route>
   </post>
</rest>

<route id="sampleSocketService">
   <from uri="netty:tcp://localhost:5020?
sync=true&encoders=#stringEncoder&decoders=#stringDecoder"/>
   <to uri="log:sampleSocketService?level=INFO&showAll=true"/>
   <setBody>
      <groovy>
         <'Server received: ' + exchange.getIn().getBody(String.class)>
      </groovy>
   </setBody>
</route>
```

The following code specifies the character set to use to encode and decode transactions. In this example, it is set to 2, UTF-16.

```
<!-- encoder and decoder to be used when communicating via socket. -->
<bean id="stringEncoder"
class="com.sas.finance.fraud.ol.converter.EopStringEncoder">
   <property name="charsetId" value="2"/>
</bean>

<bean id="stringDecoder"
class="com.sas.finance.fraud.ol.converter.EopStringDecoderChannelHandle
rFactory"/>
```

Here is the corresponding **`charset_definition.properties`** file:

```
#sas.nonlocalizable.properties
#Add as many defaults as you wish. For a whole list of supported
names, check
#https://docs.oracle.com/javase/8/docs/technotes/guides/intl/
encoding.doc.html
```

```
#DEFAULT_CHARSETS = {"ASCII", "UTF8", "UTF-16", "Big5", "Cp1047"};
0=ASCII
1=UTF8
2=UTF-16
3=Big5
4=Cp1047
5=TIS620
6=Cp1255
7=MS932
8=Cp1250
```

# Encrypting Credentials

## Configure Credentials Using Jasypt

The Ulisesbocchio Jasypt library is not included in the SAS Business Orchestration Services application. If you need to configure credentials using Jasypt, then download the Ulisesbocchio Jasypt JAR file and install it in the `${BOSS_LIB}` folder.

To configure credentials using Jasypt, complete these steps:

1. Download the following JAR files to the `${BOSS_LIB}` folder.
   - https://repo1.maven.org/maven2/com/github/ulisesbocchio/jasypt-spring-boot/2.1.2/jasypt-spring-boot-2.1.2.jar
   - https://repo1.maven.org/maven2/com/github/ulisesbocchio/jasypt-spring-boot-starter/2.1.2/jasypt-spring-boot-starter-2.1.2.jar
2. Set the `JASYPT_ENCRYPTOR_PASSWORD` environment variable using the following command:

   ```
   export JASYPT_ENCRYPTOR_PASSWORD=<myPassPhrase>
   ```
3. Encrypt the plaintext passwords in the `application.properties` file using these steps:
   a. `cd` to **`${BOSS_BIN}`**.
   b. For each password that you want to encrypt, issue the following command:

      ```
      ./encrypt.sh <myPassword>
      ```

      **Note:** In the following steps, `<myPassPhrase>` is used to encrypt the password. It is later used to decrypt the encrypted values to get the actual password. You can choose any value for `<myPassPhrase>`.

      Here is what the output from this command looks like:

      ```
      Source text: <myPassword>
      Encrypted text: rP9MQdXYSNCCToS0rrV9iqb/Fq4lq1C7
      ```

c Replace the plain source text with the encrypted text from the step above in the `${BOSS_CONFIG}/application.properties` file:

```
passPhraseForKeyStoreRest=ENC(rP9MQdXYSNCCToS0rrV9iqb/Fq4lq1C7)
```

4 Restart SAS Business Orchestration Services.

5 (Recommended) Clear your command history and unset the exported environment variable to make sure that no one can see your JASYPT_ENCRYPTOR_PASSWORD.

**CAUTION**

If you choose to download and use the Jasypt-Spring-boot JAR file, then you might have to modify some of your existing beans that use the `MethodInvokingFactoryBean` bean to avoid a bug that was introduced by the Jasypt-Spring-boot JAR file. For example, in the `odeLoadBalancer` bean shown below, the Spring `MethodInvokingFactoryBean` is used to build the `odeLoadBalancer` bean using an upstream bean named `odeLoadbalancerFactory`. The `odeLoadbalancerFactory` bean has the `CamelContext` bean as one of its dependency beans. This condition triggers a bug that causes SAS Business Orchestration Services to not pick up routes and REST routes. Besides the `CamelContext` bean, the Camel `TypeConverter`, `ProducerTemplate`, and `ConsumerTemplate` beans also trigger this issue. Do not use the `MethodInvokingFactoryBean` bean if any Camel entities are used as an upstream dependency.

```
<bean id="odeLoadbalancerFactory"
class="com.sas.finance.fraud.ol.monitor.OdesDynamicFailoverLoadbalancerFactory">
    <constructor-arg name="camelContext" ref="camelContext"/>
        ......
        ......
</bean>
<bean id="odeLoadBalancer"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="odeLoadbalancerFactory" />
    <property name="targetMethod" value="build" />
</bean>
```

The following example shows a workaround for this issue that uses factory-bean and factory-method instead of the MethodInvokingFactoryBean bean. Factory-bean and factory-method require that the method being called has a return type. This is generally fine for a bean definition where the bean itself is returned.

```
<bean id="odeLoadbalancerFactory"
class="com.sas.finance.fraud.ol.monitor.OdesDynamicFailoverLoadbalancerFactory">
    <constructor-arg name="camelContext" ref="camelContext"/>
        ......
        ......
</bean>
<bean id="odeLoadBalancer" factory-bean="odeLoadbalancerFactory" factory-method="build"/>
```

# Appendix 1

# Renewing Your Software License



## Software License Checks

A valid license file is required to run SAS Business Orchestration Services. Typically, this license file is provided electronically by SAS during the order fulfillment process.

The product license file should be named `setinit.txt`. It should be placed in the root of configuration folder, `$BOSS_CONFIG`.

SAS Business Orchestration Services executes license checks at start-up and every 24 hours from that moment onward. Whenever SAS Business Orchestration Services detects that the license has expired, you will find warnings in the application's log file. If you do not provide an updated license file before the warning period expires, then SAS Business Orchestration Services shuts down and it cannot be restarted without a valid license file.

For more information, see Licensing: How To Apply Licenses in SAS® Viya® 3.5 Administration.

# Appendix 2

# XML Configuration File Templates



## Route Definitions

```
<?xml version="1.0" encoding="UTF-8"?>
<routes xmlns="http://camel.apache.org/schema/spring">
   <route id="routeId">
      <!-- Route definition -->
   </route>

   <!-- More routes -->
</routes>
```

## REST Endpoint Definitions

```
<?xml version="1.0" encoding="UTF-8"?>
<rests xmlns="http://camel.apache.org/schema/spring">
   <rest path="path">
      <!-- REST endpoint definitions -->
   </rest>

   <rest path="anotherPath">
      <!-- REST endpoint definitions -->
   </rest>
```

```
   <!-- ... -->
</rests>
```

# Bean Definitions

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="beanId" class="fully.qualified.bean.class.name">
      <!-- Bean definition parameters, if any -->
   </bean>

   <!-- More beans -->
</beans>
```

# Converter Mapping File

```
<?xml version="1.0" encoding="UTF-8"?>
<fieldMapper xmlns="https://www.sas.com/boss/fieldMapper-10.2"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://www.sas.com/boss/
fieldMapper-10.2 fieldMapper-10.2.xsd"
             sourceType="source.type"
             targetType="target.type"
             id="mapperId">
   <field name="...">
      <!-- source definition -->
   </field>

   <!-- More field definitions -->
</fieldMapper>
```

# Appendix 3

# SAS Business Orchestration Services 10.2 Migration Guide for Running in Legacy Mode



## Prerequisite

This document applies only to customers who are migrating from SAS Business Orchestration Services 10.1 to SAS Business Orchestration Services 10.2. If you are running SAS Business Orchestration Services 1.x, follow the steps in Migrating to SAS Business Orchestration Services 10.1 and upgrade to SAS Business Orchestration Services 10.1 first.

# Overview

There have been significant changes made in SAS Business Orchestration Services 10.2 with the goal of making the product easier to use and to introduce additional capabilities. Besides functional changes, SAS Business Orchestration Services has been upgraded to use the latest version of the Apache Camel run-time engine.

Every effort was made to maintain backward compatibility, but there were some changes that were introduced that require you to change your configuration. Some of these changes are the result of the Camel 3.x run-time upgrade and some changes were made with the goal to enhance product usability.

In situations where a feature was enhanced, the older feature has been deprecated. The sections where the older feature is deprecated are highlighted with warnings to provide you with information about how to use the enhanced feature. You should make the necessary changes to your configurations to remove the use of any deprecated features. Deprecated features will be removed from SAS Business Orchestration Services in a future release.

# Migration Steps

## Logging

SAS Business Orchestration Services 10.2 uses Logback with Spring Boot auto-configuration. In SAS Business Orchestration Services 10.1, Log4j2 was used.

For more information about how to configure the Logback with Springboot auto-configuration or a Logback XML file, see Chapter 4, “Logging,” on page 13.

Once the configuration is complete, the Log4j2 configuration XML file or files can then be backed up to a safe location or removed.

## Changes Driven by the Camel 3.x Upgrade

### Overview

For complete information about the changes that were made to Camel 3.x and the migration tasks that you need to complete when you migrate your Camel 2.x configuration to Camel 3.x run time, see the APACHE CAMEL 2.X TO 3.0

MIGRATION GUIDE. The following is a quick summary of the changes that might affect you.

> **IMPORTANT** You are strongly advised to go through the official Camel Migration Guide for a complete list of the changes and migrations needed.

# Renamed Components

The following components have been renamed:

- HTTP4 has been renamed to HTTP.
  - All endpoint URLs in the configuration files that use the format `http4://` or `https4://` need to be changed to `http://` and `https://`.
- NETTY4 has been renamed to NETTY.
  - All endpoint URLs in the configuration files that use the format `netty4://` need to be changed to `netty://`.
- JETTY9 has been renamed to JETTY.
  - All endpoint URLs in the configuration files that use the format `jetty9://` need to be changed to `jetty://`.

# Features That Are No Longer Supported

If your existing configuration files defined more than one Camel context, then the configuration files need to be refactored to bring all the routes and service endpoints into a single context.

# Components That Are No Longer Available

**Camel Components**

None of the Camel components that were included in previous releases of SAS Business Orchestration Services have been removed. However, if you downloaded additional Camel components and added them to your implementation, you need to replace these components with another component that offers similar functionality. For more information, see REMOVED COMPONENTS in the *Camel 2.x to 3.0 Migration Guide*.

**Note:** In Camel 2.x, there were two variations of some components, each with a different name. The Camel 3.x release discontinued the older variation of the component and reused the old variation name for the newer component. Components that fall under this change include `camel-http` and `camel-netty`. Camel 3.x has components with these names that are equivalent to `camel-`

`http4` and `camel-netty4`. The original Camel 2.x components `camel-http` and `camel-netty` are no longer available.

**SAS Components**

> **TIP** None of the custom components built by SAS were removed in SAS Business Orchestration Services 10.2.

# Changes to Endpoint Options

Endpoint options with the `consumer.` prefix (for example, `consumer.delay=5000`) are no longer supported. The endpoint options should be changed to use the option name, without the `consumer.` prefix (for example, `delay=5000`).

# Calling a Processor as an Endpoint

Calling a processor as an endpoint using the syntax `<to uri="myProcessorBean"/>` is no longer supported. Use either the bean component (`<to uri="bean:myProcessorBean"/>`) or use the process call (`<process ref="myProcessorBean"/>`). For more information and for Java samples, see CALLING A PROCESSOR AS AN ENDPOINT.

# XML Element and Attribute Changes

The following elements and attributes have been changed:

- The `headerName` attribute of the `setHeader` element has been renamed to `name`.
  - Migrate `<setHeader headerName="foo">...</setHeader>` changes to `<setHeader name="foo">...</setHeader>`.
- The `propertyName` attribute of the `propertyHeader` element has been renamed to `name`.
  - Migrate `<setProperty propertyName="foo">...</setProperty>` changes to `<setProperty name="foo">...</setProperty>`.
- In Aggregate EIP, the optional nested elements `completionSize` and `completionTimeout` have been renamed to `completionSizeExpression` and `completionTimeoutExpression`.
  - Migrate `<aggregate ...><completionSize>...</completionSize></aggregate>` changes to `<aggregate ...><completionSizeExpression>...</completionSizeExpression></aggregate>`.
  - Migrate `<aggregate ...><completionTimeout>...</completionTimeout></aggregate>` changes

to `<aggregate ...><completionTimeoutExpression>...</completionTimeoutExpression></aggregate>`.

- The `custom` element of the load balancer EIP has been renamed to `customLoadBalancer`.
  - Migrate `<loadBalance ...><custom ref="xyz"/></loadBalance>` changes to `<loadBalance ...><customLoadBalancer ref="xyz"/></loadBalance>`.
- The `ref` attribute on the `marshal` and `unmarshal` elements, used to supply a reference to a custom data format, have been removed from those elements. The data format reference is now supplied using the `ref` attribute on the nested `custom` element.
  - Migrate `<unmarshal ref="decryptBean"/>` changes to `<unmarshal><custom ref="decryptBean"/></unmarshal>`.
- The package name of the `MemoryIdempotentRepository` class has been changed.
  - Migrate `<bean id="idempotentRepository" class="org.apache.camel.processor.idempotent.MemoryIdempotentRepository"/>` changes to `<bean id="idempotentRepository" class="org.apache.camel.support.processor.idempotent.MemoryIdempotentRepository"/>`.

For more information, see <setHeader> and <setProperty> in XML DSL, <AGGREGATE> EIP in XML DSL, and XML DSL Migration in the *Apache Camel 2.X to 3.0 Migration Guide*.

# Changes to the Netty Component

The `encoder` and `decoder` endpoint options were deprecated in the Camel 2.x Netty4 component. The endpoint options have also been removed in the Camel 3.x Netty component. Instead of using `encoder` and `decoder`, use the `encoders` and `decoders` endpoint options.

- Migrate `<to uri="netty4://...?encoder=#customEncoder"/>` changes to `<to uri="netty://...?encoders=#customEncoder"/>`.
- Migrate `<to uri="netty4://...?decoder=#customDecoder"/>` changes to `<to uri="netty://...?decoders=#customDecoder"/>`.

The `encoders` and `decoders` parameters accept a list of comma-separated references if you need to chain multiple encoders and decoders. For more information, see Using Multiple CODECS.

# Changes to the Camel Kafka Component

The `topic` option that was an endpoint option now becomes the endpoint name that replaces `brokers`. `brokers` is now the endpoint option.

- Migrate `<to uri="kafka:hostname:port?topic=myTopic"/>` changes to `<to uri="kafka:myTopic?brokers=hostname:port"/>`.

## Changes to the Camel Throttle Component

In the previous Camel version, the `Throttle` component was used to enclose all the steps in the route for which you intended to throttle the throughput. In the current Camel release, the `Throttle` component is now a preceding step to all the steps that are to be throttled.

- Migrate

```
<throttle>
   <constant>1000</constant>
   <to uri="myStep1"/>

   .........
</throttle>
```

changes to

```
<throttle><constant>1000</constant></throttle>

<to uri="myStep1"/>...........
```

# Changes Driven by the Introduction of the Spring Boot Framework

## RedisTemplate Bean

If you created any bean or beans of the class `org.springframework.data.redis.core.RedisTemplate`, then make sure that one of those beans has a bean ID of `redisTemplate`.

# Changes That Are Needed for Third-Party Adaptors to Run in Legacy Mode in

# SAS Business Orchestration Services 10.2

To run these SAS Business Orchestration Services 10.1 samples in SAS Business Orchestration Services 10.2 in legacy mode, complete these steps:

1. Copy all configuration files that are found in the **$VIYA_HOME/share/boss/samples/APP_NAME** folder to the **$VIYA_CONFIG/etc/boss** folder.

   **$VIYA_HOME** refers to **/opt/sas/viya/home**. **$VIYA_CONFIG** refers to **/opt/sas/viya/config** (unless it has been customized in your environment).

   After completing this step, you should have one or all of the following folders in your **$VIYA_CONFIG/etc/boss** folder:

   - data
   - groovy
   - mappings/inbound
   - spring
   - templates/outbound

2. Update the `BokuUtils.groovy` file.

   Import the `org.apache.commons.lang.StringEscapeUtils` object.

   Import the `groovy.json.StringEscapeUtils` object.

3. Update the `payfone-route-context.xml` file.

   Change `<bean id="payfoneTokenAggregator" class="org.apache.camel.util.toolbox.FlexibleAggregationStrategy"/>` to `<bean id="payfoneTokenAggregator" class="org.apache.camel.builder.FlexibleAggregationStrategy"/>`

4. Update the `camel-context.properties` file. Change

   ```
   data-provider.intellicheck.login-api-base-url=test-onlineid-jja.icmobil.com/api/logindata-provider.intellicheck.processphoto-api-base-url=test-onlineid-jja.icmobil.com/api/CamUploader/ProcessPhoto
   ```

   to

   ```
   data-provider.intellicheck.login-api-base-url=retailid-test.intellicheck.com/api/logindata-provider.intellicheck.processphoto-api-base-url=retailid-test.intellicheck.com/api/CamUploader/ProcessPhoto
   ```

5. To make code in the sample files that reside in the **$VIYA_CONFIG/etc/boss/spring** folder compatible with SAS Business Orchestration Services 10.2, see "Changes Driven by the Camel 3.x Upgrade" on page 110.

6. Start SAS Business Orchestration Services in legacy mode. Run the sample code as documented in the respective `Readme.md` file for each of the adapters.

# Start SAS Business Orchestration Services in Legacy Mode

1. Start SAS Business Orchestration Services 10.2 by running it in legacy mode using the following command:

   ```
   ./boss.sh stop start tail --context spring/camel-context-simple.xml
   ```

2. Check the **$VIYA_LOG/boss/boss.log** and the **$VIYA_LOG/boss/application.log** files to make sure that there are no issues when the SAS Business Orchestration Services process starts. `$VIYA_LOG` refers to the `/opt/sas/viya/config/var/log` folder.

# Appendix 4

# SAS Business Orchestration Services 10.2 Migration in Native Mode

# Overview

SAS Business Orchestration Services 10.2 has design changes and feature enhancements to make it easier to use. For a complete list of changes, see “What’s New in SAS Business Orchestration Services 10.2” on page vii. Although it is possible for you to use the previous version’s configuration files with minimal changes in legacy mode, further modifications are required in order to take advantage of all the new capabilities introduced in this release. This guide provides information about the changes needed to the configuration files to run them in native mode.

# Prerequisite

You are required to complete the migration steps to run the application in the *legacy* before you perform the migration steps provided in this appendix to run the application in *native* mode.

For information about the migration steps involved to run the application in legacy mode, see Appendix 3, “SAS Business Orchestration Services 10.2 Migration Guide for Running in Legacy Mode,” on page 109.

The commands to run the application in legacy mode are explained in “Running in Legacy Mode” on page 8.

# Migration Steps

The following are the typical steps involved in migration. You can skip some of the steps if you are not using the components or features addressed in those steps.

- Consolidate or move application properties: Place all externalized properties in the `application.properties` file located at the root of the configuration folder.
- Migrate Spring bean definitions: Starting from the XML file that contains the `CamelContext` definition, extract definitions for all beans, routes, and REST APIs used in your configuration into separate XML files. For information about the recommended file and folder layout, see "Key Folders and Files" on page 4.
- Migrate automatically configured resources.
- Remove the use of the deprecated classes in the bean definitions.
- Create any custom producer templates.
- Migrate the data type converter mapping files.
- Migrate to use the built-in data formats to marshal and unmarshal content.
- Migrate to use the new sas-ode component to call SAS OnDemand Decision Engine.
- Migrate to use the new store and forward feature.

Further details about these steps are covered in the sections below.

# Consolidate or Move Application Properties

In SAS Business Orchestration Services 10.2, you should place all externalized application properties in the `application.properties` file in the root of the configuration folder, `$BOSS_CONFIG`. If you prefer to use the YAML syntax, use the `application.yml` file.

- Locate all the properties files in your existing configuration and consolidate all the files into one `application.properties` file.
- If there is a property named `boss.setinit.location` in your old properties file, then remove it.

  The `setinit.txt` file that is already at the root of the configuration folder is the most current one.

# Migrate Spring Bean Definitions

In SAS Business Orchestration Services 10.2, you should place all files that have bean definitions in the **$BOSS_CONFIG/beans** folder. You can have as many bean definition XML files in this folder as necessary. For ease of maintenance, you should consolidate closely related bean files into a single file. You should also name the files to indicate the type of beans that it contains.

A template for the bean definition file is provided in "Bean Definitions" on page 108.

Starting from the XML file that contains the `CamelContext` definition, extract all bean definitions and place them into a file under the **$BOSS_CONFIG/beans** folder. The `CamelContext` definition should be similar to the following:

```
<camelContext id="camelContext" xmlns="http://camel.apache.org/schema/
spring" messageHistory="true"/>
```

> **TIP** There are two types of uses for the `bean` XML element in Camel configurations. One is to define Spring beans that are registered in the application context and are accessed by other beans or services. The other is to invoke a bean method to process the message inside a route. The beans being migrated in this section are the Spring bean definitions that get registered in the application context. The `bean` elements that are used inside the routes are migrated during route migration.

The XML snippets for the Spring bean definitions to migrate look like the following:

```
<bean id="..." ...>
  <!-- Constructor args and properties -->
</bean>

or

<bean id="..." .../>
```

Here is an example of a bean that is references as part of a route:

```
<route id="multitenantFromJson">
  ...
  <bean ref="someBeanId" method="someMethod"/>
  ...
</route>
```

Beans that are instantiated from Groovy scripts look like the following:

```
<lang:groovy id="..." script-source="classpath:groovy/
<className>.groovy"/>
```

When adding these Groovy beans to an XML file in the **$BOSS_CONFIG/beans** folder, you are required to include the Spring `lang` namespace in the namespace declarations in that XML file. The highlighted sections in the following snippet show the `lang` namespace declarations that need to be present.

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:lang="http://www.springframework.org/schema/lang"

  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd

 http://www.springframework.org/schema/lang http://
www.springframework.org/schema/lang/spring-lang.xsd">
```

XML files can be imported into other XML files using the `import` element. Look for these elements in old configuration files and traverse the imported files to repeat the process of moving bean definitions from those files. The `resource` attribute of the `import` element specifies the file being imported. This process needs to be completed recursively until all bean definitions have been extracted and moved to files in the **`$BOSS_CONFIG/beans`** folder.

At this point, you should still be able to launch the application in legacy mode without any errors.

# Migrate Auto-Configured Resources

Because SAS Business Orchestration Services 10.2 uses Spring Boot, some of the configuration that you did manually by using bean definitions in XML files is now achieved automatically by the application. You can eliminate these bean definitions in the XML files and specify the appropriate properties directly in the `application.properties` file. The following lists the bean definitions that are commonly found in configuration files for prior releases and how to use the Spring Boot auto-configuration for them. The class names specified below or a subclass of the class names is the value specified for the class attribute of the bean element.

- **org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer:** This bean is automatically configured by the application. It is no longer necessary to define this bean in the XML file.
  - If you are using the Jasypt library for property encryption to secure property values, see “Encrypting Credentials” on page 102 for information about how to enable that functionality.
- **javax.sql.DataSource:** Because this is actually an interface, beans of this type use a concrete class. Implement this interface as the value for the class attribute on the bean element.
  - If you are using the Apache Commons DBCP, look for beans with the DataSource implementation classes provided by that library.
  - If you are using the C3P0 library in your configuration for DataSource implementations, you should migrate to using DBCP because of the native integration that Spring Boot provides for the DBCP library. If you prefer not to make this change, keep the C3P0 DataSource bean definition as it is and no further migration is needed.
  - If you use non-pooled DataSource implementations, you should switch to using one of pooled implementations for better application performance.

Pooled DataSource implementations manage the connections to the database more efficiently, avoiding connection overhead whenever possible.

- After identifying these beans, the properties set in the XML files need to be transferred to the `application.properties` file with the appropriate keys. For more information, see the Spring Boot Reference Guide .

Here is an example:

```
<bean name="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="${MySql_Server_Port}"/>
    <property name="username">
        <bean class="com.sas.finance.fraud.ol.secure.OLSecureUsername">
            <constructor-arg name="identity" value="$
{MYSQL_CREDENTIAL}"/>
        </bean>
    </property>
    <property name="password">
        <bean class="com.sas.finance.fraud.ol.secure.OLSecurePassword">
            <constructor-arg name="identity" value="$
{MYSQL_CREDENTIAL}"/>
        </bean>
    </property>
</bean>
```

This would be set up automatically if you specified these properties in the `application.properties` file.

```
spring.datasource.type=org.springframework.jdbc.datasource.DriverManage
rDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=<database JDBC connection URL, which would look
like jdbc:<vendor>://<host>:<port>/<databaseName>. Refer to database
driver documentation for precise syntax.>
spring.datasource.username=<database username>
spring.datasource.password=<database password>
```

**TIP** The classes `com.sas.finance.fraud.ol.secure.OLSecureUsername` and `com.sas.finance.fraud.ol.secure.OLSecurePassword` have been deprecated in SAS Business Orchestration Services 10.2. Values provided using those classes for user names and passwords should be specified directly with the appropriate properties in the `application.properties` file. To encrypt passwords for security, see “Encrypting Credentials” on page 102.

If you are using Apache DBCP for DataSource implementations, refer to the DBCP library documentation for the property names that can be specified with the prefix `spring.datasource.dbcp2`. Use `*` in the `application.properties` files to customize the run-time behavior of the DBCP DataSource implementations.

If you have defined only one DataSource bean in your configuration, then delete the corresponding XML bean definitions once the properties for the DataSource beans are migrated to the corresponding properties in the `application.properties` file. The DataSource bean that was created automatically can be referenced with the bean ID dataSource. If the original bean had a different SAS Business Orchestration Services identifier assigned, then replace all references to that bean with the previous identifier with the dataSource identifier.

**TIP** If you have defined more than one DataSource bean in your configuration, then you can instantiate only one using the properties in the `application.properties` file. In such cases, configure the primary DataSource using the properties and keep the bean definitions for the others as they were. The `com.sas.finance.fraud.ol.secure.OLSecureUsername` and `com.sas.finance.fraud.ol.secure.OLSecurePassword` classes can still be eliminated in the XML by using the property placeholders.

- **org.springframework.jdbc.core.JdbcTemplate:** Spring Boot creates an instance of this bean automatically when a DataSource bean is configured. Spring Boot provides the following three properties if you want to customize the behavior of this bean. Add these properties to the `application.properties` file, if you want to customize the behavior of this bean. For more information about these properties, see Spring Boot Common Application Properties.

```
spring.jdbc.template.fetch-size
spring.jdbc.template.max-rows
spring.jdbc.template.query-timeout
```

**TIP** If you have defined only one `JdbcTemplate` bean in your XML files, then delete the XML bean definition. If you have defined more than one `JdbcTemplate` bean, then you can delete the one linked to the primary data source. The `JdbcTemplate` bean that was created automatically can be referenced with the `jdbcTemplate` bean ID. If the original bean had a different identifier assigned, then replace all references to that bean with the previous identifier with the `jdbcTemplate` identifier.

- **ActiveMQ connection beans:** If you have been using an XML configuration to define the beans for an ActiveMQ connection, you should be able to configure an ActiveMQ connection by setting the auto-configuration properties in the **`${BOSS_CONFIG}/application.properties`** file. The ActvieMQ auto-configuration properties can be found at Spring Boot Common Application Properties. The following is an example of an XML configuration of an ActiveMQ connection and the equivalent in the **`${BOSS_CONFIG}/application.properties`** file. Once a bean definition has been migrated to the **`${BOSS_CONFIG}/application.properties`** folder, that XML bean definition must be deleted.

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

<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-
beans-3.0.xsd">

    <bean id="pooledConnectionFactory"
class="org.apache.activemq.pool.PooledConnectionFactory" init-
method="start" destroy-method="stop">
        <property name="maxConnections" value="10"/>
        <property name="maximumActiveSessionPerConnection" value="10"/>
        <property name="connectionFactory">
```

```
            <bean
class="org.apache.activemq.ActiveMQConnectionFactory">
                <property name="brokerURL" value="tcp://
localhost:61616?wireFormat.maxInactivityDuration=0"/>
            </bean>
        </property>
    </bean>

    <bean id="jmsConfig4ActiceMq"
class="org.apache.camel.component.jms.JmsConfiguration">
        <property name="connectionFactory"
ref="pooledConnectionFactory"/>
        <property name="transacted" value="false"/>
        <property name="concurrentConsumers" value="2"/>
        <property name="maxConcurrentConsumers" value="10"/>
        <property name="requestTimeout" value="10000"/>
    </bean>

    <bean id="activemq"
class="org.apache.camel.component.jms.JmsComponent">
        <property name="configuration" ref="jmsConfig4ActiceMq"/>
    </bean>
</beans>
```

The following shows the equivalent for the above using ActiveMQ auto-connection properties:

```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=10
spring.jms.listener.concurrency=2
spring.jms.listener.max-concurrency=10
spring.jms.listener.acknowledge-mode=false
spring.activemq.send-timeout=10000
```

- **IBM MQ connection beans:** You must continue using an XML configuration to define the beans for an IBM MQ connection. This is due to the fact there are no auto-configuration properties for a message converter for IBM MQ.
- **Redis beans:** You must continue using an XML configuration to define the beans for a Redis connection. This is because there are no auto-configuration properties to configure the key and value serializers that might be needed to configure the RedisTemplate.

# Remove Deprecated Classes in Bean Definitions

## Dealing with Resource Credentials

The following classes have been deprecated in SAS Business Orchestration Services 10.2:

```
com.sas.finance.fraud.ol.secure.OLSecureCredentialStore

com.sas.finance.fraud.ol.secure.OLSecureUsername

com.sas.finance.fraud.ol.secure.OLSecurePassword
```

These classes were used to read a text file that contained credentials for various resources. In SAS Business Orchestration Services 10.2, these credentials should be provided in the `application.properties` file. If you want to keep the credentials in a separate file, then you can use Spring profiles, and enable those profiles that are appropriate for the environment. For more information, see Profile-specific Properties.

If you want to use property encryption to secure the property values, then see "Encrypting Credentials" on page 102.

## Credentials for Resources with Auto-configuration Support

For information about how to configure credentials for the resources that are supported by the Spring Boot auto-configuration mechanism, see "Migrate Auto-Configured Resources" on page 121 .

> **TIP** If you have defined more than one bean for the resources that are supported by auto-configuration (for example, DataSources), auto-configuration can be used for the primary bean while all additional beans must be defined through the bean definitions in the XML files while externalizing property values in the `application.properties` file. Follow the process outlined in the next section to update those bean definitions and move the credentials into the `application.properties` file.

# Credentials for Resources without Auto-configuration Support

If any of the credentials referred to are for a custom resource that is not supported by the Spring Boot auto-configuration mechanism, those properties must be added to the `application.properties` file with unique keys. You then reference them with placeholders in the XML bean definitions.

For example, the following shows a custom bean without auto-configuration support defined with credentials:

```
<bean id="tokenService"
class="com.sas.finance.fraud.ol.tokenization.TokenService">
    <constructor-arg name="user">
        <bean class="com.sas.finance.fraud.ol.secure.OLSecureUsername">
            <constructor-arg name="identity" value="$
{SVT_CREDENTIAL}"/>
        </bean>
    </constructor-arg>
    <constructor-arg name="passwd">
        <bean class="com.sas.finance.fraud.ol.secure.OLSecurePassword">
            <constructor-arg name="identity" value="$
{SVT_CREDENTIAL}"/>
        </bean>
    </constructor-arg>
    <!-- more constructor-args and properties -->
</bean>
```

This could be refactored to look like the following:

```
<bean id="tokenService"
class="com.sas.finance.fraud.ol.tokenization.TokenService">
    <constructor-arg name="user" value="$
{com.mycompany.tokenization.service.username}"/>
    <constructor-arg name="passwd" value="$
{com.mycompany.tokenization.service.password}"/>
    <!-- more constructor-args and properties -->
</bean>
```

In the example above, `com.mycompany.tokenization.service.username` and `com.mycompany.tokenization.service.password` are the two new properties created in the `application.properties` file to hold the user name and password for the tokenization service. You could come up with any string that you would like to use as the property name. You should separate your custom properties with an appropriate prefix (for example, `com.mycompany` as shown in the example above) so that it is easier to identify which properties are custom to your environment and are not the property keys that are built into the product.

If any of these deprecated classes are used in other bean definitions, remove those occurrences following the same process.

# Remove Artifacts

After all the resource credentials have been transferred to the `application.properties` file and bean definitions have been modified, perform the following:

- Remove any beans of type `com.sas.finance.fraud.ol.secure.OLSecureCredentialStore` that are in your XML files.
- Remove the text file that was originally used to contain these credentials.
- Remove the sections in your `application.properties` file that look like the following:

```
credentialFilePath=syscfg/credentials.txt
         MYSQL_CREDENTIAL=SFMOL MySql
         ORACLE_CREDENTIAL=SFMOL Oracle
         SVT_CREDENTIAL=SFMOL TOKENIZATION
```

# Remove the DatasourceRepository Bean Definition

The `com.sas.finance.fraud.ol.enrich.DatasourceRepository` class has been deprecated in SAS Business Orchestration Services 10.2. Bean definitions of this class can be deleted. These bean definitions look like the following:

```
<bean id="dataSourceRepo"
class="com.sas.finance.fraud.ol.enrich.DatasourceRepository">
        <constructor-arg name="sources">
            <map>
                <entry key="derby" value-ref="dataSource"/>
            </map>
        </constructor-arg>
    </bean>
```

These beans were used to make `DataSource` beans accessible inside mapping files. SAS Business Orchestration Services 10.2 uses a different mechanism that removes the need to create a separate repository.

**TIP** The `DataSource` beans should still be instantiated either using the auto-configuration mechanism (that is, using only property values in the `application.properties` file) or by a combination of properties and XML bean definitions when there is more than one bean of the same type in the application. For more information, see "Migrate Auto-Configured Resources" on page 121.

# Migrate Route Definitions

SAS Business Orchestration Services 10.2 requires you to place all files that have Apache Camel route definitions in the **$BOSS_CONFIG/routes** folder. You can have as many XML files in this folder as necessary. For ease of maintenance, you should consolidate all closely related routes into a single file You should name the files to indicate the type of routes that it contains.

In the case of Spring bean definitions, start with the file that contains the `camelContext` definition and go through all the other files to find routes that are used in your configuration. When other files with routes are being linked from a configuration file, there are XML snippets that look like the following:

```
<routeContextRef ref="safRoutes"/>
```

This links to other routes defined in another XML file with the corresponding `routeContext` definition that looks like the following:

```
<routeContext id="safRoutes" xmlns="http://camel.apache.org/schema/
spring">
        <!-- One or more routes defined here -->
    </routeContext>
```

If the existing organization for routes in separate files is still good and if you want to keep that organization, then move all those routes into a dedicated file under the **$BOSS_CONFIG/routes** folder. “Bean Definitions” on page 108 provides a template for the route definition file.

The new file that contains the routes does not have the `routeContext` element in it. The equivalent, as the root element of the XML document, is the `routes` element, as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
    <routes id="safRoutes" xmlns="http://camel.apache.org/schema/
spring">
        <!-- One or more routes defined here -->
    </routes>
```

There is no need to use the `routeContextRef` element in other files to make these routes part of the application configuration. The application automatically loads all routes defined in the files in the **$BOSS_CONFIG/routes** folder that have the .xml extension.

# Migrate REST Endpoint Definitions

## Overview

SAS Business Orchestration Services 10.2 requires that you place all files that have Apache Camel REST endpoint definitions in the **$BOSS_CONFIG/rest** folder. You can have as many XML files in this folder as necessary. For ease of maintenance, you should consolidate all closely related REST endpoints into a single file. You should also name the files to indicate the type of REST endpoints that it contains.

In the case of Spring bean definitions and route definitions, start with the file that contains the `camelContext` definition and go through all the other files to find the REST endpoints that are used in your configuration. When other files with REST endpoints are being linked from a configuration file, there are XML snippets that look like the following:

```
<restContextRef ref="safRest"/>
```

This links to other REST endpoints that are defined in another XML file with the corresponding restContext definition that looks like the following:

```
<restContext id="safRest" xmlns="http://camel.apache.org/schema/
spring">
        <rest path="..." ...>
            <!-- One or more REST endpoints (get, post, etc.) defined
here -->
        </rest>
    </restContext>
```

If the existing organization for REST endpoints in separate files is still good and if you want to keep that organization, move all those REST endpoints into a dedicated file under the **$BOSS_CONFIG/rest** folder. "Bean Definitions" on page 108 provides a template for the REST endpoint definition file.

The new file that contains the folder does not have the `restContext` element in it. The equivalent, as the root element of the XML document, is the `rests` element, as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
    <rests id="safRest" xmlns="http://camel.apache.org/schema/spring">
        <rest path="..." ...>
            <!-- One or more REST endpoints (get, post, etc.) defined
here -->
        </rest>
    </routes>
```

There is no need to use the `restContextRef` element in other files to make these REST endpoints part of the application configuration. The application automatically loads all the REST endpoints that are defined in files in the **$BOSS_CONFIG/rest** folder that have the .xml extension.

Although a single XML file can have more than one nested `rest` element inside a `rests` element (the root element),you should place them in separate files if the REST endpoint paths are not related.

# Customizing the REST API Configuration

SAS Business Orchestration Services 10.2 creates the `RestConfiguration` bean automatically using the properties in the `application.properties` file. The auto-configured bean uses Jetty to provide consumer functionality (that is, a RESTful service endpoint is exposed in the application).

In previous releases, an XML bean definition was used to achieve similar functionality. The following shows an example of a bean definition:

```
<restConfiguration component="jetty" host="{{rest_hostname}}"
port="{{rest_listen_port}}" bindingMode="off" enableCORS="true">
        <!-- Other property configurations -->
    </restConfiguration>
```

To achieve the equivalent functionality after migration, set the host and port for the REST consumer service using the `camel.rest.host` and `camel.rest.port` properties. For the list of properties that can be specified in the `application.properties` file, see Camel REST component. You can then remove the `restConfiguration` bean.

# Migrate Producer Templates

In SAS Business Orchestration Services 10.2, a default instance of the `ProducerTemplate` bean is automatically created. If you have customized instances of these classes, convert the template instance creation XML as follows for these releases.

In SAS Business Orchestration Services 10.1, the customized `ProducerTemplate` can be created within the `CamelContext` definition in an XML file. The following code snippet creates a `ProducerTemplate` bean with the bean ID of producerRecordOffline, which uses direct:recordOffline as its default endpoint URI.

```
<template id="producerRecordOffline"
defaultEndpoint="direct:recordOffline"/>
```

In SAS Business Orchestration Services 10.2, the above `ProducerTemplate` bean must be created using a bean definition as shown below:

```
<bean id="producerRecordOffline"
class="org.apache.camel.impl.engine.DefaultProducerTemplate" init-
method="start">
        <constructor-arg name="camelContext" ref="camelContext"/>
        <property name="defaultEndpointUri"
value="direct:recordOffline"/>
    </bean>
```

# Migrate to the New Data Type Converters

## Overview

The data type converter feature, also referred to as the data type mapper feature, has gone through a major overhaul in SAS Business Orchestration Services 10.2. This was done for enhanced usability and enhanced functionality to support the following:

- use of familiar component architecture: eliminated the use of utility beans by introducing the `sas-convert-body-to` component
- complex bean and Map types that contain nested objects, mapped properties, indexed properties, and object references
- all Java `Collection` and `Iterable` types
- all primitive Java data types are also supported, compared to only string values in prior releases
- support for Date and Time classes in the `java.time` package in addition to the `java.util.Date` package
- fully standards-compliant XPath
- JSON Pointers
- formatted input and output string values representing numeric, date, and time values
- named parameters in Groovy scripts
- import statement in Groovy scripts

These changes enable you to eliminate custom Groovy code that was necessary for earlier releases. The structure of the XML mapping files has also been simplified to make it easier to write and manage configurations.

You should go through “Overview” on page 16 before attempting to migrate the configuration from previous releases. It would also help to create a few data type converters in a SAS Business Orchestration Services 10.2 installation from scratch to get familiar with the file layout and the new features. The sections below highlight the major changes that you should be aware of while migrating data type converter configurations, also called mapping files, from earlier releases.

# Mapping File Location

The data type converter configuration files (that is, the mapping files) are in the **$BOSS_CONFIG/converters** folder. In previous releases, they were typically placed in the **$BOSS_CONFIG/mappings** folder.

All files with an `.xml` extension placed in the `$BOSS_CONFIG/converters` folder are automatically loaded during the application start-up and are watched for changes once the application is running. SAS Business Orchestration Services attempts to process changes that are made to these files without the need to restart the application. The addition of new mapping files and the removal of existing mapping files is also supported. This makes the corresponding data type converter become available or removed accordingly. Adding new converters and removing existing converters without modifying the routes in the application is useful only if dynamic URIs are being used in the routes that use these converters.

Errors in data mapping files do not stop SAS Business Orchestration Services from starting. Rather, errors are printed to the application log and the application continues with the start-up operation. For any data type converter with errors in the mapping file, exceptions are raised at run time when the data type converter gets invoked using the `mapperId` attribute provided in the mapping file.

> **TIP** All mapping files must be placed in the **$BOSS_CONFIG/converters** folder. Subdirectories are not supported. Mapping files placed in a subdirectory do not get loaded by the application.

# Invoking Data Type Converters

In previous releases, invoking data type converters was done through various utility beans, either provided as part of the product or custom beans that you created to suit your implementation. Message headers and exchange properties were also commonly used to communicate contextual information into the data type conversion bean methods:

```
<setHeader name="mapperId"><constant>myCustomMapper</constant></
setHeader>
    <bean
beanType="com.sas.finance.fraud.ol.xmlmapping.utils.XmlMapperUtil"
method="mapToTransaction"/>
```

This mechanism has been streamlined in SAS Business Orchestration Services 10.2. A new component named `sas-convert-body-to` has been introduced to provide a streamlined way to invoke data type converters. This is modeled similarly to the `convertBodyTo` feature of Camel. Whereas the `convertBodyTo` feature of Camel works without any additional input and is limited to the data type converters that come with the product, the `sas-convert-body-to` component allows custom data type converters to be configured using XML mapping files. For information

about how to use the `sas-convert-body-to` endpoint, see "Data Type Converters" on page 17.

# Mapping File XML Schema Changes

## Overview

In order to simplify the XML mapping file structure and to make it easier to use, the XML schema for the data type converter mapping files has gone through some changes. The new XSD schema is named `fieldMapper-10.2.xsd` and is available in the `boss-api.jar` file. This file is installed in the **`$VIYA_HOME/share/boss/lib`** folder, which is located in the **`converters/schema`** folder inside the JAR file. A template for the new data type converter mapping file can be found in "Converter Mapping File" on page 108.

## Split Mapping Files Containing Multiple Mappers

The first change you would notice with the new mapping file structure is that each mapping file contains file mapping for only one data type converter. Previous releases allowed more than one message element to be nested inside a `messages` element. You are now required to create separate XML files for each data type converter.

# fieldMapper Element

The `fieldMapper` element is the root element in the new XML structure for the mapping files. This element is approximately equivalent to the message element that existed before, holding all the field mappings for a given data type converter. Because this is the root element of the XML document, general namespace declarations are placed in this element. For more information about the `fieldMapper` element and its use, see "The fieldMapper Element" on page 23..

This element accepts two attributes, `sourceType` and `targetType`, to indicate the source object type expected as input and the target object type created by the converter. The `messageType` and `subType` attributes that existed on the message element in the previous releases are no longer available. The `mapperId` attribute has been renamed as `id`. The value for the `id` attribute must be unique among all the data type converters defined in the application as this is the handle used to invoke a specific data type converter.

# field Element

The `field` elements are no longer nested inside a `fields` element, but are nested directly in the `fieldMapper` element. The `field` element serves the same purpose as in previous releases, providing the specification for the field to be populated in the target object.

The `name` attribute of the `field` element accepts a more complex notation to work with simple properties, nested properties (that is, properties of an object nested inside another object), indexed properties (that is, elements of arrays or lists), and mapped properties (that is, key for an entry in a Map).

The `field` element now has a `type` attribute to indicate the data type of the value to be set in the target object. If the input value is not of the type specified here, the data type converter attempts to convert using any of the available data type converters in the product. If the data type is a complex object type, a value can be supplied for the `mapperId` attribute to perform the conversion using another data type converter that is registered in the application. Two more attributes that are new on the `field` element are `pattern` and `allowZeroLengthString`.

The functionality of the `default` and `override` attributes that existed in previous releases have been changed. The `override` attribute is no longer available and the `default` attribute works more like what the `override` attribute used to achieve before: that is, to use the given value if the input value is null.

For more information about the field element and all its attributes, see “The field Element” on page 24..

**TIP** The `field` element cannot be empty in SAS Business Orchestration Services 10.2. It must have a nested element in it to supply the input value for the field. Previous releases allowed you to use the default attribute of the `field` element to supply a constant value to the target field. An equivalent configuration in SAS Business Orchestration Services 10.2 is to nest a `constant` element with the constant value. For example, `<field name="acct_type" default="CS"/>` would need to be configured as the following:

```
<field name="acct_type">
        <constant>CS</constant>
    </field>
```

# sourceField Element

The `sourceField` element replaces the `srcField` element in previous releases. It serves the same purpose as before with some additional functionality. The `sourceField` element is used to refer to a property in the input object.

Similar to the field element, the name attribute of the sourceField element also supports complex notation to refer to simple properties, nested properties, indexed properties, and mapped properties.

The type and pattern attributes are also similar in functionality to the attributes found on the field element. The type attribute value indicates the type of the object to be expected as input. The pattern attribute provides a pattern to convert a string value to the given type. The pattern attribute is used along with the type attribute only when the input value is a string that needs to be converted into another data type. Conversion of string values to numeric, date, and time data types is supported.

In previous releases, a Groovy element could be nested inside a srcField element. This is no longer allowed. The only elements that can be nested inside the sourceField element are xmlQueryNamespace, xmlQuery, and jsonQuery. These elements are used in rare circumstances when the input is actually an XML or JSON string and a part of that value needs to be extracted using the XPath or JSON Pointer expression. For more information about the sourceFieldelement, see "The sourceField Element" on page 25.

For information about how to migrate the Groovy code nested inside the srcField element, see "custom Element" on page 135.

# constant Element

The constant element has no changes. As in previous releases, it can contain simple string values or XML CDATA blocks.

# custom Element

The functionality that enabled you to embed Groovy language scripts in the srcField element, dbField element, and so on has been refactored in SAS Business Orchestration Services 10.2. Groovy language code can now be nested only inside the custom element using the groovy element. A custom element can be nested only inside a field element.

In addition to the groovy element containing the Groovy language code, the custom element allows param elements. The param elements are used to define named parameters that are automatically passed into the Groovy script at run time.

By using the valueSource and valueSourceType attributes of the param element, you can make the following four types of values accessible to your Groovy language code via a named parameter:

- SOURCE_FIELD: a field value from the input object supplied to the data type converter. The valueSource attribute takes values similar to the name attribute on the sourceField element.
- TARGET_FIELD: a field value from the output object created by the data type converter. The valueSource attribute takes values similar to the name attribute on the field element.

**Note:** The data type converter processes output field values in the order that they are declared in the mapping file. Accessing a target field before it is processed would result in a null value being passed into the Groovy language code for the named parameter.

- `PROPERTY`: a property that is defined in the application. The `valueSource` attribute takes values of the form `{{com.mycompany.property.key}}`.
- `BEAN`: a bean registered in the application registry. The `valueSource` attribute takes the identifier that is used to register the bean and takes the form `#jdbcTemplate` where `jdbcTemplate` is the bean identifier. The # sign is optional, but it is recommended for clarity and readability.

There is no limit on the number of named parameters. This enables you to use the `custom` element wherever Groovy script was embedded in previous releases. If the Groovy code was used to do data type conversions, check if that conversion functionality is covered by the new features introduced in SAS Business Orchestration Services 10.2. If the new features cover the conversion, the use of Groovy code could be eliminated completely. For use cases where custom logic is still needed, use the new `custom` element to achieve the same functionality. A common use case for the use of Groovy script in previous releases was to pull data from external data sources like a database server to enrich the data converter output. That use case is covered in more detail in the next section.

For more information about how to use the `custom` element with the Groovy script and named parameters, see "The custom Element" on page 28.

# Enrichment Using External Data Sources

In previous releases, a mapping file performing data enrichment by using data from a database server might look like the following:

```
<field name="rua_8byte_string_003">
        <dbField rscName="derby" default="XYZ">
            <groovy>
                <![CDATA[
                    String sql = 'select value from enrichments where
unqKey=\'0000000000000001038\' and fieldname=
\'rua_8byte_string_003\'';
                    return jdbcTemplate.queryForObject(sql,
String.class);
                ]]>
            </groovy>
        </dbField>
    </field>
```

In SAS Business Orchestration Services 10.2, the fragment would look like the following:

```
    <field name="rua_8byte_string_003" default="XYZ">
        <custom>
            <param name="jdbcTemplate" valueSourceType="BEAN"
valueSource="#jdbcTemplate"/>
            <groovy>
```

```
                <![CDATA[
                    String sql = 'select value from enrichments where
unqKey=\'0000000000000001038\' and fieldname=
\'rua_8byte_string_003\'';
                    return jdbcTemplate.queryForObject(sql,
String.class);
                ]]>
            </groovy>
        </custom>
    </field>
```

The key points to note are the following:

- The `name` attribute of the `param` element has been set as `jdbcTemplate` because that was the variable name used inside the Groovy code. It could have been named anything, as long as it matches the use inside the code.
- The `valueSourceType` attribute of the `param` element has been set as `BEAN` to indicate that the value provided for the parameter is a reference to a bean that has been defined in the application registry.
- The `valueSource` attribute of the `param` element has been set as `#jdbcTemplate` to refer to the instance of the `JdbcTemplate` bean that was automatically created using the properties specified in the `application.properties` file. For more information, see . on page 121 If the `JdbcTemplate` instance was configured through XML or it has a name other than `jdbcTemplate`, this value should be adjusted to match that name (for example, `txnJdbcTemplate`). It is not necessary to make the bean identifier the same as the parameter name. The parameter name must match how the value is accessed in the Groovy code, whereas the bean reference must match the identifier used to register the bean in the Spring application context.

# Migrating XPath Functionality

In previous releases, there was a feature that was used to parse XML files to extract element values and attributes. The implementation was a bit nonstandard. The feature has been redesigned in SAS Business Orchestration Services 10.2 and is now fully XPath standard compliant.

The implementation in the earlier releases was nonstandard because of the following:

- It did not support all the features of the XPath standard like filtering expressions and so on.
- There was a short-circuiting mechanism used that looked for a start tag and then all the paths were coded relative to this start tag.

Invoking this functionality also needed multiple beans to work together, as shown in the sample below:

```
<lang:groovy id="gvyIso20022XmlToTransaction">
        <lang:inline-script>
            <![CDATA[
                import java.util.function.Function;
                import
com.sas.finance.fraud.ol.xmlmapping.utils.XmlMapperUtil;
```

```
                { msg -> XmlMapperUtil.xmlToTransaction(msg,
"iso20022", "CstmrCdtTrfInitn") } as Function
            ]]>
        </lang:inline-script>
    </lang:groovy>


    <bean id="flexIso20022InboundXmlMapper"
class="com.sas.finance.fraud.ol.mapper.InBoundMapper">
        <constructor-arg name="mapper"
ref="gvyIso20022XmlToTransaction"/>
    </bean>


    <route id="...">
        <from uri="..."/>
        <bean id="Iso20022_Xml_Mapper"
ref="flexIso20022InboundXmlMapper"/>
        <!-- more config -->
    </route>
```

The functionality was invoked with a bean as a processor step in the route while using multiple utility classes
like `com.sas.finance.fraud.ol.mapper.InBoundMapper`, `com.sas.finance.fraud.ol.xmlmapping.utils.XmlMapperUtil`, and so on. The value `CstmrCdtTrfInitn` is being passed as the start tag with the entries in the corresponding mapping file, iso20022, using relative paths as shown below:

```
<field name="TBT_GROUP_CNT">
        <srcField name="CstmrCdtTrfInitn/GrpHdr/NbOfTxs"/>
    </field>
```

In previous releases, this configuration worked even though the actual XML document structure might have looked like the following. Notice that the root element of the XML content is the `Document` element. The `CstmrCdtTrfInitn` element is nested inside the `root` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<Document xmlns="urn:iso:std:iso:20022:tech:xsd:pain.001.001.03">
    <CstmrCdtTrfInitn>
        <GrpHdr>
            <MsgId>000001</MsgId>
            <CreDtTm>20120502T14:52:09</CreDtTm>
            <NbOfTxs>100</NbOfTxs>
        </GrpHdr>
    </CstmrCdtTrfInitn>
</Document>
```

Any namespaces that are declared in the document were also being ignored. The document was being processed as if there were no namespaces associated with the document.

In SAS Business Orchestration Services 10.2, the functionality is fully compliant with the XPath standard and the need for all these other utility classes and beans has been eliminated. The new implementation is fully aware and capable of processing namespaces.

An equivalent configuration would look like the following:

```
<fieldMapper xmlns="https://www.sas.com/boss/fieldMapper-10.2"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="https://www.sas.com/boss/
fieldMapper-10.2 fieldMapper-10.2.xsd"
                 targetType="java.util.Map"
                 id="iso20022">
        <xmlQueryNamespace prefix="txn"
uri="urn:iso:std:iso:20022:tech:xsd:pain.001.001.03"/>

        <field name="TBT_GROUP_CNT">
            <xmlQuery expression="/txn:Document/txn:CstmrCdtTrfInitn/
txn:GrpHdr/txn:NbOfTxs/text()"/>
        </field>

        <!-- more fields -->
    </fieldMapper>
```

The following are things to note:

- The `xmlQueryNamespace` element associates a prefix to the default namespace. This association is needed to be able to refer to elements that are associated with the default namespace. If the XML file has defined a prefix for a given namespace, that prefix can be used without any additional declarations in the mapping file.
- The `expression` attribute of the `xmlQuery` element starts with a `/`.
- The `expression` attribute starts from the root of the document.

  **Note:** Given the structure of this simple document, there are other ways to refer to the fields without providing the path from the root of the document. For more information, see XPath documentation.

- Unless there are no namespaces associated with the XML document, a namespace prefix is required in all element references. In the example above, `txn:` is used as the prefix to refer to the elements from the default namespace in the XML document.
- `/text()` is needed in the expression to read the text content of an element.

Use the following to invoke this data type converter:

```
<to uri="sas-convert-body-to:?mapperId=iso20022"/>
```

There is no need for any additional beans and utility classes.

Because SAS Business Orchestration Services 10.2 is fully compliant with the XPath standard, you can use any legal XPath expression. This includes accessing collections of values, attributes, filtering expressions, and so on. For more information, see XPath reference material.

# Convert to a SAS OnDemand Decision Engine Transaction

In SAS Business Orchestration Services 10.2 native mode, you should use a multi-step process to replace the `requestDispatcher` bean for converting incoming messages in various formats to a SAS OnDemand Decision Engine transaction.

For an ISO 8583 message in SAS Business Orchestration Services 10.1, there was a `requestDispatcher` bean to convert the ISO 8583 message directly to a transaction. The Camel Exchange header is used to tell the bean what message format it is. In this case, it is ISO 8583. The following route shows its use in the route.

```
<bean ref="requestDispatcher" method="process"/>
```

In SAS Business Orchestration Services 10.2, a three-step process is recommended:

1. Convert the ISO 8583 message to a Map with field numbers as keys using the Iso8583 data format. For more information, see Chapter 6, “Working with Data Formats,” on page 61.
2. Convert the Map to another Map with the `Transaction` field names as the keys using the data type converter. For more information, see “Data Type Converters” on page 17.
3. Convert the output map from the previous step to a transaction using the `OdeUtils` class bean.

Here is an example of the three-step process.

```
<unmarshal><custom ref="sas-iso8583"/></unmarshal>
       <to uri="sas-convert-body-to:?mapperId=iso8583ToOdeTxn"/>
       <bean beanType="com.sas.finance.fraud.ol.utils.OdeUtils"
method="toTransaction2"/>
```

# Migrate to the New Data Formats

## JSON Data Format

In SAS Business Orchestration Services 10.1, the `Gson` bean or utility class was used to parse JSON messages. In SAS Business Orchestration Services 10.2, you should use the Camel JSON data format with the Jackson library for parsing JSON messages. For more information, see “JSON Data Format” on page 63.

# ISO 8583 Data Format

In SAS Business Orchestration Services 10.1, the `Iso8583SerializerRepository` bean class was used to parse an ISO 853 message. In SAS Business Orchestration Services 10.2, an ISO8583 data format (Camel native entity) is created. It can be used like any other Camel data formats to marshal and unmarshal ISO 8583 messages in the route.

To migrate ISO 8583 from SAS Business Orchestration Services 10.1 to SAS Business Orchestration Services 10.2, you should remove the `Iso8583SerializerRepository` bean definition class. This bean should look like the following:

```
<bean id="iso8583Repository"
class="com.sas.finance.fraud.ol.xmlmapping.utils.Iso8583SerializerRepos
itory">
        <constructor-arg name="configFiles">
            <array value-type="java.lang.String">
                <value>iso8583/codec8583_87.xml</value>
                <value>iso8583/codec8583_93.xml</value>
                <value>iso8583/codec8583_03.xml</value>
            </array>
        </constructor-arg>
    </bean>
```

The ISO 8583 data format in SAS Business Orchestration Services 10.2 is auto-configured and is provided out-of-the-box for ASCII encoding. It can also be customized by changing the properties in the `application.properties` file. For more information about how to customize the ISO 8583 data format and use it in the route, see “ISO 8583 Data Format” on page 61.

# CSV Data Format

In SAS Business Orchestration Services 10.1, `csvId` and `delimiter` in the headers of the `requestDispatcher` bean were used to parse a CSV data entry. In SAS Business Orchestration Services 10.2, the Camel-CSV data format, which uses the Apache Commons CSV, should be used to parse CSV data entry.

To set up the Camel CSV data format, you can create beans. In the two following examples, two different Camel CSV data formats are created with different headers (schema) and different delimiters. If you need to customize more options, see Camel-CSV.

```
<bean id="myCustomCsv1"
class="org.apache.camel.dataformat.csv.CsvDataFormat">
    <property name="header" value="csv1Key1,csv1Key2,csv1Key3"/>
    <property name="useMaps" value="true"/>
    <property name="delimiter" value="|"/>
</bean>
<bean id="myCustomCsv2"
class="org.apache.camel.dataformat.csv.CsvDataFormat">
```

```
    <property name="header" value="csv2Key1,csv2Key2,csv2Key3"/>
    <property name="useMaps" value="true"/>
    <property name="delimiter" value=","/>
</bean>
```

The following provides information about how to use the two custom Camel CSV data formats shown above in the routes. A single CSV data entry (single line) is parsed and converted to a Map of key-value pairs. The second line of each extracts the first item out of a collection. That is because the Camel CSV data format is intended to parse batch CSV data (multiple lines) and convert them to a collection. The following is a single data entry and needs to extract the first item out of the collection.

```
<unmarshal><custom ref="myCustomCsv1"/></unmarshal>
        <transform><simple>${body.get(0)}</simple></transform>

<unmarshal><custom ref="myCustomCsv2"/></unmarshal>
        <transform><simple>${body.get(0)}</simple></transform>
```

If only one Camel CSV data format is needed, Spring Boot auto configuration can be used. For more information, see Camel-CSV.

# Migrate to the New SAS-ODE Component

In SAS Business Orchestration Services 10.1, the Camel-Netty component is called directly in the route. Encoder and decoder beans were created manually and provided as options in the Netty step. In the following example, there are two SAS OnDemand Decision Engine netty connections defined inside the load balancer in SAS Business Orchestration Services 10.1 or in a SAS Business Orchestration Services 10.2 legacy route.

```
<to uri="netty:tcp://{{ode_host_1}}:{{ode_port_1}}?
sync=true&encoders=#encoder&decoders=#decoder"/>
<to uri="netty:tcp://{{ode_host_2}}:{{ode_port_2}}?
sync=true&encoders=#encoder&decoders=#decoder"/>
```

In SAS Business Orchestration Services 10.2, the `sas-ode` Camel component is created for connections to SAS OnDemand Decision Engine. It is auto-configured with the Spring Boot properties. Modify the following property names so that they can be picked up by the `sas-ode` component.

```
# rename ode_host_1
sas.integration.component.ode.instances.ode1.hostname=ode1.mycompany.co
m
# rename ode_port_2; if default (5018) is used, this property can be
omitted.
sas.integration.component.ode.instances.ode1.port=5018

# rename ode_host_2
sas.integration.component.ode.instances.ode2.hostname=ode2.mycompany.co
m
```

```
# rename ode_port_2; if default (5018) is used, this property can be
omitted.
sas.integration.component.ode.instances.ode2.port=5018
```

Encoder and decoder beans are automatically created so that the old SAS Business Orchestration Services 10.1 or SAS Business Orchestration Services 10.2 legacy beans can be safely deleted.

The following example makes a call to the SAS OnDemand Decision Engine that is inside the load balancer in the route:

```
<to uri="sas-ode://ode1"/>
<to uri="sas-ode://ode2"/>
```

For more options and customizations, see Chapter 11, “Integrating with SAS OnDemand Decision Engine,” on page 87.

# Migrate to the New Store and Forward Mechanism

## Overview of SAS Business Orchestration Services 10.1 Configuration

Configuration of store and forward in SAS Business Orchestration Services 10.1 was done by configuring both store and forward routes and beans in the same XML context file in the **${BOSS_CONFIG}/spring** folder. The following is a sample file for a Redis store and forward in a SAS Business Orchestration Services 10.1 configuration:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd
   http://camel.apache.org/schema/spring http://camel.apache.org/
schema/spring/camel-spring.xsd
">

    <!-- Section 1: SAF routes configuration -->
    <routeContext id="safRoutes" xmlns="http://camel.apache.org/schema/
spring">
        <route id="safForwardRoute" startupOrder="37">
            <from uri="direct:safForward"/>
            <setExchangePattern pattern="InOut"/>
            <to uri="disruptor:sendTransactionToODE?
timeout=0&waitForTaskToComplete=Always"/>
        </route>
```

```
        <route id="safStoreRoute" startupOrder="22">
            <from uri="direct:safStore" />
            <bean ref="ODEStatus" method="setLastSendErrorTime" />
        </route>

        <route id="SafForwardTimer" startupOrder="54">
            <!-- change to fix delay. next task is scheduled with a
delay after current task is complete; single thread is doing it -->
            <from uri="scheduler://SAF?
useFixedDelay=true&delay={{saf_interval_milli}}&initialDelay={{saf_init
ial_delay_milli}}"/>
            <choice>
                <when>
                    <simple>${bean:ODEStatus?method=isActive} ==
'YES'</simple>
                    <bean id="SAF_Forward" ref="saf" method="forward"/>
                </when>
                <otherwise>
                    <transform><constant>ODEs not available, SAF
forward postponed.</constant></transform>
                    <to uri="log:?level=INFO"/>
                </otherwise>
            </choice>
        </route>
    </routeContext>

    <!-- Section 2: SAF beans configuration -->

    <bean id="messageConverter"
class="com.sas.finance.fraud.ol.saf.MessageConverterFactory"
          factory-method="createNewMessageConverter">
        <constructor-arg name="clazz"
value="com.sas.finance.fraud.transaction.Transaction"/>
    </bean>

        <bean id="safJedisConnectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnectionF
actory"
                p:host-name="${saf.redis.host}" p:port="$
{saf.redis.port}" p:use-pool="true" />
        <bean id="safRedisTemplate"
class="org.springframework.data.redis.core.RedisTemplate">
                <property name="connectionFactory"
ref="safJedisConnectionFactory"/>
                <property name="keySerializer">
                        <bean
class="org.springframework.data.redis.serializer.StringRedisSerializer"
/>
                </property>
                <property name="valueSerializer">
                        <bean
class="com.sas.finance.fraud.ol.standin.TransactionRedisSerializer"/>
                </property>
        </bean>
```

```
        <bean id="saf"
class="com.sas.finance.fraud.ol.saf.redis.RedisSiMqSaf" init-
method="init">
                <constructor-arg name="waitSecondsAfterLastTimeout"
value="${saf.redis.wait.afterlasttimeout.sec}"/>
                <constructor-arg name="waitSecondsAfterLastError"
value="${saf.redis.wait.afterlasterror.sec}"/>
                <constructor-arg name="maxTxnsPerSecond" value="$
{saf.redis.max.txnpersec}"/>
                <constructor-arg name="queueName" value="$
{saf.redis.queue.name}"/>
                <constructor-arg name="safName" value="$
{saf.redis.consumer.name}"/>
                <constructor-arg name="redisTemplate"
ref="safRedisTemplate"/>
                <property name="forwardTemplate"
ref="producerTemplateForSaf"/>
                <property name="sericeStatus" ref="ODEStatus"/>
                <property name="preForwardProcessor"
ref="uniqueKeySaver"/>
        </bean>

</beans>
```

As shown above, there are two sections. Section 1 defines the store and forward route configuration. Section 2 defines the store and forward beans configuration.

# Migration of the ActiveMQ and In-Memory Configuration

## Overview

To migrate the store and forward configuration from SAS Business Orchestration Services 10.1 to SAS Business Orchestration Services 10.2, you must remove any SAF beans and any JMS factory and connection XML configuration files. The SAS Business Orchestration Services 10.2 store and forward configuration can be fully configured by setting a few store and forward properties in the `application.properties` file. You need to configure only the routes and REST end points (if any) in the corresponding Camel XML files. You can still configure the JMS factories and connections using XML if you need to customize or add additional custom properties to a JMS connection factory. SAS Business Orchestration Services 10.2 also enables you to use an auto-configured JMS connection. For more information about the store and forward configuration, see Chapter 7, “Store and Forward,” on page 65.

## Migrating the ActiveMQ and In-Memory Configuration

To migrate the ActiveMQ and in-memory configuration, complete these steps:

1. You must follow the steps listed in the SAS Business Orchestration Services 10.1 to SAS Business Orchestration Services 10.2 migration steps section of this document.
2. Configure store and forward using the information in Chapter 7, “Store and Forward,” on page 65.

# Migration of an IBM MQ, Kafka, RabbitMQ, Oracle, and Postgres Configuration

To migrate a store and forward implementation that uses IBM MQ, Kafka, RabbitMQ, Oracle, or Postgres, there are two options: legacy mode and native mode.

## Legacy Mode

1. You must follow Enabling Store in the *SAS Business Orchestration Services 10.1: User’s Guide* to configure store and forward for a specific store.
2. Start SAS Business Orchestration Services in compatibility mode by following the steps in the Appendix 3, “SAS Business Orchestration Services 10.2 Migration Guide for Running in Legacy Mode,” on page 109.

## Native Mode

Migrate the store and forward routes and bean configurations to the new configurations and start SAS Business Orchestration Services in native mode.

The following sections describe the different steps that are needed to migrate of each of these data stores.

**IBM MQ**

Complete the following steps to migrate a SAS Business Orchestration Services 10.1 IBM MQ store and forward configuration to a SAS Business Orchestration Services 10.2 configuration:

1. You must follow the steps listed in this migration appendix..

2 If a bean defines an IBM MQ connection factory, then the bean can be configured using the IBM MQ auto-configuration properties in the `application.properties` file. For more information, see IBM MQ JMS Spring Components.

**Kafka**

Complete the following steps to migrate SAS Business Orchestration Services 10.1 Kafka store and forward configuration to SAS Business Orchestration Services 10.2 configuration:

1 You must follow the steps listed in this migration appendix.

2 If a bean defines a Kafka connection, then the bean can be configured to use the Kafka auto-configuration properties in the `application.properties` file. For more information, see Kafka auto-configuration.

**RabbitMQ**

Complete the following steps to migrate SAS Business Orchestration Services 10.1 RabbitMQ store and forward configuration to SAS Business Orchestration Services 10.2 configuration:

1 You must follow the steps listed in this migration appendix.

2 If a bean defines a RabbitMQ connection, the bean can be configured to use the RabbitMQ auto-configuration properties in the `application.properties` file. For more information, see RabbitMQ auto-configuration.

**Oracle**

Complete the following steps to migrate SAS Business Orchestration Services 10.1 Oracle store and forward configuration to SAS Business Orchestration Services 10.2 configuration:

1 You must follow the steps listed in this migration appendix.

2 If a bean defines an Oracle database connection, then the bean can be configured to use the DataSource auto-configuration properties in the `application.properties` file. For more information, see DataSource auto-configuration.

**Postgres**

Complete the following steps to migrate SAS Business Orchestration Services 10.1 Postgres store and forward configuration to SAS Business Orchestration Services 10.2 configuration:

1 You must follow the steps listed in this migration appendix.

2 If a bean defines a Postgres database connection, the bean can be configured to use the DataSource auto-configuration properties in `application.properties` file. For more information, see DataSource auto-configuration.

# Final Steps

As the last step, you should remove any bean, route, and REST endpoint definitions from the XML files that are no longer referenced in the other components of the configuration.