# webMethods SWIFT Module Installation and User's Guide

Version 7.1

June 2011

This document applies to webMethods SWIFT Module Installation and User’s Guide Version  7.1 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID:  ESTD-SWIFT-IUG-71SP1-20111111

# Table of Contents

# About this Guide

This guide describes how to install, configure, and use webMethods SWIFT Module.

To use this guide effectively, you should:

- Have a basic knowledge of SWIFT and SWIFT terminology. For more information, go to http://www.swift.com.
- Have installed all necessary SWIFT software. You must work with SWIFT to determine the appropriate software needs for your company.
- Have installed webMethods Integration Server, webMethods Developer, My webMethods Server, webMethods Trading Networks, Software AG Designer, and webMethods Monitor. For more information about installing these components, see *Installing webMethods Products*.
- Have installed the webMethods WebSphere MQ Adapter. For more information, see *webMethods WebSphere MQ Adapter Installation and User's Guide*.
- Be familiar with webMethods Integration Server, Integration Server Administrator, and webMethods Developer and understand the concepts and procedures described in *Developing Integration Solutions: webMethods Developer User's Guide*.
- Be familiar with webMethods Trading Networks and understand the concepts and procedures described in the various webMethods Trading Networks guides.
- Be familiar with using Software AG Designer for creating processes and tasks and understand the concepts and procedures described in the *Software AG Designer Online Help*.
- Be familiar with webMethods Monitor and understand the concepts and procedures described in *Monitoring BPM, Services, and Documents with BAM: webMethods Monitor User's Guide*.

## Document Titles

Some webMethods document titles have changed during product releases. The following table will help you locate the correct document for a release on the Software AG Documentation Web site or the Empower Product Support Web site.

| Documentation | Title |
|---|---|
| Designer Process Development online help | ■ For Designer 8.2 and later, use *webMethods BPM Process Development Help*.<br>■ For Designer 8.0 and 8.1, use *webMethods Designer BPM Process Development Help*.<br>■ For Designer 7.1.1 and 7.1.2, use *webMethods Designer Process Development Help*. |
| Designer Service Development online help | ■ For Designer 8.2 and later, use *webMethods Service Development Help*.<br>■ For Designer 7.2, 8.0, 8.0 SP1, and 8.1, use *webMethods Designer Service Development Help*. |
| Developer user's guide | ■ For Developer 8.0 SP1 and 8.2, use *Developing Integration Solutions: webMethods Developer User's Guide*.<br>■ For Developer 8.0 and earlier, use *webMethods Developer User's Guide*. |
| Integration Server administration guide | ■ For Integration Server 8.0 SP1 and later, use *Administering webMethods Integration Server*.<br>■ For Integration Server 8.0 and earlier, use *webMethods Integration Server Administrator's Guide*. |
| Integration Server built-in services reference guide | *webMethods Integration Server Built-In Services Reference* |
| Integration Server clustering guide | *webMethods Integration Server Clustering Guide* |
| Integration Server publish-subscribe developer's guide | *Publish-Subscribe Developer's Guide* |
| My webMethods administration guide | ■ For My webMethods Server 8.0.1 and later, use *Administering My webMethods Server*.<br>■ For My webMethods Server 8.0 and earlier, use *My webMethods Server Administrator's Guide*. |
| Optimize administration guide | ■ For Optimize for Infrastructure 8.0 SP1 and later, use *Administering webMethods Optimize*.<br>■ For Optimize for Infrastructure 8.0 and earlier, use *webMethods Optimize Administrator's Guide*. |

| Documentation | Title |
|---|---|
| Optimize user's guide | ■ For Optimize for Infrastructure 8.0 SP1 and later, use *Optimizing BPM and System Resources with BAM: webMethods Optimize User's Guide*.<br>■ For Optimize for Infrastructure 8.0 and earlier, use *webMethods Optimize User's Guide*. |
| Trading Networks administration guide | ■ For Trading Networks 8.0 and later, use *Building B2B Integrations: webMethods Trading Networks Administrator's Guide*.<br>■ For Trading Networks 7.1.2, use *webMethods Trading Networks Administrator's Guide*. |
| Trading Networks built-in services reference guide | ■ For Trading Networks 8.0 and later, use *webMethods Trading Networks Built-In Services Reference*.<br>■ For Trading Networks 7.1.2, use *webMethods Trading Networks Built-In Services Reference*. |
| Trading Networks concepts guide | ■ For Trading Networks 8.0 and later, use *Understanding webMethods B2B: webMethods Trading Networks Concepts Guide*.<br>■ For Trading Networks 7.1.2, use *webMethods Trading Networks Concepts Guide*. |
| Trading Networks user's guide | ■ For Trading Networks 8.0 and later, use *Managing B2B Integrations: webMethods Trading Networks User's Guide*.<br>■ For Trading Networks 7.1.2, use *webMethods Trading Networks User's Guide*. |
| webMethods installation guide | ■ For webMethods product suite 8.2 and later, use *Installing webMethods Products* and *Using the Software AG Installer*.<br>■ For webMethods product suite 8.0 SP1 and 8.1, use *Software AG Installation Guide*.<br>■ For webMethods product suite 8.0 and earlier, use *webMethods Installation Guide*. |
| webMethods logging guide | ■ For Integration Server 8.0 SP1 and later, use *webMethods Audit Logging Guide*.<br>■ For Integration Server 8.0 and earlier, use *webMethods Logging Guide*. |

| Documentation | Title |
|---|---|
| webMethods upgrade guide | ■ For webMethods product suite 8.2 and later, use *Upgrading webMethods Products*.<br>■ For webMethods product suite 8.1 and earlier, use *webMethods Upgrade Guide*. |

## Document Conventions

| Convention | Description |
|---|---|
| **Bold** | Identifies elements on a screen. |
| Narrowfont | Identifies storage locations for services on webMethods Integration Server, using the convention *folder.subfolder:service*. |
| UPPERCASE | Identifies keyboard keys. Keys you must press simultaneously are joined with a plus sign (+). |
| *Italic* | Identifies variables for which you must supply values specific to your own situation or environment. Identifies new terms the first time they occur in the text. |
| `Monospace font` | Identifies text you must type or messages displayed by the system. |
| { } | Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols. |
| \| | Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the \| symbol. |
| [ ] | Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols. |
| ... | Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...). |

## Documentation Installation

You can download the product documentation using the Software AG Installer. Depending on the release of the webMethods product suite, the location of the downloaded documentation will be as shown in the table below.

| For webMethods... | The documentation is downloaded to... |
|---|---|
| 6.x | The installation directory of each product. |
| 7.x | A central directory named _documentation in the main installation directory (webMethods by default). |

| For webMethods... | The documentation is downloaded to... |
|---|---|
| 8.x | A central directory named _documentation in the main installation directory (Software AG by default). |

## Online Information

You can find additional information about Software AG products at the locations listed below.

**Note:** The Empower Product Support Web site and the Software AG Documentation Web site replace Software AG ServLine24 and webMethods Advantage.

| If you want to... | Go to... |
|---|---|
| Access the latest version of product documentation. | **Software AG Documentation Web site**<br>http://documentation.softwareag.com |
| Find information about product releases and tools that you can use to resolve problems.<br>See the Knowledge Center to:<br>■ Read technical articles and papers.<br>■ Download fixes and service packs.<br>■ Learn about critical alerts.<br>See the Products area to:<br>■ Download products.<br>■ Download certified samples.<br>■ Get information about product availability.<br>■ Access older versions of product documentation.<br>■ Submit feature/enhancement requests. | **Empower Product Support Web site**<br>https://empower.softwareag.com |

| If you want to... | Go to... |
|---|---|
| ■ Access additional articles, demos, and tutorials.<br>■ Obtain technical information, useful resources, and online discussion forums, moderated by Software AG professionals, to help you do more with Software AG technology.<br>■ Use the online discussion forums to exchange best practices and chat with other experts.<br>■ Expand your knowledge about product documentation, code samples, articles, online seminars, and tutorials.<br>■ Link to external Web sites that discuss open standards and many Web technology topics.<br>■ See how other customers are streamlining their operations with technology from Software AG. | Software AG Developer Community for webMethods<br>http://communities.softwareag.com/ |

# I Getting Started

# 1 Concepts

# What Is the SWIFT Network?

SWIFT (Society for Worldwide Interbank Financial Telecommunication) and its networks provide a secure, global financial IP-based messaging platform that enables financial institutions to exchange formatted financial information and transactional data. The SWIFT networks enable you to exchange SWIFT FIN messages using the original SWIFT Transport Network (STN) or the new SWIFT Secure IP Network (SIPN).

## What Is SWIFTNet?

SWIFTNet is SWIFT's advanced IP-based messaging solution, which provides an alternate method for transferring information to SWIFT. It consists of a portfolio of products and services enabling the secure and reliable communication of financial information and transactional data.

## What Is SWIFTNet Link?

SWIFTNet Link (SNL) is an application programming interface that offers access to all SWIFTNet services. Business applications can use SNL with SWIFT FIN interface products such as SWIFT Alliance Access to connect to and use the SWIFTNet FIN services. Applications can use SNL directly, or with an interface product such as the SWIFT Alliance Gateway (SAG), to enable application-to-application communication over the SWIFTNet services.

SNL functionality includes messaging, security, and service management. The security and service management functions are beyond the scope of this guide and are not discussed here.

### SNL Messaging Services

The messaging services that SNL supports are: SWIFTNet InterAct, SWIFTNet FileAct, SWIFTNet FIN, and SWIFTNet Browse.

#### SWIFTNet InterAct

SWIFTNet InterAct allows the exchange of messages between parties in synchronous mode, using the Exchange Request function, or asynchronous mode, using the Send/Wait Request function.

- In synchronous mode, the Exchange Request function sends data to the Responder application. This function blocks the Requestor application until the response is returned from the Responder and delivered to the Requestor.

  The Responder SAG sends a Handle Request primitive to the Responder application, which processes it and returns a Handle Response primitive. The Responder SAG then sends the Exchange Response primitive back to the Requestor application.

- In asynchronous mode, the Send/Wait Requests function sends data. The Requestor application initiates a send request to SNL. SNL accepts or rejects the request immediately, and simultaneously unblocks the Requestor application so that it can perform other tasks. The message is forwarded to the destination SNL and delivered to the Responder application. The subsequent response is returned to the Requestor SNL where the Requestor collects it through a Wait Request.
- The requests and responses for both Exchange and Send/Wait requests are coded in XML and passed between the communicating SNL instances over SIPN.

### SWIFTNet FileAct

SWIFTNet FileAct allows the automated exchange of files, supporting both synchronous and asynchronous modes. SWIFTNet FileAct is oriented toward transferring data larger than the SWIFTNet InterAct payload can accommodate.

- The Exchange File Request function transfers files to server applications.
- The Handle File Request function requests that the server application receive the file transfer and send a response.
- Both requests have an optional delivery notification primitive that acknowledges that a file has been received and transferred to a reliable storage environment.

### SWIFTNet FIN

SWIFTNet FIN allows the use of the standard SNL APIs with the SWIFTNet FIN interface to do the following:

- Sign SWIFTNet FIN messages using SWIFTNet PKI security profiles.
- Send SWIFTNet InterAct requests that contain input messages and user acknowledgements of previously received output messages.
- Handle SWIFTNet InterAct requests that contain output messages and SWIFTNet FIN acknowledgements of previously sent input messages.

### SWIFTNet Browse

SWIFTNet Browse enables secure communication between standard browsers and web servers. SWIFTNet Browse supports the messaging functions of SWIFTNet InterAct and SWIFTNet FileAct, and request/response interactions with a web server. The message flow path in SWIFTNet Browse is identical for both SWIFTNet InterAct and SWIFTNet FileAct messages/files.

For more information, see the documentation provided by SWIFT or go to http://www.swift.com.

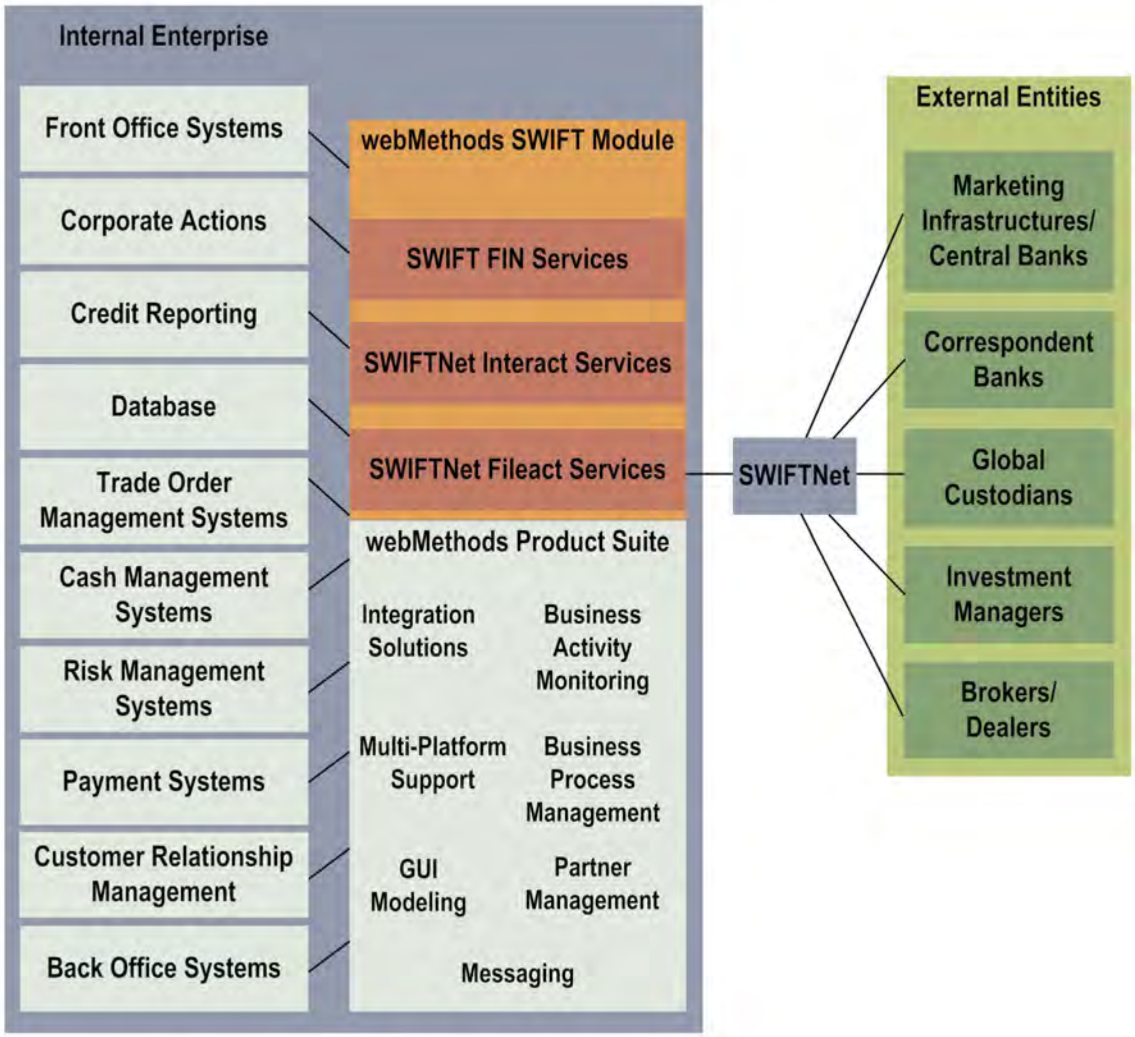
## What Is webMethods SWIFT Module?

webMethods SWIFT Module provides message exchange over the SWIFTNet messaging services and enables you to do the following:

- Easily map data from any source format to any target format.
- Manage data dictionary standards and validation.
- Monitor and control message flow.
- Provide exception handling.
- Archive inbound and outbound messages.

SWIFT Module consists of two components to facilitate parsing, validation, and transport of messages: SWIFT FIN and SWIFTNet.

- SWIFT FIN interacts with the SWIFT Network through the SWIFT Alliance Access interface (SAA).
- SWIFTNet interacts with the SWIFT Network through the SWIFT Alliance Gateway (SAG) interface.



## webMethods SWIFT Module Packages

webMethods SWIFT Module uses services and other elements included in packages that you install on Integration Server. Some of those packages contain services that are common to both the SWIFT FIN component and the SWIFTNet component, while other packages contain services that are specific to each component. The following table describes the contents of each package and the functionality that it supports.

| Package | Description |
|---|---|
| WmEstdCommonLib | Contains generic services that enable you to use various webMethods eStandards Modules with Integration Server.<br>For a list of services that SWIFT Module uses from this package, see "WmEstdCommonLib Package" on page 274. For detailed information about those services, see *webMethods eStandards Modules Common Built-In Services Reference*. |
| WmFIN | Contains services used to implement and support the SWIFT FIN-compliant functionality of SWIFT Module. |
| WmSWIFTClient | Contains the elements (flow services, Java services, record descriptions, and wrapper services) that support SWIFTNet component client-side functionality. |
| WmSWIFTCommon | Contains common services that are used by other SWIFT Module packages. |
| WmSWIFTServer | Contains the elements (flow services, Java services, record descriptions, and wrapper services) that support SWIFTNet component server-side functionality. |
| WmSWIFTSamples | Contains sample services that show how to use different features of SWIFT Module. You can also use the sample services as examples how to create your own services.<br>**Important!** The SWIFT Module samples only demonstrate the features of the module and must not be used in production environment. You must delete the WmSWIFTSamples package before you go into production. |

For detailed information about the contents of each package, see Chapter 11, "Configuring SWIFT Interfaces" and Appendix A, "Services". For information about the samples see *webMethodsSWIFT Module Samples Guide*.

# SWIFT FIN Component

The SWIFT FIN component enables Integration Server to do the following:

- Receive inbound SWIFT FIN messages from the SWIFT network.

- Convert SWIFT FIN messages into your back-end format and process the messages according to your settings.
- Send SWIFT FIN messages to the SWIFT network with correct header information, according to your settings.

The SWIFT FIN component interfaces with SWIFT Alliance Access (SAA) software via MQHA, CASmf, or AFT. SAA, in turn, communicates with SNL, which sends and receives messages securely via SWIFT Secure IP Network. SAA and SNL software modules are provided by SWIFT and must be installed and configured at a customer site by a SWIFT professional or by a trained expert.

The SWIFT FIN component provides the ability to seamlessly integrate SWIFT FIN messages as webMethods documents into a solutions architecture and validate those messages at the syntax and network level. Messages sent and received by SWIFT Module are validated at the individual field level and across the fields using network validation rules. The SWIFT FIN component also supports Market Practices among partners located in a particular market.

## What Is a SWIFT FIN Message?

SWIFT FIN messages transmit financial information from one financial institution to another. These messages are classified into different message categories. There are 10 categories of FIN messages (Category 0 through Category 9) and each category relates to a particular topic. For example, Category 5 contains messages related to Securities.

Each SWIFT message is represented by a three-digit number (for example, MT 541). The MT represents SWIFT's “Message Type.” The first number (5) identifies the category to which the message belongs; the second and third numbers (41) identify the message type.

SWIFT updates the MT specifications every year. SWIFT Module maintains these specification changes in the swiftMT and dfdMT XML files. Based on the specified version, SWIFT Module uses the corresponding swiftMT or dfdMT files to define the IS document that is created.

For details about SWIFT specifications, see http://www.swift.com. For details about the SWIFT specification versions that SWIFT Module supports, see *webMethods eStandards Modules System Requirements*.

### About SWIFT Message Format

All SWIFT messages must adhere to a defined format or “block structure.” There are five possible blocks within a message, each consisting of fields that provide specific information related to the block type. Blocks are distinguishable by brace delimiters that start and end each individual block. Additionally, each block begins with a block identifier (number) and a colon. The identifier indicates the block type as header, trailer, or text.

A SWIFT message may have the following five blocks:

- 1: Basic Header Block—Mandatory. This contains basic header information. A SWIFT message always has header Block 1. No field separators are used within this block.
- 2: Application Header Block—Contains header information about the message itself. The content of this block depends on whether it is a GPA or a FIN message. No field separators are used within this block.
- 3: User Header Block—Contains header information for user-to-user messages only within the FIN application. This identifies the version of the message standard.
- 4: Text Block—Contains the text of the SWIFT message. This is the “body” of the message that provides the message data. Each field within this block starts with a message tag, followed by the values for that tag, for example, 22F::MICO/A2C4E6G8/A2C4, where 22F is the message tag, and the information that follows is the value for that field. The format of this block is always the tag number, followed by a colon, and then the field values. This block begins with a carriage return and line feed and ends with a carriage return and line feed followed by a hyphen.
- 5: Trailers Block—Contains the trailer information to indicate any special handling conditions or additional information.

Sample SWIFT FIN Message, MT 541, Receive Against Payment

```
{1:F01CLSAHKHHXXXX0116013185}{2:I541CLSAHKHHXXXXN}
{3:{108:MT535 004 OF 006}}
{4:
:16R:GENL
:20C::SEME//01430
:23G:NEWM/CODU
:98C::PREP//19991231232359
:99B::SETT//123
:16R:LINK
:22F::LINK/A2C4E6G8/A2C4
:13A::LINK//513
:20C::PREV//x
:16S:LINK
:16S:GENL
:16R:TRADDET
:94B::TRAD//EXCH/30x
...
:97A::CASH//x
:97A::SAFE//x
:16S:OTHRPRTY-}
```

For more detailed information about SWIFT FIN messages, see the documentation provided by SWIFT or go to http://www.swift.com.

## What Is a SWIFT MX Message?

The SWIFT FIN component also supports the SWIFT Standards MX messages. MX messages are represented using eXtensible Markup Language (XML). With MX messages, you can transport structured information using XML and specify the structure of the message. At the highest level, an MX message is categorized by its business area, represented by four letters. For example, in camt.029.001.01, “camt” specifies the Cash Management business area. The three numbers that follow the letters identify the message functionality. The next three numbers identify the Variant ID, and the last two numbers show the version number.

An MX message contains the business area specific payload. Its structure is defined by the corresponding XML schema. The MX message is wrapped as the RequestPayload within the XML envelope. This request payload also contains the ApplicationHeader. This application header contains general information, and its usage is specific to the context of the service.

Sample SWIFT MX Message

```
<AppHdr xmlns="urn:swift:xsd:$ahV10">
<MsgRef>TRNREF001</MsgRef>
<CrDate>2009-05-08T22:02:36.218+02:00</CrDate></AppHdr>
<tns:Document xmlns:tns="urn:swift:xsd:setr.010.001.03">
<tns:SbcptOrdrV03><tns:MsgId>
<tns:Id>TRNREF001</tns:Id>
<tns:CreDtTm>2007-04-25T10:10:30.000+02:00</tns:CreDtTm></tns:MsgId>
<tns:MltplOrdrDtls><tns:InvstmtAcctDtls>
<tns:AcctId><tns:Prtry><tns:Id>1111
</tns:Id></tns:Prtry></tns:AcctId>
<tns:AcctDsgnt>SMART INVESTOR</tns:AcctDsgnt>
</tns:InvstmtAcctDtls>
<tns:IndvOrdrDtls><tns:OrdrRef>TRNREF001</tns:OrdrRef>
<tns:FinInstrmDtls><tns:Id><tns:ISIN>GB1234567890</tns:ISIN></tns:Id>
</tns:FinInstrmDtls>
<tns:GrssAmt Ccy="GBP">1050</tns:GrssAmt>
<tns:IncmPref>CASH</tns:IncmPref>
<tns:PhysDlvryInd>false</tns:PhysDlvryInd>
<tns:ReqdSttlmCcy>GBP</tns:ReqdSttlmCcy>
<tns:ReqdNAVCcy>GBP</tns:ReqdNAVCcy>
</tns:IndvOrdrDtls></tns:MltplOrdrDtls>
</tns:SbcptOrdrV03></tns:Document>
```

For information about MX messages, see *SWIFT User Handbook*.

## SWIFT FIN Component Parts

The following parts compose and support the SWIFT FIN component:

- **The WmFIN package.** This package contains services, mappings, and records for using SWIFT Module with Integration Server. For a complete list of packages, see “webMethods SWIFT Module Packages” on page 27.
- **SWIFT Interfaces.** You can connect to SWIFT using one of the following interfaces:
  - **MQHA (MQ Host Adapter).** To communicate with SWIFT using MQHA, use the webMethods WebSphere MQ Adapter. For more information about using the WebSphere MQ Adapter with SWIFT Module, see “Using WebSphere MQ Adapter to Communicate with SWIFT” on page 110.

  - CASmf (Common Application Server message format). To communicate with SWIFT using CASmf, use the CASmf services provided in the WmFIN package. For more information about CASmf services, see "Using the CASmf Services to Communicate with SWIFT" on page 112.
  - AFT (Automated File Transfer). To communicate with SWIFT using AFT, use the File Polling Listener and File Drop capabilities. For more information about using AFT with SWIFT Module, see "Using AFT to Communicate with SWIFT" on page 116.

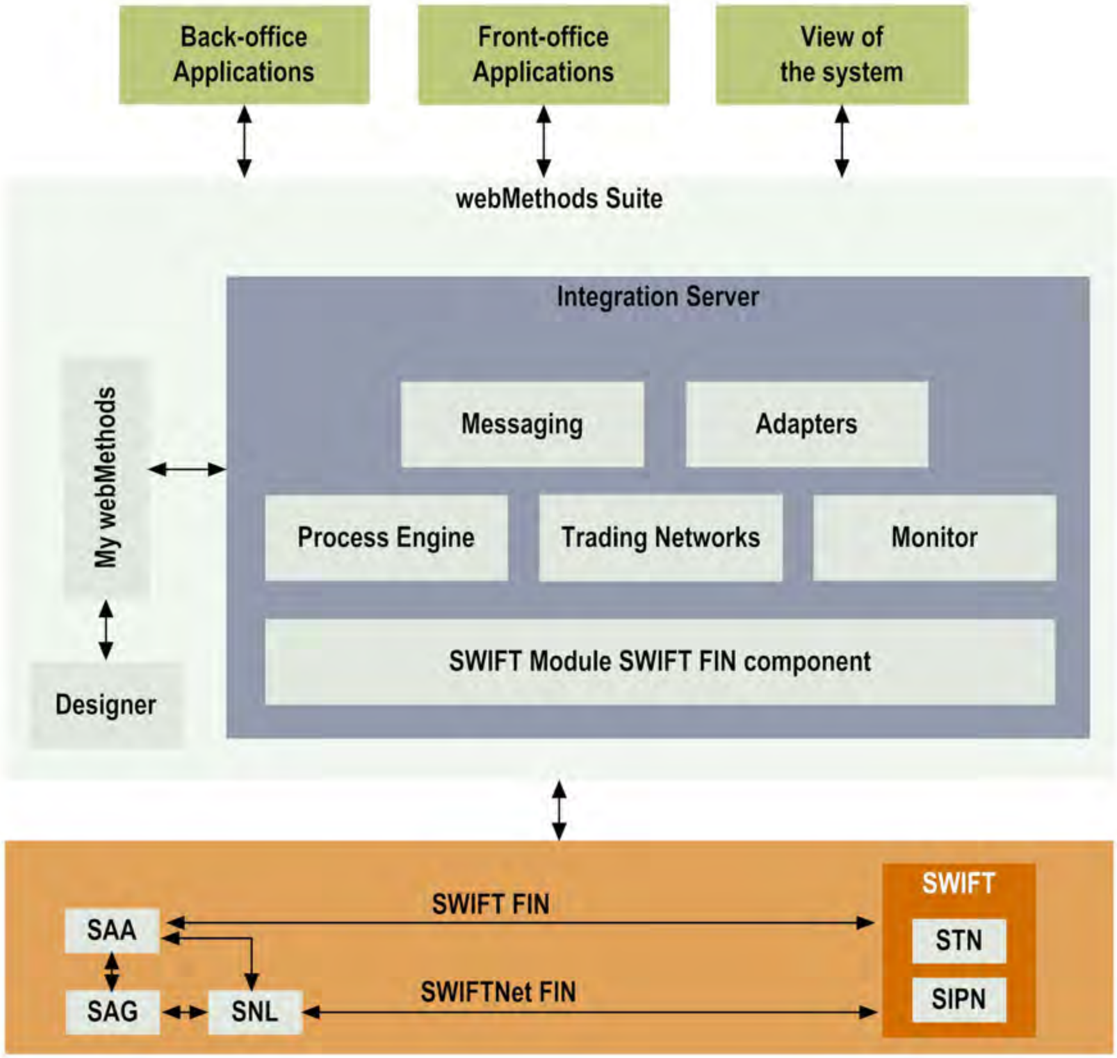
  For more information about these interfaces, see Chapter 11, "Configuring SWIFT Interfaces"

- Integration Server. This is the underlying server of the webMethods product suite. Use the web-based user interface, Integration Server Administrator, to manage, configure, and administer all aspects of Integration Server, such as users, security, packages, and services. For more information, see the Integration Server administration guide for your release. See "About this Guide" for specific document titles..
- webMethods Trading Networks. webMethods Trading Networks enables your enterprise to link with other financial institutions and marketplaces to form a business-to-business trading network. For more information about using Trading Networks, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.
- Software AG Designer. Software AG Designer is a design-time tool that you can use to create processes and easy-to-understand, visually-based process models. You can also use Designer to create, update, and execute services from the Package Navigator. For more information about Designer, see the Designer online help for your release. See "About this Guide" for specific document titles.
- webMethods Monitor. webMethods Monitor allows you to manage and monitor business processes. Access Monitor functionality through the My webMethods user interface. Monitor displays information about a business process by retrieving information from the Process Audit Log. For more information about Monitor, see *Monitoring BPM, Services, and Documents with BAM: webMethods Monitor User's Guide*.

## SWIFT FIN Component Architecture

The SWIFT FIN component uses either the publish and subscribe Process Engine functionality of SWIFT Module, or Trading Networks processing rules to send and receive SWIFT FIN messages. When used with Trading Networks, the SWIFT FIN component leverages the archiving, Trading Partner Agreement (TPA), and document type components of Trading Networks to work with your enterprise to exchange SWIFT FIN messages.

The following figure shows the SWIFT FIN component architecture.



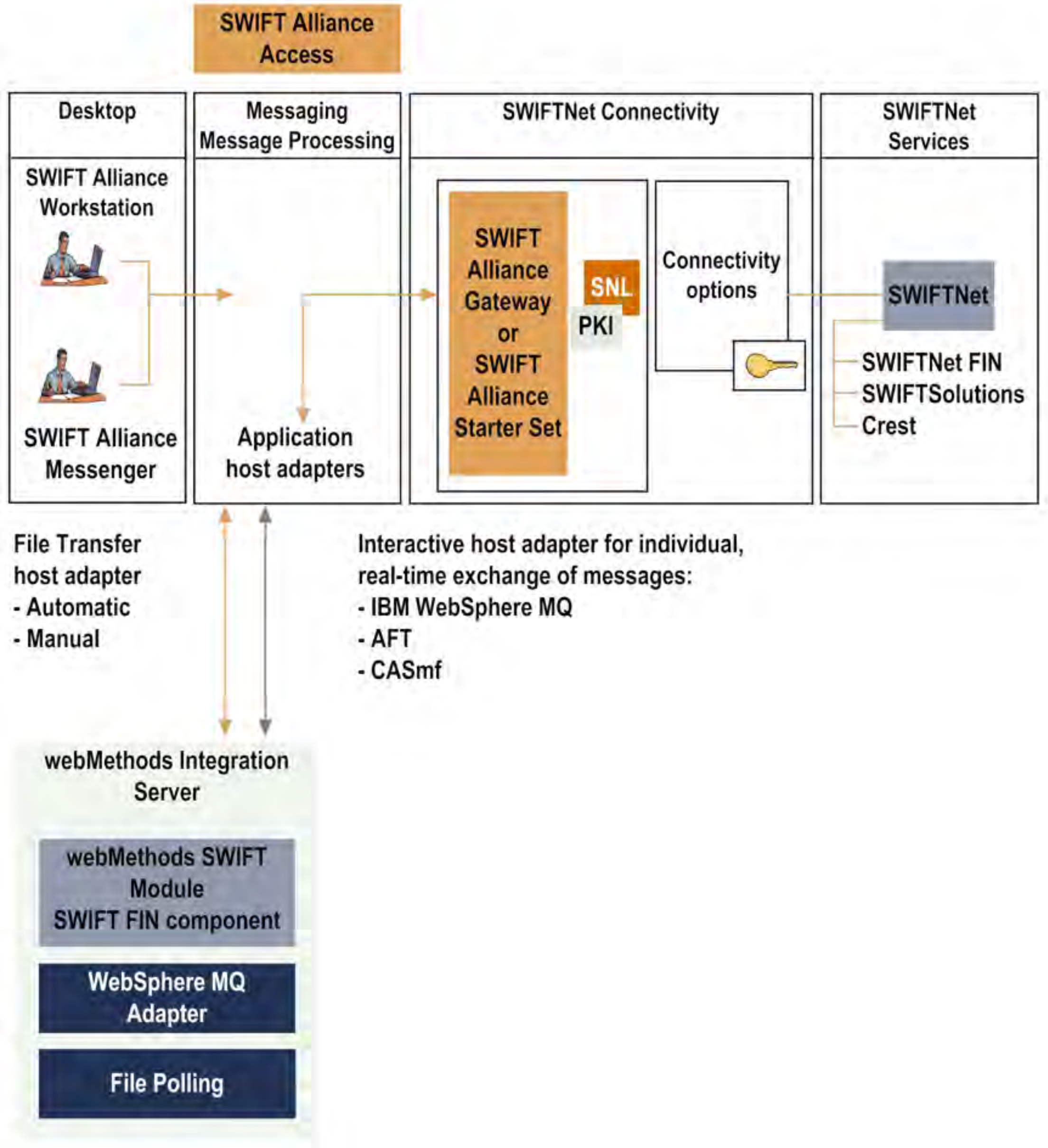
The SWIFT FIN component consists of a set of design-time and run-time component parts, both of which are discussed in this section. For information about design-time component parts, see "“SWIFT FIN Component Parts” on page 30. For information about run-time component parts, see "“SWIFT FIN Component Architecture” on page 31.

When the SWIFT FIN component creates an outbound document, it formats, validates, and publishes the SWIFT message. When the SWIFT FIN component receives an inbound document, it parses, formats, validates, and publishes the message for a back-end application.

To communicate with SWIFT using SAA, there are three options:

- Use webMethods WebSphere MQ Adapter to interface with MQHA.
- Use the CASmf services provided in the WmFIN package to interface with CASmf.
- Use the File Polling Listener and File Drop capabilities to interface with AFT.

The following diagram illustrates the end-to-end architecture of the SWIFT FIN component messaging solution.


SWIFT Alliance Access
Desktop
Messaging Message Processing
SWIFTNet Connectivity
SWIFTNet Services
SWIFT Alliance Workstation
SWIFT Alliance Messenger
Application host adapters
SWIFT Alliance Gateway or SWIFT Alliance Starter Set
SNL
PKI
Connectivity options
SWIFTNet
SWIFTNet FIN
SWIFTSolutions
Crest
File Transfer host adapter
- Automatic
- Manual
Interactive host adapter for individual, real-time exchange of messages:
- IBM WebSphere MQ
- AFT
- CASmf
webMethods Integration Server
webMethods SWIFT Module SWIFT FIN component
WebSphere MQ Adapter
File Polling

## SWIFT FIN Component Features

The SWIFT FIN component runs on top of Integration Server and provides the following functionality:

- **Current messages.** The component supports the following SWIFT messages:
  - The latest release of SWIFT FIN messages.
  - The SWIFT Standards MX messages.

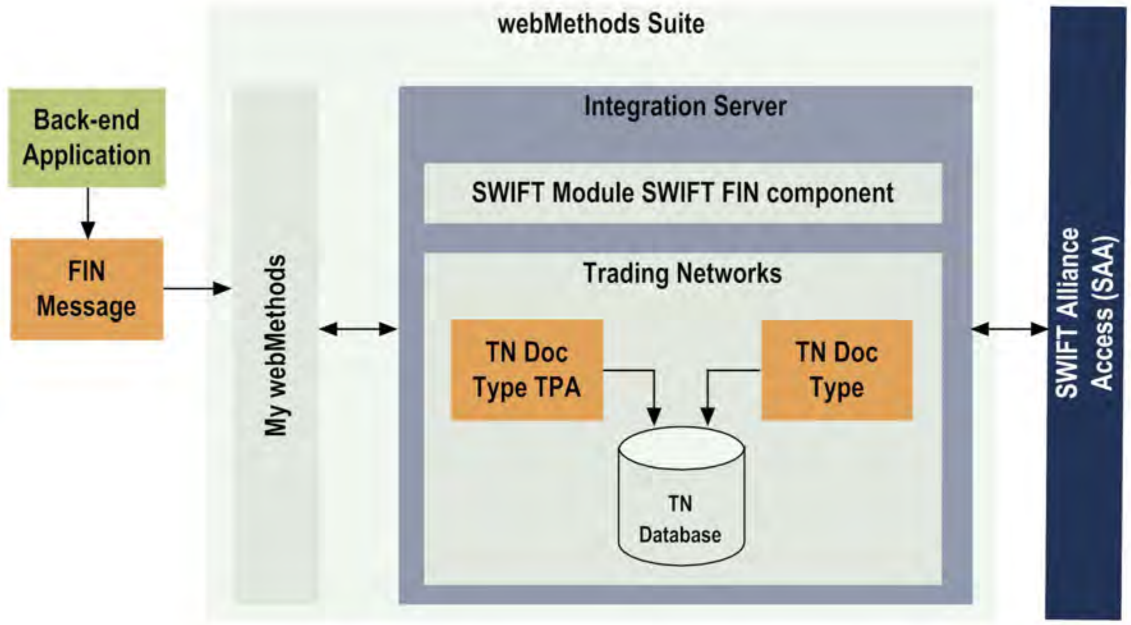
- **Data field dictionary.** The SWIFT FIN component provides a data field dictionary (DFD) based on the ISO 15022 standards for SWIFT FIN messages. This DFD enables translation of a message tag number (for example, “22F::SFRE”) into a meaningful business name (for example, “Statement Frequency Indicator”). In addition, the SWIFT FIN component enables you to choose how you want to display each message in Software AG Designer:
  - Tag number only (for example, “22F::SFRE”)
  - Equivalent message business name only (for example, “Statement Frequency Indicator”)
  - Both the tag number and the equivalent message business name (for example, “22F::SFRE_Statement Frequency Indicator”)
  - XML data tag (for example, “22FSFRE”)
- **Message archival.** All SWIFT FIN messages can be archived in Trading Networks.
- **SWIFT interfaces.** The SWIFT FIN component provides out-of-box support to interface to SWIFT using MQHA, CASmf, and AFT. For more information, see Chapter 11, “Configuring SWIFT Interfaces”.
- **BICPlusIBAN validation and searching.** The SWIFT FIN component provides support for deriving or validating data against the BICPlusIBAN directory. BICPlusIBAN is a SWIFT directory that contains identifiers recognized by financial institutions, such as Bank Identifier Codes (BICs), International Bank Account Numbers (IBANs), and national clearing codes. The SWIFT BICPlusIBAN directory serves two main purposes:
  - To provide or validate data in international payment messages, for example to translate the beneficiary bank's BIC into national (clearing, sort) code, or validate the banks' details (such as name and address).
  - To provide or validate data in SEPA (Single Euro Payment Area) payments, for example, to derive the BIC from the IBAN if the IBAN is missing, or to validate IBAN/BIC combinations.

  For more information about the directories, see Chapter 4, “Importing BICPlusIBAN and SEPA Routing Directories”.
- **Syntax and network validation.** The SWIFT FIN component enables you to validate the message structure, field formats, and network rules of inbound and outbound SWIFT FIN messages. SWIFT Module provides network validation rules for a few commonly used message types. In addition to these rules, the component enables you to create network, Market Practice, and usage validation rules for additional messages as well. For more information, see “Creating Validation Rules” on page 83 and “Working with Market Practices” on page 143.
- **Market practices.** Market Practices are specific requirements for individual markets. Using Trading Partner Agreements (TPAs), the SWIFT FIN component supports customization of SWIFT FIN messages based on specific trading partner pairs. For more information about SWIFT-related Market Practices and TPAs, see “Working with Market Practices” on page 143 and “Customizing Trading Partner Agreements”

on page 93.

- **Processing rule support.** You can use custom-created Trading Networks processing rules for each SWIFT message record. For information about creating processing rules, see the Trading Networks administration guide for your release. See “About this Guide” for specific document titles.
- **SWIFT error codes.** The component supports SWIFT error codes for field level and network validation. It also supplies resource bundles so that all error codes can be localized.
- **Integration Server clustering.** The SWIFT FIN component can be used in a clustered Integration Server environment. For more information about clustering see
- **Subfield parsing.** The SWIFT FIN component automatically parses messages into blocks and fields. You can configure further parsing into subfields with the *subfieldFlag* variable, which is included in the following services:
  - wm.fin.dev:importFINItems
  - wm.fin.dfd:convertTagFormat
  - wm.fin.dfd:convertBizNameFormat

The following figure illustrates how the SWIFT FIN component interacts with other components. For further explanation, see the table that follows the figure.



| Component | Description |
|---|---|
| SWIFT FIN component | The SWIFT FIN component uses Trading Networks processing rules to manage the execution of SWIFT FIN messages. When the component receives a message from a back-end system, it invokes a Trading Networks service to recognize the message. For information about creating processing rules, see Chapter 9, “Configuring Processing Rules to Send and Receive SWIFT FIN Messages”. |
| Trading Networks | Trading Networks uses the information defined in trading partner profiles to enable your enterprise to link to the financial institutions with whom you want to exchange SWIFT FIN messages. You can customize TPAs and view TN document types when you create your message records.<br>For more information about Trading Networks, trading partner profiles, TN document types, and TPAs, see the Trading Networks administration guide for your release. See “About this Guide” for specific document titles. You can also find information about trading partner profiles and TN document types in Chapter 5, “Defining Trading Networks Information”, and information about TPAs in Chapter 8, “Customizing Trading Partner Agreements”. |
| Trading Networks Database | The Trading Networks database stores TN document types, TPA, and trading partner profile information, among other things. |
| Integration Server | Integration Server contains the documents, services, and IS documents that you need when creating your process models. Integration Server also contains the elements (services, triggers, and process fragments) that were generated by the automated controlled steps within the process model. |

## SWIFTNet Component

The SWIFTNet component supports communication of SWIFT messages and files between clients and servers:

- The client *sends* a request and receives a response.
- The server *receives* a request and sends a response.

The SWIFTNet component provides client-side and server-side support for the following messaging services and capabilities:

- InterAct Services
- FileAct Services

Both InterAct and FileAct Services can work in either real-time mode or in store-and-forward mode. In real-time mode, both the requestor and the responder must be online at the same time, but in store-and-forward mode, they do not both need to be online.

The client uses the SNL function SwCall() to access the server application through SWIFTNet. The server uses the SNL function SwCallback() to respond to clients through SWIFTNet.

As mentioned earlier, InterAct and FileAct Services are implemented as a set of SNL primitives that are exchanged between the client or server application program and the SNL software on your SAG. Along with its packages, the SWIFTNet component provides two DLLs, WmSWIFTNetClient.dll and WmSWIFTNetServer.dll, that invoke the functionality of the SNL libraries to transfer the SNL primitives between the client and server.

## Client Functionality

The WmSWIFTNetClient libraries (that is, WmSWIFTNetClient.dll or WmSWIFTNetClient.so) invokes functionality for a client, which sends a request to and receives a response from a server in real-time or store-and-forward mode. When using SWIFT Module with a client, you can do the following:

- `Send` an InterAct request message and receive a response in real-time or store-and-forward mode.
- `Put` a file using FileAct service in real-time or store-and-forward mode.
- `Get` a file using FileAct service in real-time mode only.
- `Pull` messages from a queue in store-and-forward mode only.
- `Fetch` a file from the SnF queue in store-and-forward mode only.

## Server Functionality

The WmSWIFTNetServer libraries (that is, WmSWIFTNetServer.dll or WmSWIFTNetServer.so) invokes functionality for a server, which receives a request from and sends a response to a client. When using SWIFT Module with a server, you can do the following:

- Receive an InterAct request message and send a response in real-time or store-and-forward mode.
- Accept a `put` file request from the client application in real-time mode only.
- Accept a `get` file request from the client application in real-time mode only.
- Receive the pushed messages from the SnF queue in store-and-forward mode only.

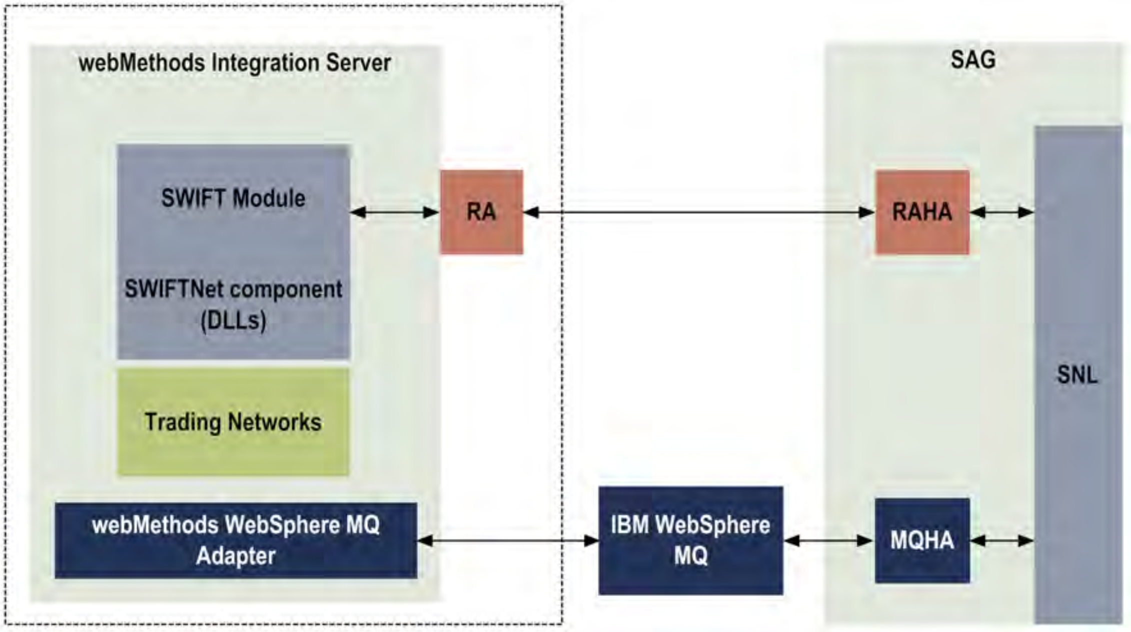
For more information about the architecture of the module, see "SWIFTNet Component Architecture" on page 38.

## SNL Request and Response Primitives Support

The SWIFTNet component supports all of the SNL request and response primitives involved in communication between the client, the server, and SWIFTNet. For a complete list of the supported primitives, see "Services and the SNL Request and Response Primitives" on page 293.

## SWIFTNet Component Architecture

The following diagram illustrates the architecture of systems and processes that enable the SWIFTNet component to exchange messages and files. See the table below the diagram for additional information.


webMethods Integration Server
SWIFT Module
SWIFTNet component (DLLs)
Trading Networks
webMethods WebSphere MQ Adapter
RA
IBM WebSphere MQ
SAG
RAHA
MQHA
SNL

The following table describes the elements of the SWIFTNet component architecture:

| Component | Description |
|---|---|
| IBM Web SphereMQ | IBM WebSphere MQ uses the store-and-forward model to enable programs to communicate by passing messages to one another via a message queue. This enables asynchronous data exchange. |
| Integration Server | Integration Server hosts the SWIFTNet component, Trading Networks, and WebSphere MQ Adapter services and related files. Use Integration Server Administrator to manage, configure, and administer all aspects of Integration Server, such as users, security, packages, and services.<br>For more information about Integration Server, see the Integration Server administration guide for your release. See "About this Guide" for specific document titles. |

| Component | Description |
|---|---|
| MQHA | The MQ Host Adapter (MQHA) enables your SWIFTNet component client and server applications to communicate with SWIFT Alliance Gateway through IBM WebSphere MQ. MQHA is the default transport.<br>To obtain MQHA, contact SWIFT. |
| webMethods Trading Networks | Trading Networks enables your enterprise to link with other financial institutions and marketplaces to form a business-to-business trading network. Trading Networks also enables the SWIFTNet component to exchange messages and files with your SWIFT Alliance Gateway.<br>For more information about Trading Networks, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles. |
| RA | The Remote API (RA) client enables the SWIFTNet component to communicate with your SWIFT Alliance Gateway and SNL through your Remote API Host Adapter (RAHA). You must install an RA client on the same machine as Integration Server.<br>To obtain an RA client, contact SWIFT. |
| RAHA | Your RAHA enables your SWIFT Alliance Gateway (SAG) to exchange messages and files with the RA client on your Integration Server. You must install RAHA on the same machine as SAG. RAHA supports single-threaded processing of messages.<br>To obtain RAHA, contact SWIFT. |
| SAG | The SWIFT Alliance Gateway (SAG) on which you install your SNL software must be configured to exchange messages and files with SWIFTNet. You also will use this configuration information to configure SWIFT Module and your RA client. |
| webMethods WebSphere MQ Adapter | The WebSphere MQ Adapter enables Integration Server to exchange information with SWIFT Alliance Gateway through an IBM WebSphere MQ message queue. This capability lets you route documents or any piece of information from Integration Server to systems that use WebSphere MQ message queuing as their information interface. |

## SWIFTNet Component Real-Time Mode

Real-time InterAct message services are typically used when the receiver and the sender are both online at the time of message or file transmission. When real-time mode is used, the responder's server application generates the response and interprets the message sent.

## Real-Time InterAct

InterAct Services ensure secure communication of request/response business messages between application-level clients and servers on SWIFTNet. It is cost-effective and ideal for online queries or reporting systems.

The sequence of the InterAct request/response session is as follows:

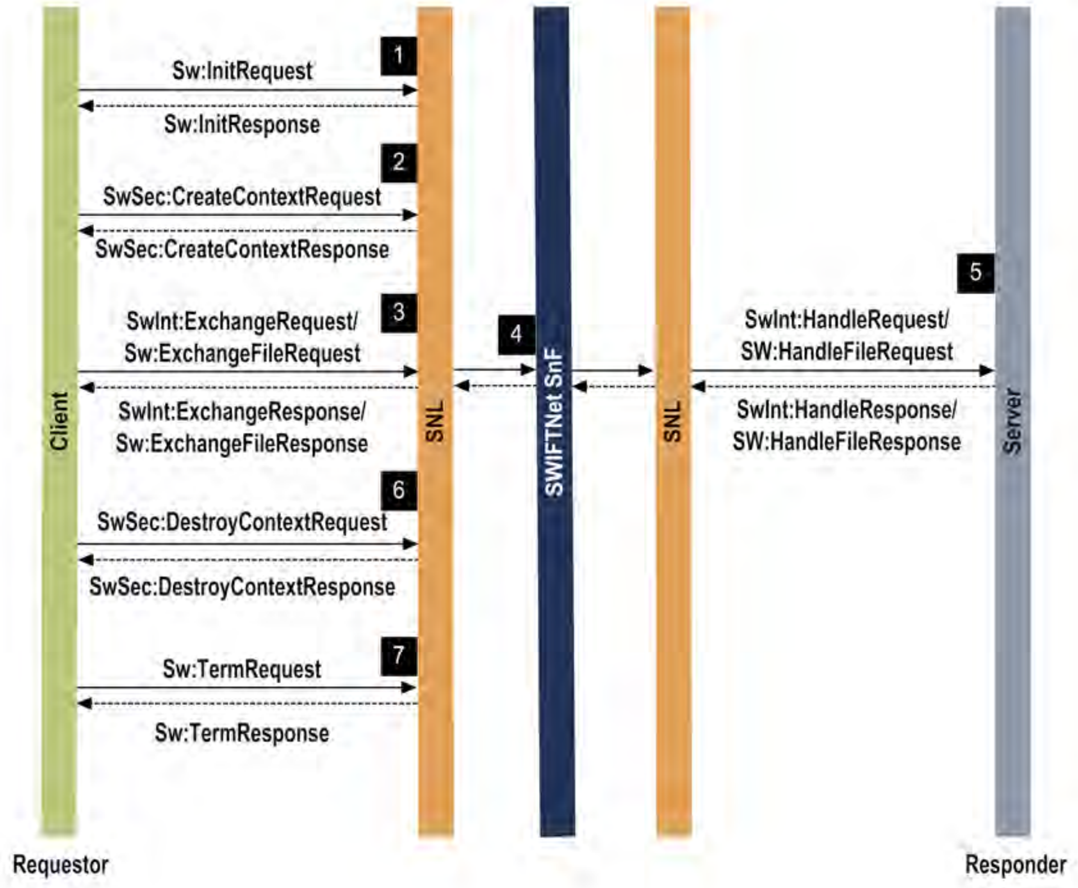
1. The requestor's client sends a request.
2. The client request is passed to SWIFTNet network, which processes the request and sends it to the responder's server.
3. The responder's server receives the request and sends the response.
4. SWIFTNet processes the response received from the responder's server and sends it to the requestor's client application.
5. The requestor's client receives the response.

## Real-Time FileAct

Real-time FileAct Services offer a secure transfer of financial files between organizations on SWIFTNet. XML based FileAct primitives are used to transfer the files and maintain the status of the file transfers. FileAct Services provide the following functionality:

- **Put File.** Send a file to another SWIFTNet user.
- **Get File.** Receive a file from another SWIFTNet user.
- **Subscribe to Transfer Events.** Receive progressive transfer status on an event-by-event basis.
- **Receive Transfer Events.** Respond to the terms of a subscription that is set up by the Subscribe Event primitive at the sending or receiving side of a transfer.

The following diagram illustrates the real-time InterAct/FileAct service. See the table below the diagram for additional information.



| Step | Description |
|---|---|
| 1 | The requestor's client sends the `Sw:InitRequest` primitive to initialize the SNL client process. |
| 2 | The requestor's client makes a SwCall() with `SwSec:CreateContextRequest` as primitive to initialize the security context. |
| 3 | The client makes a request using the appropriate primitive for the service type:<br>■ For an InterAct service, the client uses `SwInt:ExchangeRequest`.<br>■ For a FileAct service, the client uses `Sw:ExchangeFileRequest`. |
| 4 | The requestor's client side SNL passes the InterAct or FileAct request to the responder's server side SNL via SWIFTNet. |
| 5 | The responder's server side SNL extracts the request from SWIFTNet and invokes the server through SwCallback()`SwInt:HandleRequest/ Sw:HandleFileRequest`. The responder's server returns a response to the client. |
| 6 | The client destroys the created security context. |

| Step | Description |
|---|---|
| 7 | The client triggers the termination with the SNL. |

## SWIFTNet Component Store-and-Forward Mode

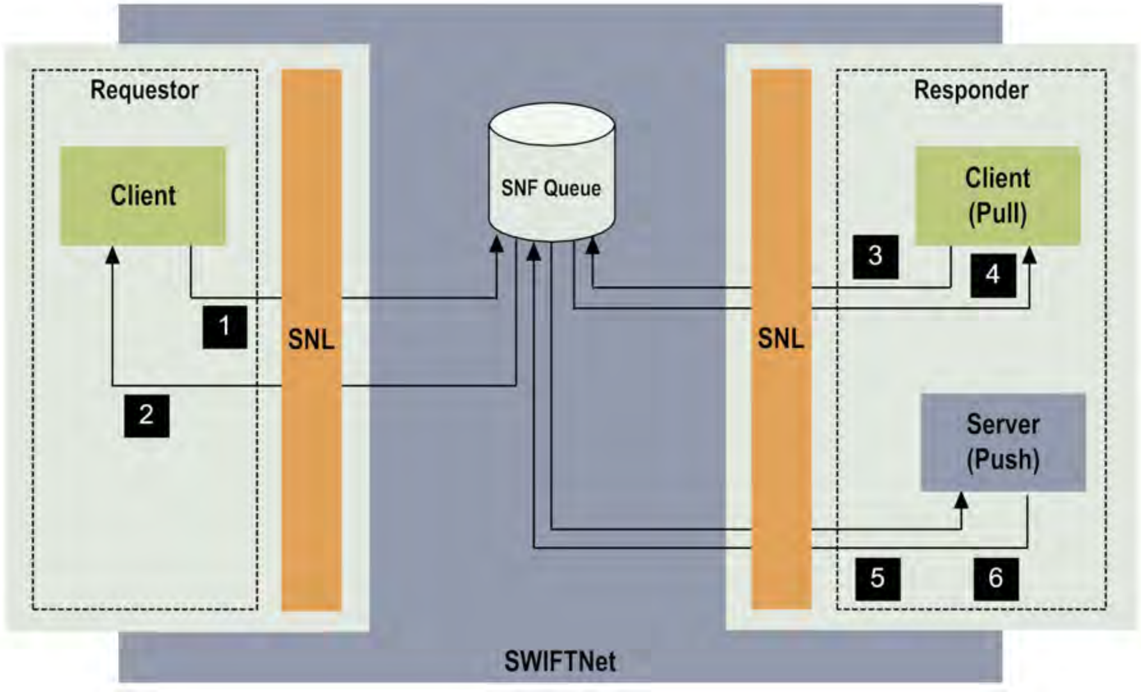
In store-and-forward (SnF) mode, messages and files are stored within SWIFTNet in a queue, and delivered to the receiver at a future time. Therefore, the requestor and responder do not need to be online at the same time. The requestor receives a notification if a message cannot be delivered.

SnF queues contain the requestor's undelivered messages and the files and delivery notifications generated by SWIFTNet SnF. Messages and files in SnF mode can be routed into queues with the same flexibility available for message routing in real-time mode.

The Message Reception Registry function (MRR) specifies the message routing details. The responder defines and configures the available queues. Then the requestor specifies which of these queues to use for the messages or files that the responder sends. (This information is not visible to the responder.)

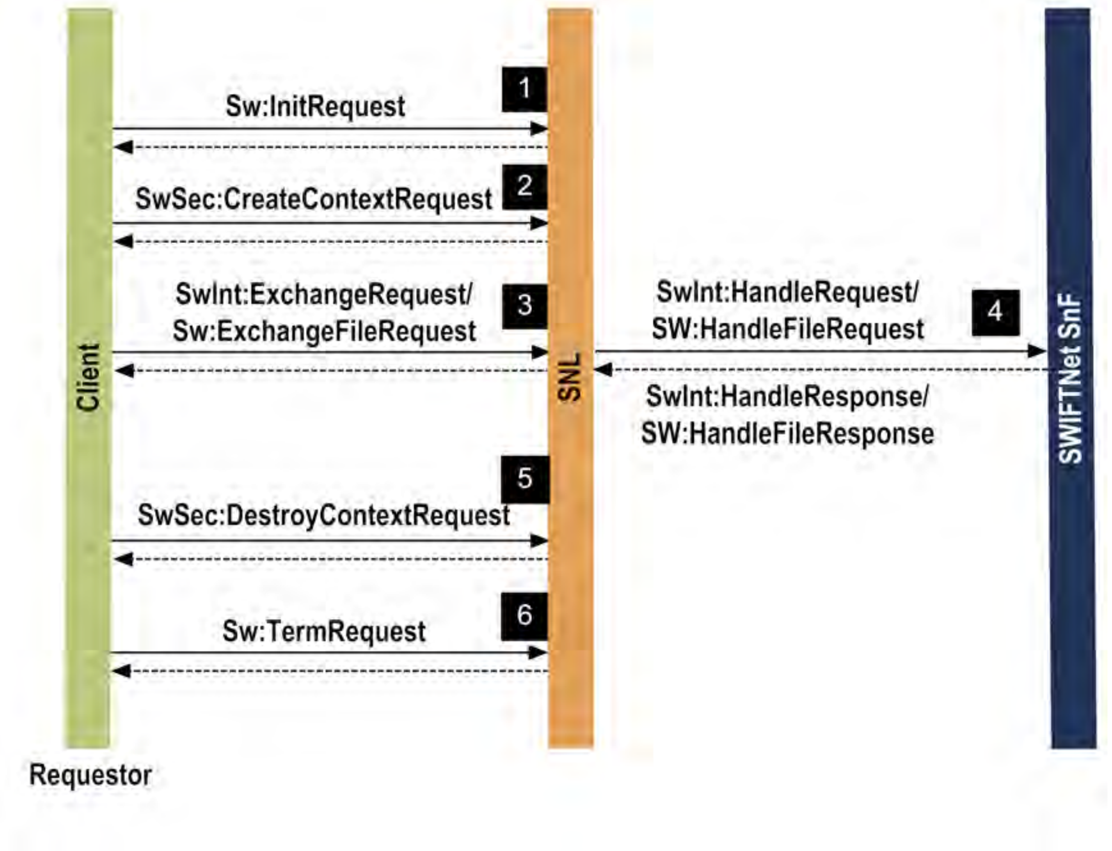
In store-and-forward mode, the response comes from the SWIFTNet SnF queue and does not contain any feedback from the responder. (When real-time mode is used, the responder's server sends the response and interprets the message sent.)

Only the messages or files that are flagged for store-and-forward delivery mode are added to the queue. Flagging can be done within the RequestControl for store-and-forward delivery mode for SWIFTNet InterAct and for SWIFTNet FileAct.



| Step | Description |
|---|---|
| 1<br>2 | Requestor's client sends messages or files to the SnF queue. The SnF queue stores the messages or files received and sends a response to the requestor. |
| 3<br>4 | Responder's client acquires the SnF queue in pull mode and pulls the messages from the SnF queue. |
| 5<br>6 | Responder's client acquires the SnF queue in push mode. The responder's server receives the pushed messages from the SnF queue and sends an acknowledgement. |

The following diagram illustrates the store-and-forward flow on the requestor's side for an InterAct send or FileAct put session.



## Store and Forward InterAct

Store-and-forward InterAct services are used for exchanging messages when the sender and receiver are not online simultaneously. To use this feature, the sender must specify that SnF be used to store the message and indicate the queue in which SWIFTNet SnF should store any delivery notifications that it generates. If the file delivery fails, the failed delivery notification, including the reason the delivery failed, is stored in the queue the sender specified in the RequestControl.

Client processes on the requestor's side initiate requests and related functions, and then pass a SWIFTNet primitive parameter to SNL representing the function to be performed.

```
<?xml version="1.0"?>
<SwInt:ExchangeRequest>
   <SwSec:AuthorisationContext>
      <SwSec:UserDN>cn=abc,o=xxxx,o=swift</SwSec:UserDN>
   </SwSec:AuthorisationContext>
   <SwInt:Request>
      <SwInt:RequestControl>
         <SwInt:RequestCrypto>TRUE</SwInt:RequestCrypto>
         <SwInt:DeliveryCtrl>
            <SwInt:DeliveryMode>SnF</SwInt:DeliveryMode>
            <SwInt:NotifQueue>xxxx_generic!x</SwInt:NotifQueue>
            <Sw:DeliveryNotif>TRUE</Sw:DeliveryNotif>
         </SwInt:DeliveryCtrl>
      </SwInt:RequestControl>
      <SwInt:RequestHeader>
            <SwInt:Requestor>o=xxxx, o=swift</SwInt:Requestor>
            <SwInt:Responder>o=xxxx, o=swift</SwInt:Responder>
            <SwInt:Service>swift.generic.iast!x</SwInt:Service>
      </SwInt:RequestHeader>
      <SwInt:RequestPayload>This is for SnF Queue</SwInt:RequestPayload>
      <SwSec:Crypto>
        <SwSec:CryptoControl>
           <SwSec:MemberRef>RequestPayload</SwSec:MemberRef>
           <SwSec:SignDN>cn=abc,o=xxxx,o=swift</SwSec:SignDN>
        </SwSec:CryptoControl>
   </SwSec:Crypto>
  </SwInt:Request>
</SwInt:ExchangeRequest>
```

If the instruction to trigger store-and-forward mode is not provided in the `SwInt:DeliveryCtrl` element for an SnF service request, then SWIFTNet will reject the message. The `SwSec:UserDN` within the `SwSec:AuthorisationContext` must have the RBAC role "SnFRequestor" with the queue, as specified in the `SwInt:NotifQueue` as qualifier.

The queue in `SwInt:NotifQueue` stores failed delivery notifications. It must belong to the same institution specified in the `SwInt:Requestor`. When the message is stored, SWIFTNet indicates this in the response.

## Store and Forward FileAct

Store-and-forward FileAct Services can only be used to send a file to a receiver. They cannot be used to request a file.

A store-and-forward FileAct request resembles a real-time FileAct request message. The sender must indicate that SnF be used to store the file and indicate the queue in which SWIFTNet SnF should store any delivery notifications that it generates. If the file delivery fails, the failed delivery notification, including the reason the delivery failed, is stored in the queue the sender specified in the RequestControl.

```
<Sw:ExchangeFileRequest>
   <SwSec:AuthorisationC ontext>
      <SwSec:UserDN>cn=abc,o=xxxx,o=swift</SwSec:UserDN>
   </SwSec:AuthorisationContext>
   <Sw:FileRequest>
      <Sw:FileRequestControl>
         <SwInt:RequestCrypto>FALSE</SwInt:RequestCrypto>
         <SwInt:DeliveryCtrl>
           <SwInt:DeliveryMode>SnF</SwInt:DeliveryMode>
           <SwInt:NotifQueue>xxxx_generic!x</SwInt:NotifQueue>
         </SwInt:DeliveryCtrl>
      </Sw:FileRequestControl>
      <Sw:FileRequestHeader>
           <SwInt:Requestor>o=xxxx, o=swift</SwInt:Requestor>
           <SwInt:Responder>o=xxxx, o=swift</SwInt:Responder>
           <SwInt:Service>swift.generic.fast!x</SwInt:Service>
      </Sw:FileRequestHeader>
      <Sw:FileOpRequest>
<Sw:PutFileRequest>
        <Sw:TransferDescription>atlog.txt</Sw:TransferDescription>
        <Sw:PhysicalName>C:\atlog.txt</Sw:PhysicalName> </Sw:PutFileRequest>
</Sw:FileOpRequest>
    </Sw:FileRequest>
</Sw:ExchangeFileRequest>
```

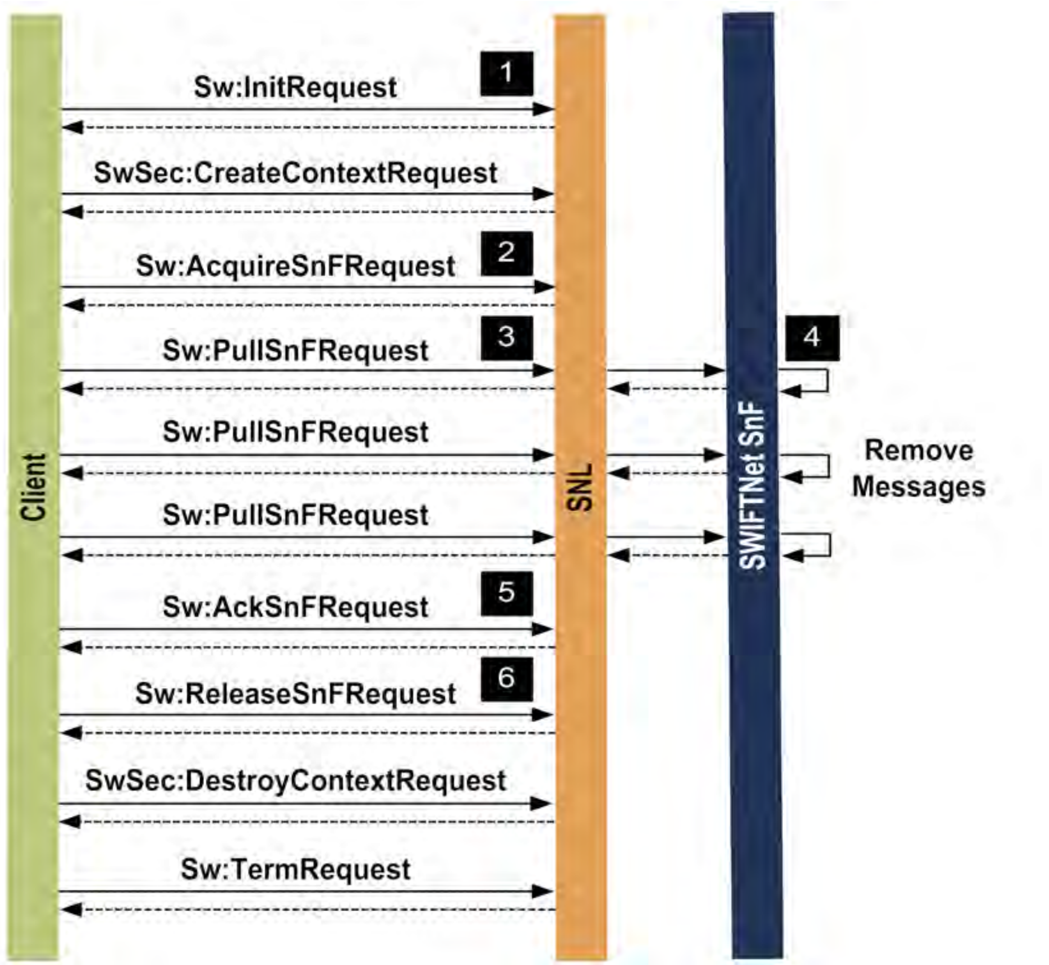
## Retrieving Messages and Files from a Queue

Messages and files can be retrieved from a queue using pull or push modes.

### Pull Mode

When the pull mode is used, the client process initiates the delivery of a message. It performs an SwCall() with `Sw:PullSnFRequest` as the input primitive. The `Sw:PullSnFResponse` contains the message pulled from the queue.

The following diagram illustrates the store-and-forward InterAct pull session. See the table below the diagram for additional information.

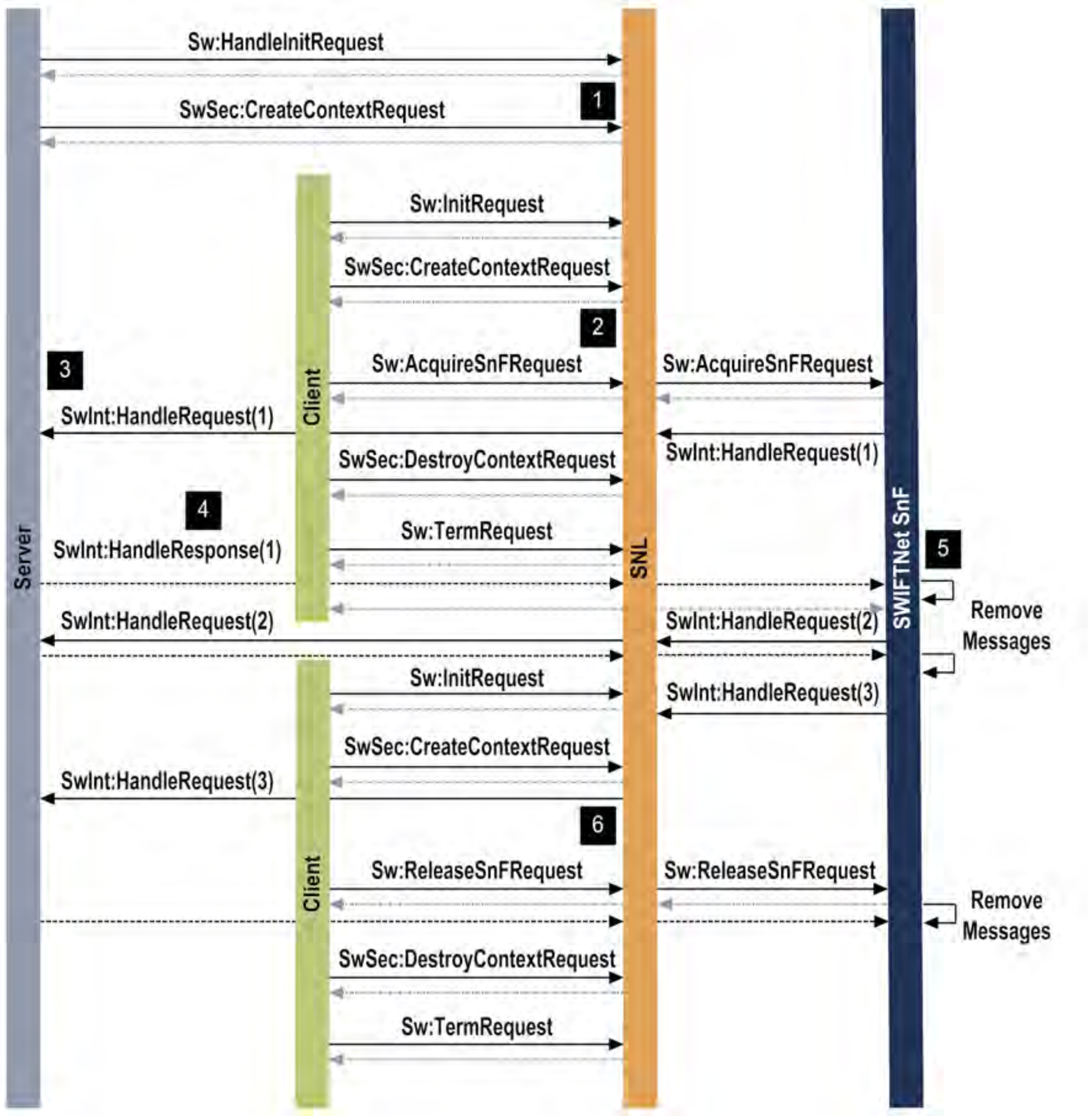


| Step | Description |
|---|---|
| 1 | The client sends the `Sw:InitRequest` to start delivering messages and files in a SnF queue. Next, the client uses `SwSec:CreateContextRequest` to open the desired security context. |
| 2 | The client sends a request to acquire the queue. After receiving the response, the client starts delivering messages by issuing the `Sw:PullSnFRequest`. |
| 3 | The first `Sw:PullSnFRequest` does not carry an acknowledgement, but all subsequent requests must acknowledge the message delivered in the previous pull request to avoid the same message being delivered again. |
| 4 | Messages are removed from the queue. |
| 5 | When the client is finished delivering messages, the client sends `Sw:AckSnFRequest (Sw:ExchangeSnFRequest)` along with the acknowledgement of the last delivered message as input primitive. |
| 6 | The client destroys the created security context and triggers the termination with the SNL. |

### Push Mode

When push mode is used, the initiative to deliver a message resides with SWIFTNet SnF. The message is pushed from SWIFTNet SnF and is received by the server on SWIFTNet Link. In this server, a regular SwCallback() is invoked. The input primitive is the message from the queue within a `SwInt:HandleRequest` or `Sw:HandleFileRequest`. The server application ensures that the message is stored safely, and then responds with an acknowledgement to SWIFTNet SnF indicating how the message was received.

The following diagram illustrates the store-and-forward InterAct Push session. See the table below the diagram for additional information.



| Step | Description |
|---|---|
| 1 | The server process opens the required security context with `Sw:HandleInitRequest` and `SwSec:CreateContextRequest`. The server prepares to process the incoming requests. |
| 2 | The client process starts, sends the `Sw:InitRequest`, opens the desired security context, acquires the queue in pull mode, and starts delivering messages. |
| 3 | The server receives a `SwInt:HandleRequest` request. |
| 4 | The server sends an acknowledgement in `SwInt:HandleResponse`. |
| 5 | Messages are removed from the queue. |
| 6 | The client releases the queue. |

## Fetching a File from a Queue

If a FileAct message is received in either pull mode or push mode, a client process must fetch a file. When the file is available to be fetched, the SWIFTNet SnF does the following:

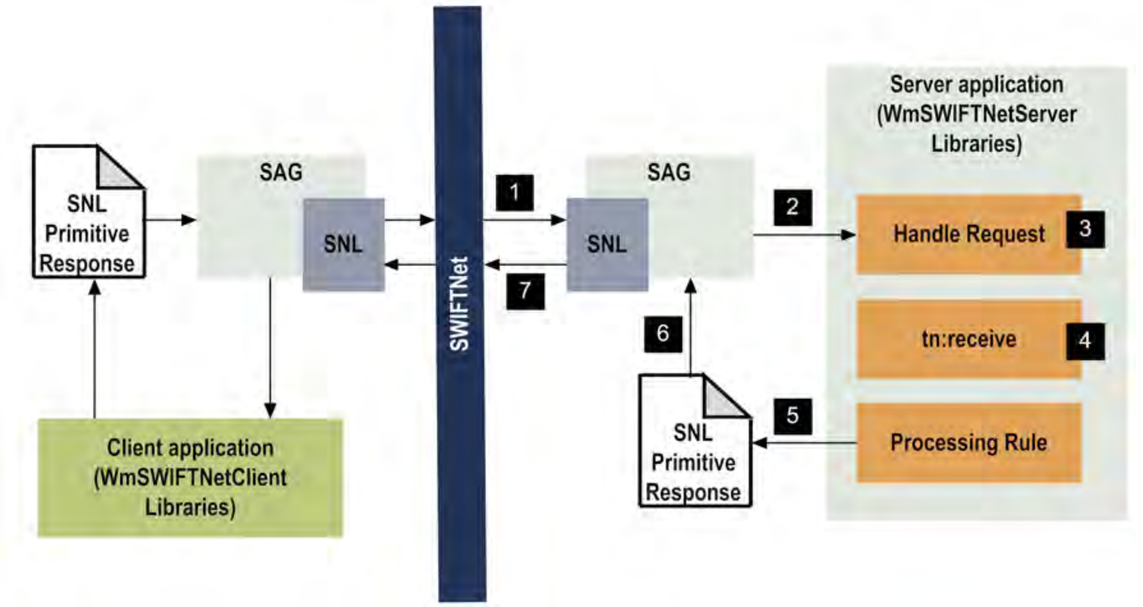
- In push mode, delivers a `Sw:NotifyFileRequestHandle` within `Sw:HandleFileRequest`.
- In pull mode, delievers a and within `Sw:PullSnFResponse` in pull mode.

The `Sw:FetchFileRequest` copies the structure received in the `Sw:FileRequestHandle`. The response is the `TransferRef` that is used to identify the file transfer from SWIFTNet SnF to the receiver. For a pull session, no other message will be delivered for that queue until the file has been fetched and a delivery acknowledgement has been sent.

**Important!** When a file is fetched from the queue, the file will remain within SWIFTNet SnF until an explicit acknowledgement has been sent by a client process.

# Server Application Processing of SNL Primitives

The following diagram illustrates how the server application processes the SNL primitives when a client application sends a request. See the table below the diagram for additional information.



| Step | Description |
|---|---|
| 1 | The requestor's client sends a request to the server via SWIFTNet. |
| 2 | The responder's SAG receives the SNL primitive request and invokes the wm.swiftnet.server.services:handleRequest service in Integration Server on which the server application is installed. |
| 3 | The handleRequest service of the server application then invokes the wm.tn:receive service of Trading Networks. |
| 4 | Trading Networks uses TN document types to recognize the incoming request, saves the request to the Trading Networks database, and invokes one of the processing rules associated with the request's TN document type. |
| 5 | The processing rule processes the document as necessary and generates the XML response. |
| 6 | The XML response is sent to SAG. |
| 7 | SAG returns the response to the requestor's client application via SWIFTNet. |

## SWIFT File Transfer Adapter Support

The SWIFTNet component of SWIFT Module provides integration support for the File Transfer Adapter (FTA) provided by SWIFT. You can use FTA to automate file transfer between parties over SWIFTNet.

SWIFT Module makes the file available on the SWIFT Alliance Gateway (SAG) host. Then the File Transfer interface (FT-interface) provided by SWIFT automatically transfers the file over SWIFTNet using FTA configuration data.

With SWIFT Module, you can also create a companion file with custom parameters to override the FTA configuration data, for example local authentication information, override values, and header information. You can also monitor the SAG input directory for reports that FTA generates about the processing status of data files. For information about how to transfer files using FTA, see Chapter 17, “Using FTA to Transfer Files over SWIFTNet”.

## FpML Message Exchange Support

SWIFT Module provides support for FpML message exchange over SWIFTNet. FpML messages are XML messages for the transfer of Over The Counter (OTC) derivatives over SWIFTNet. FpML-compliant messages in XML format are transferred over SWIFTNet using the SWIFTNet InterAct store-and-forward messaging service.

**Important!** Before you can exchange FpML messages over SWIFTNet, you must register with SWIFT.

SWIFT Module provides the following support for the transfer of FpML messages:

- Store-and-Forward mode of message exchange over SAA in XML v2 format or directly over SAG.
- Reconciliation of delivery notifications with the original messages.
- Populating the XML v2 header when sending FpML messages to SAA or SAG. You can set similar flags in an XML v2 message when sending a message to the counterparty.
- Message validation.
- Built-in support for schema validation.
- Semantic validation of all XML messages.

The non-repudiation of emission and reception of transferred messages is also required for the exchange of FpML messages over SWIFTNet, but this requirement is handled by SAA or SAG. SWIFT Module does not provide any support for signature generation.

For more information about FpML message exchange over SWIFTNet, see the SWIFT documentation.

FpML messages are based on FpML schemas provided by SWIFT that you can import using the Integration Server functionality for creating a schema. You can create the corresponding document types for the FpML schemas using the Integration Server functionality for creating IS document types. For more information about creating IS schemas and IS document types, see the Designer online help for your release and “Step 3: Create IS Schema and IS Document Type” on page 137. See “About this Guide” for specific document titles.

# 2 Installing webMethods SWIFT Module

## Overview

This chapter, along with thewebMethods installation guide for your release, explains how to install, upgrade, and uninstall webMethods SWIFT Module 7.1 SP1. See "About this Guide" for specific document titles.

**Important!** For the webMethods 8 release, the installer you use to install the module is named Software AG Installer 8. Previously, the installer was named webMethods Installer 7. If you are installing the module with webMethods 8 products, you must use Software AG Installer 8 and the webMethods installation guide for the 8.x release. If you are installing the module with webMethods 7.x, you can use Software AG Installer 8 with the webMethods installation guide for the 8.x release or webMethods Installer 7 with *webMethods Installation Guide 7.x*. See "About this Guide" for specific document titles.

## Requirements

For a list of the operating systems and webMethods products supported by SWIFT Module 7.1 SP1, see *webMethods eStandards Modules System Requirements*. SWIFT Module 7.1 SP1 has no hardware requirements beyond those of its host Integration Server.

Depending on the type of transport you use, you will need either the MQ Host Adapter (MQHA) or the Remote API Host Adapter (RAHA). If you are using the SWIFTNet component of SWIFT Module for a server application, you must install RAHA or MQHA on the same machine as SWIFT Alliance Gateway.

Regardless of whether you are using the SWIFTNet component for a client or server application, you must install a Remote Access (RA) client on your Integration Server. The RA client, RAHA, and MQHA are provided by SWIFT. For more information, see your SWIFT documentation or go to http://www.swift.com.

If you are using CASmf as the interface to SWIFT, you must install a CASmf client (provided by SWIFT) on the same machine as your Integration Server. For more information, see your SWIFT documentation or go to http://www.swift.com.

SWIFT Module references SWIFT Alliance Access and SWIFT Alliance Gateway through the interfaces provided by SWIFT. SWIFT Alliance Access and SWIFT Alliance Gateway use SWIFTNet Link to communicate with SWIFT.

## Installing webMethods SWIFT Module 7.1 SP1

The instructions in this section are meant to be used with the more complete instructions in the webMethods installation guide for your release. The instructions explain how to use the Software AG Installer wizard.

**Note:** If you are installing SWIFT Module in a clustered environment, you must install it on each Integration Server in the cluster, and each installation must be identical. For more information about working with SWIFT Module in a clustered environment, see Appendix C, "Administering webMethods SWIFT Module in a Cluster".

### To install SWIFT Module 7.1 SP1

1. Download the Software AG Installer from the Empower Product Support Web site at https://empower.softwareag.com.
2. If you are installing SWIFT Module on an existing Integration Server, shut down the Integration Server.
3. Start the Software AG Installer wizard.
   - Choose the webMethods release that includes the Integration Server on which to install the module. For example, if you want to install the module on Integration Server 7.1, choose the 7.1 release.
   - If you are installing on an existing Integration Server, specify the webMethods installation directory that contains the host Integration Server. If you are installing both the host Integration Server and the module, specify the installation directory to use. The installer will install the module in the *Integration Server_directory*\packages directory.
   - In the product selection list, navigate to **eStandards > webMethods SWIFT Module7.1 SP1**. You can also choose to install any required products indicated in the *webMethods eStandards Modules System Requirements*.

     The installer installs the following components:

     - webMethods Integration Server
     - webMethods Trading Networks
     - eStandards Common Library

     SWIFT Module installs the following packages in the *Integration Server_directory*\ packages directory:

     - WmFIN
     - WmSWIFTCommon
     - WmSWIFTNetClient
     - WmSWIFTNetServer

   **Note:** If Integration Server and Trading Networks are already installed from a previous installation, the installer does not reinstall these products.
4. After the installer completes the installation, close it.
5. Start the host Integration Server.

6 If you are using Integration Server with webMethods Broker, enable publication of messages to webMethods Broker as follows:

   a Navigate to the *Integration Server_directory*\packages\WmFIN\config directory.

   b Open the fintransport.cnf file in a text editor and change the *fin.message.publishLocal* parameter to `false`.

   c Save and close the file.

## Installing the SWIFT Module Samples Package

The SWIFT Module samples package (WmSWIFTSamples) contains the sample services to run SWIFT Module. The samples package is not installed with SWIFT Module 7.1 SP1. To download the WmSWIFTSamples package and *webMethods SWIFT Module Samples Guide*, go to the Developer Community for webMethods at http://communities.softwareag.com/ecosystem/communities/public/Developer/webmethods/products/esb/ and see the Code Samples.

## Upgrading to SWIFT Module 7.1 SP1

This section describes how to upgrade and migrate the services created in:

- webMethods SWIFT Module 7.1 to SWIFT Module 7.1 SP1
- webMethods SWIFT FIN Module 6.1 Service Pack 4 to SWIFT Module 7.1 SP1
- webMethods SWIFTNet Module 6.0.1 Service Pack 1 to SWIFT Module 7.1 SP1

### Before You Begin

Before you upgrade to SWIFT Module 7.1 SP1, you must migrate to Integration Server, Trading Networks, Designer, and Monitor version 7.1.2 or later from the equivalent 6.1.x or 6.5.x versions of those products. For instructions, see the webMethods upgrade guide for your release. See “About this Guide” for specific document titles.

When you are upgrading to a newer version of Trading Networks make sure that you back up the SWIFT Module packages that contain Trading Networks specific data. For more information on how to upgrade Trading Networks, see the webMethods upgrade guide for your release. See “About this Guide” for specific document titles.

**Note:** The 6.5.x equivalent for Designer was Modeler.

## Upgrading from SWIFT Module 7.1

### To upgrade from SWIFT Module 7.1 with latest fixes

1. Back up your existing SWIFT Module 7.1 (with latest fixes applied) installation and all custom packages that are used by SWIFT Module.

2. Export all SWIFT Module 7.1 Trading Networks information (profiles, TN document types, processing rules, TPAs and TN attributes) from Trading Networks using My webMethods Server.

   For information about exporting Trading Networks assets, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

3. Shut down Integration Server if it is running.

4. Uninstall SWIFT Module 7.1. For instructions, see *webMethods SWIFT Module 7.1 Installation and User's Guide.*

   **Note:** You must remove SWIFT Module related packages manually. To do so, navigate to the *Integration Server_directory*\packages directory and delete the WmFIN and WmSWIFT-related folders.

5. Install SWIFT Module 7.1 SP1 on a supported version of Integration Server. For instructions, see "Installing webMethods SWIFT Module 7.1 SP1" on page 52. For a list of supported Integration Server versions, see *webMethods eStandards Modules System Requirements*.

6. If you want to preserve the previous configuration values, replace the following files:

   - \WmFIN\config\properties.cnf with the backup of \WmFIN\config\properties.cnf from the SWIFT Module 7.1 installation.
   - \WmFIN\config\wmcasmf.cnf with the backup of \WmCASmf\config\wmcasmf.cnf from the SWIFT Module 7.1 installation.
   - \WmFIN\config\fintransport.cnf with the backup of \WmFINTransport\config\fintransport.cnf from the SWIFT Module 7.1 installation.

7. If you want to use any of the SWIFT Module 7.1 configuration values for the RAHA transport, you must configure the SWIFTNet server application and client applications.

   a. In Integration Server Administrator, select **Adapters > SWIFT**.

   b. Configure the SWIFTNet server with the following information from the SWIFT Module 7.1 installation:

1. From the SWIFT navigation area, select **SWIFTNet Server Config > Edit.**

2. In the SWIFTNet Remote Process Connection Configuration section, enter the values from the backup of the \WmSWIFTNetServer\config\connect.cnf file. This file stores the password handle for the user password that you need to connect to Integration Server.

3. In the SWIFTNet Server Environment Information section, enter the values from the backup of the \WmSWIFTNetServer\config\env.cnf file.

4. In the SWIFTNet Server SAG Connection Properties section, enter the values from the backup of the \WmSWIFTNetServer\config\snl.cnf file.

5. Click **Save** when you are done.

c Configure the SWIFTNet client, providing the following information from the SWIFT Module 7.1 installation, as follows:

1. From the SWIFT navigation area, select **SWIFTNet Client Config > Edit.**

2. In the SWIFTNet Client Environment Information section, enter the values from the backup of the \WmSWIFTNetClient\config\env.cnf file.

3. In the SWIFTNet Client SAG Connection Configuration section, enter the values from the backup of the \WmSWIFTNetClient\config\snl.cnf file.

4. Click **Save** when you are done.

For information about the fields in the SWIFTNet Server and Client Configuration screen, and how to configure the SWIFTNet server application and client application over RAHA, see Chapter 15, "Configuration Steps for InterAct and FileAct Messaging Services over SAG RAHA" on page 149.

8. Start Integration Server, the Integration Server Administrator, and My webMethods Server.

9. Import the custom SWIFT Module 7.1 Trading Networks information that you exported in step 2 into Trading Networks. For instructions about how to import Trading Networks information, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

10. In **Designer > Package Navigator**, run the wm.fin.dev:importFINItems service to re-import all message types that you have been using with SWIFT Module 7.1.

**Important!** Before re-importing the message types, you must delete any existing IS document types in the wm.fin.doc.*version.category* folders in Designer to avoid problems that may arise from having older versions of the IS document types.

## Upgrading from SWIFT FIN Module 6.1 Service Pack 4

### To upgrade from SWIFT FIN Module 6.1 Service Pack 4

1. Back up your existing SWIFT FIN Module 6.1 Service Pack 4 installation and all custom packages that are used by SWIFT FIN Module.
2. Export all SWIFT FIN Module 6.1 Service Pack 4 Trading Networks information (partner profiles, processing rules, TPAs, and TN document types) from Trading Networks using the Trading Networks Console export function. (For information about exporting Trading Networks assets, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.)
3. Shut down Integration Server if it is running.
4. Uninstall SWIFT FIN Module 6.1 Service Pack 4. For instructions, see *webMethods SWIFT FIN Module Installation and User's Guide*.

   **Note:** You must remove SWIFT FIN Module related packages manually. To do so, navigate to the *Integration Server_directory*\packages directory and delete the WmFIN-related folders.

5. Install SWIFT Module 7.1 SP1 on a supported version of Integration Server. For instructions, see "Installing webMethods SWIFT Module 7.1 SP1" on page 52. For a list of supported Integration Server versions, see *webMethods eStandards Modules System Requirements*.
6. If you want to preserve the previous configuration values, replace the following files:
   - \WmFIN\config\properties.cnf with the backup of \WmFIN\config\properties.cnf from the SWIFT FIN Module 6.1 Service Pack 4 installation.
   - \WmFIN\config\wmcasmf.cnf with the backup of \WmCASmf\config\wmcasmf.cnf from the SWIFT FIN Module 6.1 Service Pack 4 installation.
   - \WmFIN\config\fintransport.cnf with the backup of \WmFINTransport\config\fintransport.cnf from the SWIFT FIN Module 6.1 Service Pack 4 installation.
7. Start Integration Server.
8. Import the custom SWIFT FIN Module Trading Networks information that you exported in step 2 into Trading Networks. For instructions for how to import Trading Networks information, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

9 In **Designer > Package Navigator**, run the wm.fin.dev:importFINItems service to re-import all message types that you have been using with SWIFT FIN Module 6.1 Service Pack 4.

**Important!** Before re-importing the message types, you must delete any existing IS document types in the wm.fin.doc.*version.category* folders in Designer to avoid problems that may arise from having older versions of the IS document types.

10 With SWIFT Module 7.1 SP1, all services from the WmIPCore package of SWIFT FIN Module 6.1 have been moved to the WmEstdsCommonLib package. The WmIPCore package no longer exists. To migrate your references to services in the WmIPCore package to the corresponding services in the WmEstdsCommonLib package, do the following:

In **Designer > Package Navigator**, run the "com.wm.common.Util:migrateServices" on page 272 service included in SWIFT Module 7.1 SP1, WmSWIFTCommon package. For details about the migration service, see Appendix A, "Services".

**Note:** If you are using mappings for MT document types from the setup of SWIFT FIN Module 6.1 Service Pack 4 with earlier fixes, the MT message structure may be different than the MT message structure required for SWIFT Module 7.1 SP1.

## Upgrading from SWIFTNet Module 6.0.1 Service Pack 1

### To upgrade from SWIFTNet Module 6.0.1 Service Pack 1

1 Back up your existing webMethods SWIFTNet Module 6.0.1 Service Pack 1 installation and all custom packages that are used by SWIFTNet Module.

2 Export all SWIFTNet Module 6.0.1 Service Pack 1 Trading Networks information (profiles, TN document types, processing rules, and TN attributes) from Trading Networks using the Trading Networks Console export function.

For information about exporting Trading Networks assets, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

3 Shut down Integration Server if it's running.

4 Uninstall SWIFTNet Module 6.0.1 Service Pack 1. For instructions, see *webMethods SWIFTNet Module Installation and User's Guide.*

**Note:** You must remove the SWIFTNet Client related packages manually. To do so, navigate to the *Integration Server_directory*\packages directory and delete the WmSWIFTNetClient-related folders.

5 Install SWIFT Module 7.1 SP1 on a supported version of Integration Server. For instructions, see "Installing webMethods SWIFT Module 7.1 SP1" on page 52. For a list of supported Integration Server versions, see *webMethods eStandards Modules System Requirements*.

6 Start Integration Server, Integration Server Administrator, and My webMethods Server.

7 If you want to use any of the SWIFTNet Module 6.0.1 Service Pack 1 configuration values for the RAHA transport, configure the SWIFTNet server application and client application as follows:

   a In Integration Server Administrator, select **Adapters > SWIFT**.

   b Configure the SWIFTNet server, as follows:

      1 From the SWIFT navigation area, select **SWIFTNet Server Config > Edit**. Provide the following information from the SWIFTNet Module 6.0.1 Service Pack 1 installation:

      2 In the SWIFTNet Remote Process Connection Configuration section, enter the values from the backup of the \WmSWIFTNetServer\config\connect.cnf file. This file stores the password handle for the user password that you need to connect to Integration Server.

      3 In the SWIFTNet Server Environment Information section, enter the values from the backup of the \WmSWIFTNetServer\config\env.cnf file.

      4 In the SWIFTNet Server SAG Connection Properties section, enter the values from the backup of the \WmSWIFTNetServer\config\snl.cnf file.

      5 Click **Save** when you are done.

   c Configure the SWIFTNet client as follows:

      1 From the SWIFT navigation area, select **SWIFTNet Client Config > Edit**. Provide the following information from the SWIFTNet Module 6.0.1 Service Pack 1 installation:

      2 In the SWIFTNet Client Environment Information section, enter the values from the backup of the \WmSWIFTNetClient\config\env.cnf file.

      3 In the SWIFTNet Client SAG Connection Configuration section, enter the values from the backup of the \WmSWIFTNetClient\config\snl.cnf file.

      4 Click **Save** when you are done.

   For information about the fields in the SWIFTNet Server and Client Configuration screen, and how to configure the SWIFTNet server application and client application over RAHA, see Chapter 15, "Configuration Steps for InterAct and FileAct Messaging Services over SAG RAHA" on page 149.

8 Import the SWIFTNet Trading Networks information that you exported in step 2 into Trading Networks. For instructions how to import Trading Networks information, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

## Uninstalling SWIFT Module 7.1 SP1

The instructions in this section are meant to be used with the uninstallation instructions in the webMethods installation guide for your release. See “About this Guide” for specific document titles.

**Important!** Before you uninstall SWIFT Module, stop all RMI Registry services that are running.

### To uninstall SWIFT Module 7.1 SP1

1. Uninstalling SWIFT Module 7.1 SP1 removes all components in the SWIFT Module packages. If you want to keep certain records or services from the existing SWIFT Module packages on your Integration Server, export them to a new package.

   To do so, open Designer, select the package or the service you want to export, and select **File > Export.**

   - If you select a package, the entire package is exported.
   - If you select a service, only the selected service is exported.

2. Shut down the Integration Server that hosts SWIFT Module 7.1 SP1.

3. Start the Software AG Uninstaller, selecting the webMethods installation directory that contains the host Integration Server. In the product selection list, select **eStandards > webMethods SWIFT Module 7.1 SP1** and any other products and items you want to uninstall.

4. Restart the host Integration Server.

5. The Software AG Uninstaller removes all SWIFT Module 7.1 SP1-related files that were installed into the *Integration Server_directory*\packages directory. However, the Uninstaller does not delete files created after you installed the module (for example, user-created files or configuration files), nor does it delete the module directory structure. You can navigate to the *Integration Server_directory*\packages directory and delete the WmSWIFT-related directories.

# II Configuring SWIFT Module for Message Exchange Over SAA

# 3 Configuration Steps for Message Exchange over SAA

## Overview

This chapter describes how to configure Integration Server to prepare to send and receive SWIFT FIN messages using the services in SWIFT Module. The subsequent chapters in this guide provide more detailed information about each of these steps.

To see sample SWIFT Module services that demonstrate how to send and receive SWIFT messages and to learn more about how to install and use other SWIFT Module samples, see *webMethods SWIFT Module Samples Guide*.

Important! The following steps assume that you have already installed Integration Server, Trading Networks, Designer, My webMethods Server, the necessary SWIFT software, the SWIFT Module packages, and the appropriate software for the SWIFT interface that you want to use. For more information about what SWIFT software you need, work with SWIFT to determine your software needs. For more information about installing SWIFT Module, see Chapter 2, “Installing webMethods SWIFT Module” on page 51.

## Step 1: Import BICPlusIBAN List

SWIFT Module provides support for deriving or validating data against the BICPlusIBAN and SEPARouting directories. You can use those directories to supply or validate data in international payment messages, for example to translate the beneficiary bank's BIC into national (clearing, sort) code, and in SEPA (Single Euro Payment Area) payments, for example to derive the BIC from the IBAN, if missing, or validate IBAN/BIC combinations. To derive or validate data against the BICPlusIBAN and SEPA Routing directories, you must import the BICPlusIBAN, IS and SR lists provided by SWIFT using the SWIFT Module user interface.

Before you import a list, you first must create a database table to hold the SWIFT list. For instructions on how to create a database table and import a list into Integration Server, see Chapter 4, “Importing BICPlusIBAN and SEPA Routing Directories”.

## Step 2: Define Trading Partner Profiles

In My webMethods, define the Trading Networksassets required for processing messages.

- For each SWIFT message that you want to exchange with your partner financial institutions, you must create a message record by running the wm.fin.dev:importFINItems service for the message DFD.
- Create trading partner profiles for your organization and for all the financial institutions with whom you will exchange SWIFT FIN messages.
- Modify Trading Partner Agreements for each of your partners.

For more information about defining trading partner profiles, see Chapter 5, “Defining Trading Networks Information”.

## Step 3: Create Validation Rules

Software AG provides network validation rules for a number of commonly used message types. In addition to these rules, SWIFT Module enables you to create network validation rules for additional messages as well as create usage validation rules.

For more information about creating validation rules, see Chapter 6, “Creating Validation Rules”.

## Step 4: Write Inbound and Outbound Mapping Services

To send and receive messages, you must create inbound and outbound mapping services that define how SWIFT Module should process each message. These services are used in the management of SWIFT message execution that you define in “Step 6: Manage SWIFT Message Processing Rules and Message Execution” on page 66.

- Create an outbound mapping service to map a message from the format of a back-end system to SWIFT message format before sending it to another financial institution.
- Create an inbound mapping service to map a SWIFT message received from another financial institution to an internal message format.

For more information about mapping business documents, see Chapter 7, “Creating Inbound and Outbound Mapping Services”.

## Step 5: Modify Trading Partner Agreements

A trading partner agreement (TPA) is a set of parameters that govern how you exchange a SWIFT message with a trading partner. When running the wm.fin.dev:importFINItems service for a message DFD to create a record, you can also choose to create the corresponding TPA.

Before the system can use the TPA, you must modify the parameters as needed to process the message, and then set the TPA's status to “Agreed.”

**Important!** The *subfieldFlag* input parameter in the TPA supports the parsing of SWIFT messages to the subfield level. If you change the value of the *subfieldFlag* after creating the message records, you must delete the message records pertaining to the TPA and recreate them.

For information about modifying TPAs for use with SWIFT FIN messages, see Chapter 8, “Customizing Trading Partner Agreements”.

## Step 6: Manage SWIFT Message Processing Rules and Message Execution

Trading Networks processing rules manage the execution of SWIFT FIN messages. When you run the importFINDev service to import the message record, a processing rule is also created.

For information about sending and receiving SWIFT FIN messages using processing rules, see Chapter 9, “Configuring Processing Rules to Send and Receive SWIFT FIN Messages”. For general information about Trading Networks processing rules, see the Trading Networks administration guide for your release. See “About this Guide” for specific document titles.

## Step 7: Configure SWIFT Interfaces

Use the instructions in the section listed to configure the SWIFT interface you are using:

| To configure the SWIFT interface... | Use instructions in this section... |
|---|---|
| MQHA | “Using WebSphere MQ Adapter to Communicate with SWIFT” on page 110. |
| CASmf | “Using the CASmf Services to Communicate with SWIFT” on page 112. |
| AFT | “Using AFT to Communicate with SWIFT” on page 116. |

## Step 8: Configure Notification Processing

SWIFT Module handles all levels of notifications: information, transmission, and delivery that SWIFT Alliance Access sends in response to requests from SWIFT Module. SWIFT Module allows the site to handle the notifications by setting-up site-specific processing using Trading Networks processing rules. SWIFT Module also provides the TN documents required to recognize the notification messages.

Incoming notifications are saved in Trading Networks and associated with the original message by a Trading Networks processing rule. You can see all related messages by viewing the documents in Trading Networks.

For more information about configuring notifications, see Chapter 12, “Configuring Notifications for Messages in XML v2 Format”.

## Step 9: Configure MT/MX Message Exchange Over SAA

SWIFT Module sends and receives MT and MX messages to SWIFT Alliance Access wrapped in XML v2 data format. XML v2 format is an XML-based format for information exchange between back-end applications and SAA. You can configure custom services process incoming messages from SAA.

For information about how to transport MT and MX messages to SAA in XML v2 data format, see Chapter 13, "Using SAA to Exchange XML v2 Wrapped MT and MX Messages".

# 4 Importing BICPlusIBAN and SEPA Routing Directories

# Overview

SWIFT Module provides support for deriving or validating data against the BIC or BICPlusIBAN directory. A BIC list provides a list of valid BICs for all existing SWIFT financial institutions. All SWIFT FIN messages are validated against this BIC list to ensure that the sender and receiver are valid. BICPlusIBAN is a SWIFT directory that contains identifiers recognized by financial institutions, such as Bank Identifier Codes (BICs), International Bank Account Numbers (IBANs), and national clearing codes. The SWIFT BICPlusIBAN directory serves two main purposes:

- Provides or validates data in international payment messages, for example to translate the beneficiary bank's BIC into national (clearing, sort) code, or validate the banks' details (such as name and address).
- Provides or validates data in SEPA (Single Euro Payment Area) payments, for example to derive the BIC from the IBAN if missing, or validate IBAN/BIC combinations.

SWIFT Module also supports the SEPA Routing Directory that contains the BICs and names of the financial institutions that have signed the SEPA Credit Transfer Adherence Agreement, the operational BICs of institutions that are able to process the SEPA payments, the channels through which the financial institutions can receive the SEPA payments, and the channel preference. With the SEPA Routing Directory, financial institutions sending SEPA payments can:

- Verify that the operational BICs of their correspondent are SEPA-adherent and operationally ready for SEPA.
- Verify the available channels (SEPA-ready Automated Clearing Houses) for payment routing.

## Using the Search BIC Information Tool

SWIFT Module enables you to import and search financial institution data in the BIC directory, BICPlusIBAN directory, and the SEPA Routing directory. The Search BIC Information or Search BICPlusIBAN Information tool is useful when you are preparing an instruction and need to verify BIC information. For more information, see “Searching BIC Information” on page 74.

### To use the search BIC Information tool and the BICPlusIBAN and/or SEPA Routing directories

1. Use Integration Server Administrator to import the following data lists:

   - BIC—Contains institution identifiers, such as BIC-related data of financial institutions.
   - BICPlusIBAN—Contains institution identifiers, such as BICs and IBAN-related data of financial institutions.
   - IBAN Structure (IS)—Contains records with the country codes and national ID codes, as well as the position of those codes in the IBAN structure. You must use the BICPlusIBAN list together with the IBAN Structure list to derive a BIC from an IBAN, to validate a BIC, a Bank ID, or an IBAN/BIC combination.
   - SEPA Routing (SR)—Contains SEPA identifiers. Use this list to validate the BICs and IBANs in SEPA payments against the SWIFT BICPlusIBAN directory.

   See “Importing Lists” on page 71 for details how to import the lists.

2. In Designer, use one of the BIC/BICPlusIBAN or SEPA built-in services supported by SWIFT Module.

For examples of business scenarios in which you would use SWIFT Module to derive or validate data against the BICPlusIBAN or SEPA Routing directory, see “Business Examples of Using the BICPlusIBAN Directory” on page 72 and “Business Examples of Using the SEPA Routing Directory” on page 73.

See Appendix A, “Services” for details about the BIC/BICPlusIBAN and SEPA related built-in services, provided by SWIFT Module.

## Importing Lists

You can import a BIC, BICPlusIBAN, IS, or SR list into Integration Server by creating a database table to hold the list data for each type of list. webMethods provides database scripts to create empty database tables for storing the imported data. These scripts support different databases.

After you have created the empty data table for each type of list, you can import the most recent list of SWIFT data into the respective database table from the CD provided to you by SWIFT. To do so, use the **Import List** tool on the SWIFT Module user interface in Integration Server Administrator.

**Note:** SWIFT Module provides sample BIC, BI (BICPlusIBAN), IS, and SR files that you can format as needed, before importing through the Import Lists feature of SWIFT Module. You can import the .txt files for the listed sample database files from the following location *Integration Server_directory*\packages\WmFIN\config\bic\samples.

## Creating an Empty Database Table

### To create an empty database table

1. With your database server running, verify that you have a Trading Networks database in which to add a table.

   **Note:** SWIFT Module works with the same database as Trading Networks for the BIC, BICPlusIBAN, and the SEPA Routing Directories feature.

2. Follow your database server's documentation instructions and import the file list into your selected database. Use the following table as a guide to locate the file that corresponds to the type of list you want to import.

   **Note:** webMethods provides scripts for SQL Server, Oracle, and DB2 databases at the specified location. The *db* in the file name indicates the name of the database to use. Valid values are SQLServer, Oracle, and DB2.

| List Type | File<br>*Integration Server_directory*\packages\WmFIN\config\bic\... |
|---|---|
| BIC | create_BIC_*db*.sql |
| BICPlusIBAN | create_IBAN_*db*.sql |
| IS | create_IS_*db*.sql |
| SR | create_SR_*db*.sql |

## Importing a List

### To import a list

1. In Integration Server Administrator, select **Adapters > SWIFT**.
2. On the SWIFT Module home page, click the name of the list you want to import, for example **Import BICPlusIBAN List**.
3. In the **File Name** box, type the full directory path and name of the list data file that you want to import. To locate the file, you can click **Browse**.
4. Click **Import**. The new list is imported into the database table for this list type and made available to SWIFT Module. This process might take a few minutes.

# Business Examples of Using the BICPlusIBAN Directory

The following table provides examples of business scenarios in which you would use SWIFT Module to derive or validate data against the BICPlusIBAN directory:

| To... | Use the... |
|---|---|
| Search for the bank details of a financial institution (for example name and address) to create or validate international payments. | Search BIC Information function of the SWIFT Module user interface as described in "Searching BIC Information" on page 74. |
| Validate the national code (National ID) of a bank and the clearing system used in an international payment. | wm.fin.bic:validateBankID service |
| Translate IBAN into BIC, if the BIC is missing, for SEPA payments or other cross-border payments. | wm.fin.bic:deriveBICfromIBAN service |
| Construct IBAN from an account number. | wm.fin.bic:generateIBAN service |
| Validate if the BIC of a financial institution is valid when you want to order payment to that institution. | wm.fin.bic:validateBICCode service |
| Validate that the BIC and the IBAN in SEPA payments belong to the same financial institution. | wm.fin.bic:validateBICIBAN service |

For information about the business logic behind these services, see your SWIFT documentation or go to http://www.swift.com.

## Business Examples of Using the SEPA Routing Directory

The following table provides examples of business scenarios in which you would use SWIFT Module to derive or validate data against the SEPA Routing directory:

| To... | Use the service... |
|---|---|
| Determine if a BIC is ready to receive SEPA payment instructions for a particular scheme and verify an institution's operational readiness for SEPA schemes. | wm.fin.sepa:checkOperationalReadiness |
| Confirm that an institution has signed an adherence agreement and that it is published in the adherence database for a particular scheme. | wm.fin.sepa:validateAdherenceStatus |
| List the different payment channels available for an institution when you want to determine the optimal channel among the available options. | wm.fin.sepa:getAvailablePaymentChannels |

| To... | Use the service... |
|---|---|
| Determine whether an institution has specified a preferred payment channel for receiving payment instructions. | wm.fin.sepa:getPreferredPaymentChannel |
| Identify other payment channels, for example when the sending institution can not use the payment channel through which the receiving institution is indirectly reachable, or there is no matching payment channel between the sending and receiving institutions. | wm.fin.sepa:getOtherPaymentChannel |

For information about the business logic behind these services, see your SWIFT documentation or go to http://www.swift.com.

## Searching BIC Information

Using the **Search BIC Information** tool on the SWIFT Module home page in Integration Server Administrator, you can search the most recent SWIFT BIC and BICPlusIBAN lists published by SWIFT in SWIFT Module.

### To search BIC information from BIC or BICPlusIBAN directories

1 In the Integration Server Administrator, on the SWIFT Module home page, click either **Search BIC Information** or **Search BICPlusIBAN Information.**

2 On the BIC Search Criteria screen, provide your search criteria. You must enter information in at least one field, but can narrow search results by entering information in as many fields as necessary.

**Note:** All fields on this screen are case-sensitive. If you want to do a partial search, enter '%partial search string%'.

| Field Name | Description |
|---|---|
| **Code** | The institution's BIC code, for example, `ABCDEFGHIJK`. |
| **Institution** | The institution's name, for example, `Citibank`. |
| **Branch** | The institution's branch name, for example, `Main`. |
| **City** | The institution's city, for example, `Miami`. |

| Field Name | Description |
|---|---|
| Modified Flag | From the drop-down list, select the modification flag for which you want to search:<br>■ `New`. Search for a new BIC entry.<br>■ `Update`. Search for a BIC entry currently being updated.<br>■ `Modified`. Search for a modified BIC entry.<br>■ `Deleted`. Search for a deleted BIC entry. |
| Location | The institution's location, for example, `Mall`. |
| Country Name | The institution's country, for example, `USA`. |

3 Click **Search**. The **Search Results** screen displays up to the first 50 matching BICs.

# 5 Defining Trading Networks Information

## Overview

This chapter provides information about defining Trading Networks assets, including trading partner profiles and TN document types.

## About Message Records

Before you can send and receive SWIFT FIN messages, you must first create a message record for each SWIFT message that you will send and receive. You can also use network and usage validation rules in your maps to validate your messages. Use SWIFT message records to create inbound and outbound maps that define how particular messages are handled. For information about mapping, see Chapter 7, "Creating Inbound and Outbound Mapping Services".

During installation, SWIFT Module automatically installs the SWIFT message DFDs (including parsing templates) that are required to create a message record. These elements are located in the *Integration Server_directory*\packages\WmFIN\import directory. (For information about parsing templates, see Appendix B, "XML Parsing Templates for SWIFT FIN Messages"

Trading Networks assets are also required to create a message record for a SWIFT message, but must be imported manually as explained in the following procedure.

## Creating Message Records

### To run the wm.fin.dev:importFINItems service

1 In Designer, navigate to the WmFIN package and run the wm.fin.dev:importFINItems service. (For information about running services in Designer, see the Designer Service Development online help.)

2 Define the fields as specified in the following table:

| Field | Description |
|---|---|
| msgType | FIN message type, for example, `564`. |
| version | FIN version, for example, `nov10`. |

| Field | Description |
|---|---|
| format | The format of the generated blocks and fields for the input FIN message. Valid values:<br>■ TAG_BIZNAME (default). SWIFT message tag followed by the business name specified in the message DFD, for example, 23G_Function of the Message. This format provides the best balance between readability and performance (causing half of the performance penalty of BIZNAMEONLY because lookups are used only when receiving a message).<br>■ TAGONLY. SWIFT message tag only, for example, 23G:. This is the simplest format, provides the best performance, and is best-suited to those already familiar with SWIFT and specific messages.<br>■ BIZNAMEONLY. Business name specified in the message DFD only, for example, Function of the Message. This format carries the largest performance penalty.<br>■ XMLTAG. XML-compatible tag name, for example, F23G. This format lets you take advantage of the XML-specific services and functionality provided by Integration Server, such as pub.xml:documentToXMLString. This format cannot contain colons or tags that begin with a number. |
| subfieldFlag | Indicates whether the IS document type generated for the FIN message is parsed to the field or subfield level. Valid values are:<br>■ true. Parses to the subfield level. For inbound messages, SWIFT Module automatically removes the SWIFT delimiter (/) from between subfields. For outbound messages, SWIFT Module adds the SWIFT delimiter (/) between subfields.<br>■ false. Parse to the field level. |
| createDocType | Indicates whether to create and insert a TN document type, used to recognize an incoming message). Set to true. |
| createProcessing Rule | Indicates whether to create a Trading Networks processing rule that specifies steps to execute for message processing.<br>■ true (default). Create a processing rule.<br>■ false. Do not create a processing rule.<br>**Important!** If you are using processing rules to manage SWIFT FIN messages, you should always set this field to true. |

| Field | Description |
|---|---|
| createTPA | Set this field to `true` to create and insert a Trading Networks trading partner agreement (TPA) for this message. A TPA is a set of parameters that govern how SWIFT FIN messages are exchanged between two trading partners.<br>**Important!** You should always set this field to `true`. |

The wm.fin.dev:importFINItems service copies the SWIFT message parsing templates and DFDs from the import folder into the DFD folder of the WmFIN package and creates the following assets for each message:

- IS Document Type
- TN document type
- Trading Partner Agreement (in “Proposed” status)
- Processing rule to process an inbound SWIFT message
- Network validation rules (if available)
- Market Practice rules (if available)

**Note:** SAA sends SWIFT Acknowledgements (ACKs) and Negative Acknowledgements (NACKs) to SWIFT Module. DFDs for SWIFT ACKs and NACKs are automatically installed for you the first time you initialize Integration Server after installing SWIFT Module. The TN document types for ACKs and NACKs are added to the Trading Networks database if they are not already installed. The corresponding IS document types ACKs and NACKs are located in the wm.fin.doc.catF folder of the WmFIN package. The SWIFT templates for ACKs and NACKs are located in the directory: *Integration Server_directory*\ packages\WmFIN\config\dfd.IS document types are created in the WmFINMessages package in the appropriate version folder and message category (for example, wm.fin.doc.nov10.cat5).

## About Trading Partner Profiles

Trading partner profiles help define how you and your trading partners exchange SWIFT messages and files. TN document types enable Trading Networks to identify a business document and determine what information to extract from it.

A trading partner is any person or organization with whom you conduct business electronically. In SWIFT Module, a trading partner is defined by the criteria that you specify in a trading partner profile. This includes the company name and other identifying information, such as contact information and preferred delivery methods.

## Why Are Trading Partner Profiles Important?

Trading partner profiles, trading partner agreements (TPAs), and processing rules, together define how you and your trading partners exchange SWIFT messages. Processing rules define the actions your company takes in certain transactions, as well as the actions you expect your trading partners to perform during those transactions. In fact, the definition of profiles, the configuration of processing rules, and the application of TPAs are what enable you to interact successfully with your trading partners.

You must define a trading partner profile for each of your trading partners and, additionally, one for your own organization (MyEnterprise).

## Defining Trading Networks Profiles

This section explains how to define Trading Networks profiles.

### To define Trading Partner Profiles

1 Define your enterprise profile (My Enterprise) in Trading Networks by completing the following required fields:

| Required Profile Field for Enterprise | Description |
|---|---|
| Corporation Name | The name of your enterprise |
| External ID Type | BIC |
| External ID Type Value | Your enterprise's BIC |

**Note:** The BIC External ID Type is added to the Trading Networks database when you run the script to create a BIC table in the database.

2 Define a trading partner profile in Trading Networks for each trading partner with whom you exchange SWIFT messages and files, completing the following required fields:

| Required Profile Field for Trading Partner | Description |
|---|---|
| Corporation Name | The name of the trading partner |
| External ID Type | BIC |
| External ID Type Value | Your trading partner's BIC |

For additional instructions on defining trading partner profiles, see the Trading Networks administration guide for your release. See “About this Guide” for specific document titles.

# About TN Document Types for SWIFT Messages

TN document types are definitions that tell Trading Networks how to identify a type of business document and specify the attributes that Trading Networks should extract from the business document.

When you run the wm.fin.dev:importFINItems service (located in the WmFIN package) to create a record for a particular SWIFT message DFD, you can specify that the service also create the TN document types for this message.

When SWIFT Module receives a message, it invokes a Trading Networks service to recognize the type of business document by using the TN document types and the message records that you created. When Trading Networks matches the TN document type to the corresponding business document, Trading Networks extracts the specific pieces of information from the business document as indicated by the TN document type definition.

Define a TN document type to identify the format of the document generated by your back-end system that must be converted to a SWIFT message. When you create the TN document type, be sure to extract the *SenderID* and *ReceiverID* system attributes. (These values are the BICs for the sender and receiver.)

For instructions on defining TN document types, see the Trading Networks administration guide for your release. See “About this Guide” for specific document titles.

# 6 Creating Validation Rules

# Creating Validation Rules

Software AG provides network validation rules for a number of commonly used message types. In addition to these rules, SWIFT Module enables you to create network validation rules for additional messages as well as create usage validation rules.

**Note:** Although Market Practice rules function in much the same way as validation rules, the configuration of Market Practice rules differs from that of validation rules. For more information about Market Practice rules, see Chapter 14, "Working with Market Practices" on page 143.

## Creating Network Validation Rules

When SWIFT Module sends and receives a SWIFT message, it validates the message at either the individual field level or across the fields using network validation rules as specified by SWIFT. SWIFT Module sends a message only when the message structure, syntax, and validation rules are applied. Software AG provides network validation rules for version `may10`, as flow services, for you to use with the SWIFT FIN messages that you import. When you create a message record, the corresponding network rule (as a flow service) is imported into Integration Server and placed in the WmFINMessages package along with the message record.

After SWIFT Module syntactically validates a message, it executes the corresponding network rule. If you are using process models, any validation errors are aggregated and reported in Integration Server and Process Engine error logs.

Software AG provides network validation rules for the following SWIFT FIN messages:

- MT 101 Request for Transfer
- MT 103 Single Customer Credit Transfer
- MT 103STP Single Customer Credit Transfer
- MT 202 General Financial Institution Transfer
- MT 300 Foreign Exchange Confirmation
- MT 320 Fixed Loan/Deposit Confirmation
- MT 502 Order to Buy or Sell
- MT 515 Client Confirmation of Purchase or Sale
- MT 535 Statement of Holdings
- MT 536 Statement of Transactions
- MT 537 Statement of Pending Transactions
- MT 540 Receive Free
- MT 541 Receive Against Payment

- MT 542 Deliver Free
- MT 543 Deliver Against Payment
- MT 544 Receive Free Confirmation
- MT 545 Receive Against Payment Confirmation
- MT 546 Deliver Free Confirmation
- MT 547 Deliver Against Payment Confirmation
- MT 548 Settlement Status and Processing Advice
- MT 564 Corporate Action Notification
- MT 565 Corporate Action Instruction
- MT 566 Corporate Action Confirmation
- MT 567 Corporate Action Status and Processing Advice
- MT 568 Corporate Action Narrative
- MT 900 Confirmation of Debit
- MT 910 Confirmation of Credit
- MT 940 Customer Statement Message
- MT 942 Interim Transaction Report
- MT 950 Statement Message

You can create additional network validation rules for particular messages by writing individual services based on the SWIFT message documentation (pdf) provided by SWIFT. To use a new validation rule, you must specify the service you created in the *ValidationRule* parameter in the TPA for the particular SWIFT message.

For more information about TPAs, see Chapter 8, "Customizing Trading Partner Agreements" on page 93.

## Creating Usage Validation Rules

Usage rules exist only for certain messages when being exchanged between two specific partners. Software AG does not provide built-in usage rules because they vary by trading partner pairs, but you can create usage rules for particular messages by writing individual services based on the message documentation (.pdf) provided by SWIFT. To implement a usage rule, you must specify the service you created in the *UsageRule* parameter in the TPA for the particular SWIFT message.

For more information about TPAs, see Chapter 8, "Customizing Trading Partner Agreements" on page 93.

# 7 Creating Inbound and Outbound Mapping Services

# What Is Message “Mapping?”

Message mapping is the process of assigning structure, values, or content of one message to a different message; that is, mapping the matching values, data, and information between messages. The reason you want to map messages is because, typically, your back-end systems have different message formats than the format of a SWIFT message.

## Why Create an Outbound Mapping Service?

Create an outbound mapping service to translate an outbound back-end proprietary message format (for example, Oracle Financials) to SWIFT message format. Elements of the proprietary message must be mapped to corresponding elements of a SWIFT message (for example, MT 541, Receive Against Payment). Extra elements in the back-end message are ignored; however, you must map values to all elements in the SWIFT message.

Examples of outbound mapping services are located in the SWIFT Module samples. For more information about the SWIFT Module sample services, see *webMethods SWIFT Module Samples Guide*.

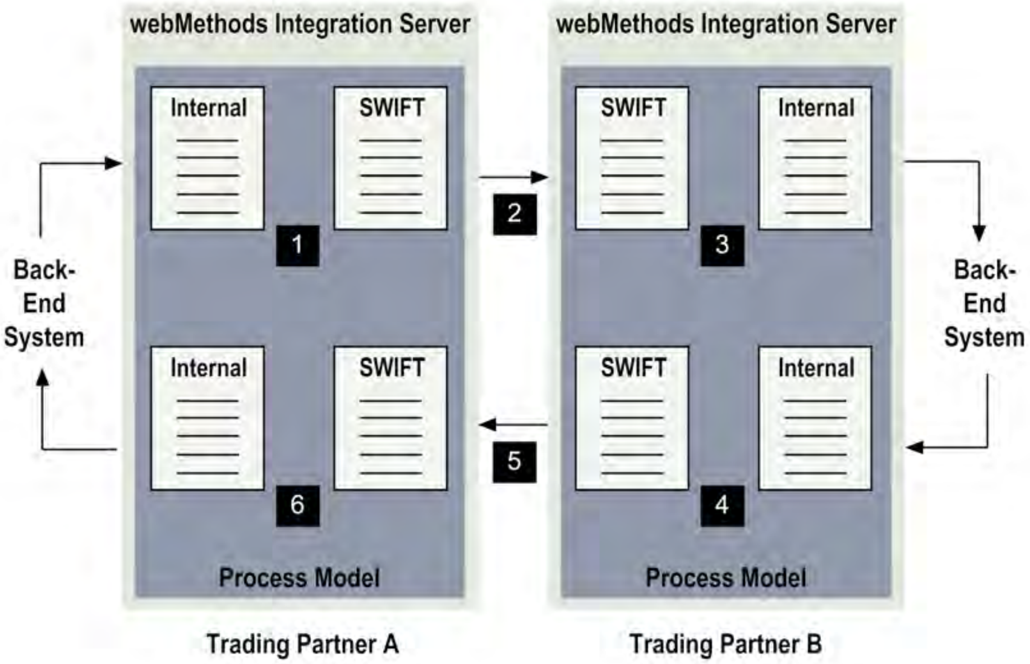
## Why Create an Inbound Mapping Service?

Create an inbound mapping service to map each element of an inbound SWIFT message to a corresponding element in your back-end proprietary message format. For example, if you use Oracle Financials and you want to receive a SWIFT message via SWIFT Module, you would create an inbound mapping service that maps each element of the SWIFT message to a corresponding element in the Oracle Financials format.

Examples of inbound mapping services are located in the SWIFT Module samples. For more information about the SWIFT Module sample services, see *webMethods SWIFT Module Samples Guide*.

## Example of Mapping a Message

The following figure illustrates the process of mapping a message. For further explanation, see the text that follows the figure.



| Step | Description |
|---|---|
| 1 | Trading Partner A uses an outbound mapping service to map an internal message from a back-end format to SWIFT format. |
| 2 | Trading Partner A sends the SWIFT message to Trading Partner B. |
| 3 | Trading Partner B receives the SWIFT message and uses an inbound mapping service to map the SWIFT message to an internal message. After the internal message is mapped, it is in a format that Trading Partner B's back-end system can process. |
| 4 | Trading Partner B responds by using an outbound mapping service to map an internal message from a back-end format to SWIFT format. |
| 5 | Trading Partner B sends the SWIFT message to Trading Partner A. |
| 6 | Trading Partner A receives the SWIFT message and uses an inbound mapping service to map the SWIFT message to an internal message. After the SWIFT message is mapped, it is in a format that Trading Partner A's back-end system can process. |

## Creating an Outbound Mapping Service

Use Designer to create an outbound mapping service. The service must contain one or more MAP entities that map the message from the format of your back-end system, through any desired intermediate steps, to the IS document type for the appropriate outbound SWIFT message.

## Inputs and Outputs

The input to the outbound mapping service is your back-end system document provided as the TN document type in the variable "*bizEnv*."

**Note:** For information about retrieving your back-end document from the *bizEnv* variable, see the outbound mapping services in the SWIFT Module samples, described in the *webMethods SWIFT Module Samples Guide*.

The outbound mapping service generates the output as the SWIFT message in the *documents\payload* IData object in the pipeline. For example, if you are mapping your back-end message to MT 564, the output of the mapping service would be `MT564` in the *documents\payload* IData object. See the figure in "Example of Mapping a Message" on page 88 for more information.

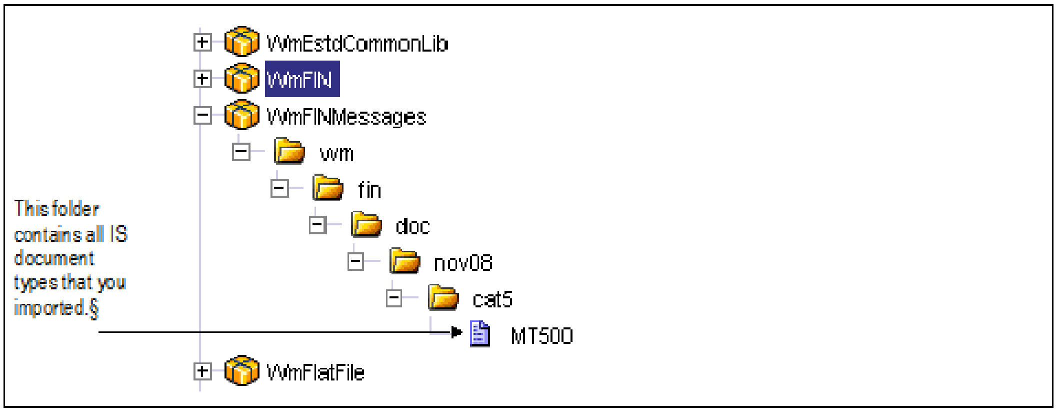
The output of the outbound mapping service must be placed in the documents\payload IData object. SWIFT Module converts an IData object into an XML string before sending the SWIFT message to the trading partner, and therefore must know the precise location of the IData object.

You must place the Agreement ID for this TPA in the *documents\tpaAgreementID* IData object. For more information about Agreement IDs, see Chapter 8, "Customizing Trading Partner Agreements"on page 93.

**Note:** If the documents for your back-end system have DTDs, you can automatically import an external DTD in Designer to provide a starting point for mapping. To do so, create an external record and specify the source as XML, DTD, or XML Schema.

## Flow Operations to Use

In the flow service, insert a MAP operation and use the service pipeline to map elements of the IS document type from your back-end message to all elements of the IS document type for the appropriate SWIFT message. Built-in IS document types for all versions of SWIFT FIN messages that you imported are located in the WmFINMessages package in wm\fin\doc\*version**category* folders as illustrated in the following figure.



This folder contains all IS document types that you imported.§

## Creating an Inbound Mapping Service

In Designer, just as with outbound mapping services, create a new inbound mapping service that contains one or more MAP entities. The MAP entities perform the actual mapping from the received SWIFT FIN messages through any intermediate steps, to the format of your back-end system messages. The inputs to any inbound mapping service include the following variables:

- *finIData*. The message in TAG format.
- *convertedFinIData*. The format specified in the message TPA (for example, `TAG_BIZNAME`).

## Parsing to the Subfield Level

You can configure SWIFT Module to parse to the subfield level in your outbound and inbound mapping services. To do so, set the *subfieldFlag* variable to `true` when invoking the following services:

- wm.fin.dev:importFINItems
- wm.fin.dfd:convertTagFormat
- wm.fin.dfd:convertBizNameFormat

For information about these services, see Appendix A, “Services” on page 179.

For inbound messages, if you want to manually parse messages to the subfield level (without using the subfield option), you must manually remove the SWIFT delimiter (/) from between subfields. For outbound messages, if you want to manually identify subfields (without using the subfield option), you must add the SWIFT delimiter.

Note: The DFDs files do not define subfields for some compound fields in the following directories: *Integration Server_directory*\packages\WmFIN\config\dfd\version and *Integration Server_directory*\packages\WmFIN\import\version. If you require subfields for a tag where they are not provided, you can edit the DFD files. Follow the SWIFT documentation when adding subfields. The DFD files located in the WmFIN\import\*version* directory take precedence over the DFD files located in the WmFIN\config\dfd\*version* directory.

## Reusing Mapping Services

In SWIFT Module, you can reuse mapping services for trading partners that submit the same business document format. For example, you can use the same mapping services for Trading Partner A and Trading Partner B if they both always submit business documents in the same document format to SWIFT Module. As the receiver of those documents, you need to define only one inbound mapping service for both trading partners because the message format versions are the same.

# 8 Customizing Trading Partner Agreements

## Overview

A Trading Partner Agreement (TPA) is a set of parameters that you use to govern how SWIFT FIN messages are exchanged between two trading partners. TPAs also enable you to define Market Practice requirements for individual markets. Using TPAs, SWIFT Module supports customization of SWIFT FIN messages based on specific trading partner pairs.

You can modify and view TPAs in the Agreement Details screen in My webMethods.

For detailed information about working with TPAs, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

## How Does SWIFT Module Identify a TPA?

Every SWIFT message in SWIFT Module is associated with a TPA. Every TPA is uniquely identified by a Sender, Receiver, and Agreement ID. During a business process between trading partners, SWIFT Module uses this information to retrieve the TPA for a specific sender-receiver pair and to process the messages exchanged.

After SWIFT Module recognizes and associates an incoming SWIFT message with a particular TN document type, it uses the TN document type name in the business document (such as "MT541,") to find a TPA matching sender-receiver pair.

When SWIFT Module creates a message record, it automatically creates the TPA for the particular SWIFT message. You can view and modify the selected "Proposed" TPA in Trading Networks.

## Modifying the TPA

This section describes how to modify a TPA to send and receive messages in SWIFT Module.

### To modify Trading Partner Agreements

1 In the Agreement Details screen, define the parameters as described in the table below:

| TPA Information | Description |
|---|---|
| Sender | The name of the sending trading partner or "Unknown.". Select the sender from the profiles defined in Trading Networks, including your own profile.<br>If you specify a partner for sender, you must also specify a partner for receiver. Likewise, if you specify Unknown for sender, you must specify Unknown for receiver. |

| TPA Information | Description |
|---|---|
| Receiver | The name of the trading partner that receives the message from SWIFT. Select the receiver from the profiles defined in Trading Networks, including your own profile. |
| Agreement ID | Uniquely identifies the agreement between two partners, for example, `MT541`. This is a placeholder for the TN document type name (for example, "MT541") |
| IS Document Type | The IS document type wm.fin.doc:UserParameters (located in the WmFIN package) specifies the SWIFT TPA parameters. |
| Agreement Status | Set to "Agreed." To process messages according to a TPA, both sender and receiver TPAs must have the Agreement Status, "Agreed." |

**Note:** The **Sender** and **Receiver** fields initially display a default value of "Unknown." The TPA identifies the business names of the sender and receiver, however, SWIFT Module identifies the sender and receiver using their BIC.

2 Complete the following additional SWIFT-specific TPA input parameters.

| TPA Section | Parameter | Description |
|---|---|---|
| FIN Process Info | *Transport* | The SWIFT interface used to send and receive SWIFT FIN messages. Valid values are:<br>■ `MQ` (Default). WebSphere MQ Adapter.<br>■ `CASmf`. CASmf interface.<br>■ `AFT`. Automated File Transfer interface.<br>■ `TEST`. Tests the processing of a FIN message without sending the message to SWIFT. |
| | *MessageType* | The SWIFT FIN message type identifier, such as 541, Receive Against Payment. |
| | *ISDocument Name* | IS document type for this message. IS document types for each message are located in the WmFINMessages package, for example, wm.fin.doc.nov10. cat5:MT541. |
| | *Version* | Version of the SWIFT message, for example `nov10`. |

| TPA Section | Parameter | Description |
|---|---|---|
| | *Message Format* | The format of the generated blocks and fields for the input FIN message. Valid values are:<br>■ `TAG_BIZNAME` (default). SWIFT Message tag followed by the business name specified in the message DFD, for example, `23G_Function of the Message`.<br>■ `TAGONLY`. SWIFT Message tag only, for example, `23G:`.<br>■ `BIZNAMEONLY`. Business name specified in the message DFD, for example, `Function of the Message`.<br>■ `XMLTAG`. XML-compatible tag name. This format cannot contain colons or tags that begin with a number, for example, `F23G`. |
| | *SubfieldFlag* | Indicates whether to parse SWIFT messages to the field or subfield level. Valid values are:<br>■ `true` (default). Parse to subfield level.<br>■ `false`. Parse to field level.<br>**Note:** If you change this setting after creating message records, you must delete the message records pertaining to the TPA and recreate them. |
| | *Inbound ProcessingRule Service* | Optional. Indicates whether to use a processing rule to manage message execution. To do so, type the service name to use. For more information, see "Receiving Messages from SWIFT" on page 103. |
| | *Validate Content* | Optional. Indicates whether to validate the message. Valid values are `Yes` and `No`. |
| | *ValidateBIC Plus* | Optional. Indicates whether to validate BIC information in the message. Valid values are `Yes` and `No`. |
| | *Validate NetworkRules* | Optional. Indicates whether to validate the message against network rules. Valid values are `Yes` and `No`. |
| | *Network ValidationServi ce* | Optional. Name of the network validation service to use to validate network rules. |

| TPA Section | Parameter | Description |
|---|---|---|
| | *ValidateMarket PracticeRules* | Optional. Indicates whether to validate the message against market practice rules. Valid values are `Yes` and `No`. |
| | *MarketPractice RulesService* | Optional. Name of the market practice rules validation service to use. |
| | *ValidateUsage Rules* | Optional. Indicates whether to validate this message against usage rules. Valid values are `Yes` and `No`. |
| | *UsageRules Service* | Optional. Name of the usage validation service to validate the usage rules. |
| Message Header | This section contains information to be populated in blocks {B1}, {B2}, and {B3} of the outbound SWIFT message. | |
| | *Logical Terminal* | The logical terminal identifier defined in SAA. |
| | *Application Identifier* | Identifies the application within which the message is sent or received. Valid values are:<br>- `F. SWIFT FIN`. All FIN user-to-user, FIN system, and FIN service messages.<br>- `A. GPA`. GPA system and service messages.<br>- `L. GPA`. Certain GPA service messages (for example, LOGIN, LAK, ABORT). |
| | *Service Identifier* | Identifies the type of data being sent or received:<br>- `01`—User-to-user messages.<br>- `21`—Message acknowledgements. |
| | *Priority* | Indicates the priority with which to deliver the message to the receiver. Valid values are:<br>- `N`—Deliver with normal priority.<br>- `U`—Deliver with urgent priority.<br>- `S`—Deliver with system priority. |

| TPA Section | Parameter | Description |
|---|---|---|
| | *Delivery Monitoring* | Optional. Enables the sender to request delivery monitoring. Valid values are:<br>■ `None`—Do not perform delivery monitoring.<br>■ `1`—Non-delivery warning. Requests automatic MT010 warning if message is not delivered within the obsolescence period, (15 minutes for `U Priority`, 100 minutes for `N Priority`).<br>■ `2`—Delivery notification. Requests automatic MT011 notification after message is delivered.<br>■ `3`—Both. Requests both automatic non-delivery warning and delivery notification. |
| | *FINCopy Service Identifier* | Optional. (Used with FINCopy messages only.) Three-character system ID used to support access to multiple services with the same CBT. |
| | *Banking Priority* | Optional. Four-character field agreed upon by two or more parties to indicate priority. |
| | *ValidationFlag* | Optional. Indicates whether special validation must be completed at SWIFT. For information about the MT message standards, see the SWIFT User Handbook/Standards. |
| | *Addressee Information* | Optional. Information from the central institution to the receiver of the payment message. This information is used in the input of MT097, FIN Copy Message Authorization Refusal Notification. |
| | *Training* | Indicates whether a message is sent to or received from a test and training logical terminal. |
| MQSeries Info | *putMessage HandlerService* | The name of the service generated when creating a message handler service for IS-to-WebSphere MQ transport. For more information, see "Using WebSphere MQ Adapter to Communicate with SWIFT" on page 110. |
| AFT | *folder* | Fully qualified path of the folder you want the file polling listener to poll for this message, for example, c:\folder\bic.dat.<br>**Important!** This folder must be accessible by both Integration Server and SAA. |
| | *extension* | Optional. File extension of the files to be received from the AFT folder. The default is `inp`. |

For complete instructions on modifying TPAs, see the Trading Networks administration guide for your release. See “About this Guide” for specific document titles..

# 9 Configuring Processing Rules to Send and Receive SWIFT FIN Messages

# Overview

SWIFT Module enables you to use customized Trading Networks processing rules to send and receive SWIFT messages. This chapter explains how to use custom-created rules to send and receive SWIFT messages.

# Sending Messages to SWIFT

You can send messages to SWIFT using a service that is invoked by a Trading Networks processing rule.

## Preliminary Steps for Sending Messages

Before you can configure Trading Networks processing rules to send outbound SWIFT messages, you must do the following:

- Create Trading Partner Profiles, as described in Chapter 5, "Defining Trading Networks Information".
- Modify Trading Partner Agreements for each of your trading partners, as described in Chapter 8, "Customizing Trading Partner Agreements".

## Assigning the Processing Rule

This section describes how to assign a processing rule to manage SWIFT FIN messages. (For instructions on how to define processing rules, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

There are four steps to configuring a Trading Networks processing rule to send SWIFT FIN messages:

- Define the criteria of the processing rule.
- Define the processing action.
- Create a service to map the document generated by the back-end system to a webMethods DFD format.
- Invoke the service to submit the document to Trading Networks.

### Step 1: Define the Processing Rule Criteria

Define the criteria of the processing rule on the **Criteria** tab as follows:

| Criteria tab field | Set to... |
|---|---|
| Document Type | The TN document type for the format of the back-end document. |

### Step 2: Define the Processing Action

Define the processing action on the Action tab. Click Add Action and define the fields as follows:

| Action drop-down list | Set to... |
|---|---|
| Execute a service | The service you created to map the back-end document to webMethods DFD format. For more information on creating this service, see "Step 3: Create a Service to Map to the DFD Format" on page 103.<br>**Important!** You must select `synchronous` to invoke this service synchronously. |
| Deliver Document By | `Immediate Delivery` and select FINTransport<br>The FINTransport delivery service uses variables specified in the TPA to govern the creation and sending of the outbound SWIFT message. |

### Step 3: Create a Service to Map to the DFD Format

Create a service to map the back-end document to webMethods DFD format using the following logic:

1. Retrieve the back-end system document content from the BizDocEnvelope and generate an IData object by invoking the wm.tn.doc.xml:bizdocToRecord service.
2. Map data from the back-end system document to webMethods DFD format for the {B4} block of the SWIFT FIN message.
3. Convert the webMethods DFD format IData object to an XML String by invoking the pub.xml:documentToXMLString service.
4. Convert the XML String to bytes by invoking the pub.string:stringToBytes service.
5. Invoke the wm.tn.doc:addContentPart service to add the bytes to the BizDocEnvelope as a new content part. When you add the content part, you must name the content part `DFD Data`.

### Step 4: Submit the Document to Trading Networks

At run time, submit the back-end system document to Trading Networks by invoking the wm.tn:receive service.

## Receiving Messages from SWIFT

You can process inbound messages from SWIFT using a Trading Networks processing rule to validate and process the inbound message.

## Preliminary Steps for Receiving Messages

Before you can configure Trading Networks processing rules to receive SWIFT FIN messages, you must do the following:

- Create Trading Partner Profiles, as described in Chapter 5, "Defining Trading Networks Information".
- Modify Trading Partner Agreements for each of your trading partners as needed, as described in Chapter 8, "Customizing Trading Partner Agreements".

## Defining the Processing Rule

### To create a service to map the webMethods DFD format to the back-end system format

1. Create a service to map the webMethods DFD format to the format of the back-end system using the following variables in the pipeline:
   - *finIData*—The message in TAG format.
   - *convertedFinIData*—The format specified in the message TPA (for example, `TAG_BIZNAME`).
2. Specify the name of this service in the message TPA *InboundProcessingRuleService* parameter.
3. For further instructions on defining processing rules, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

### Inbound Message Processing

When a SWIFT message is received by the wm.fin.trp:receiveMessage service, Integration Server does the following automatically:

- Recognizes the SWIFT message using TN document types.
- Forms a BizDocEnvelope for the SWIFT message.
- Saves the content of the SWIFT message to the Trading Networks database.
- Submits the SWIFT message to the Trading Networks processing rules engine, which validates the message. If validation succeeds, the output of the validation includes the *finIData* and *convertedFinIData* variables. Trading Networks then invokes the service specified in the message TPA *InboundProcessingRuleService* parameter.

# 10 Using SWIFT Module SDK Services

## What Is the SWIFT SDK

The SWIFT Standards Developer Kit (SDK) is a library of tools that includes automated updates to XSDs and IS document types as new message standards are released annually. The library includes MT XML schema definitions (XSDs) and Java services for performing MT message conversion. The XSDs provide the ability to enforce MT standards by capitalizing on the benefits of XML, including defining and restricting the message elements, constraining field data values to SWIFT standards, and automatic validation of the message format and field data.

SWIFT Module's integrated SDK tools provide several benefits including:

- The use of one technology for sending and receiving both MT and MX messages.
- XML integration between applications, while using the MT (FIN) format between partners.
- Fewer errors caused by incorrect FIN syntax.
- Automated annual updates to MT message standard changes.

## About the SWIFT Module SDK Features

SWIFT Module integrates the SDK tools, enabling automated update of the XSDs and IS document types provided with SWIFT Module as new message standards are released. In addition, SWIFT Module capitalizes on XSD support by providing services to convert between flat file and XML format, including the flexibility to convert entire messages or just the block 4 portion of the message.

The following services convert between XML and flat file format and perform message validation:

- wm.sdk.fin.converter:convertMTBlock4ToMTXML—This service converts only block 4 of the MT flat file into XML format.
- wm.sdk.fin.converter:convertMTFlatFileToMTXML—This service converts the entire MT flat file into XML format.
- wm.sdk.fin.converter:convertMTXMLblock4ToMTFlatFile—This service converts only block 4 of the XML file into MT flat file format.
- wm.sdk.fin.converter:convertMTXMLToMTFlatFile—This service converts the entire XML file into MT flat file format.
- wm.sdk.fin.validator:validateMTXML—This service validates any MT XML message against the SWIFT SDK MT schema.

To use these services, you must map them in Designer. For more information about using Designer, see the Designer online help for your release. See “About this Guide” for specific document titles. SWIFT Module automatically performs file conversion based on the corresponding XSD or matching IS document type.

For information about these services, see "Using SWIFT Module SDK Services" on page 105.

## SWIFT Module SDK Document Formats

As part of the SDK feature, SWIFT Module provides pre-bundled SDK MT and MX XSDs for the creation of MT and MX IS document types. SWIFT Module also provides services that automatically create the back-end IS document types and IS schemas from the MT and MX schemas.

The following IS and XSD documents are included as part of the SDK feature:

- IS document types—The creation services generate a separate IS document type for each message type, per SDK version (provided as part of the SWIFT FIN component). The conversion services use these document types when converting an XML message into flat file format.
- MT and MX XSDs—The following XSDs are pre-bundled within SWIFT Module:
  - Block 4 XSDs—A separate XSD to define the business content (block 4) of a message for each message type, per SDK version. These are used when converting flat file messages into XML format.
  - File Definition XSDs —A separate XSD that defines the header and trailer elements common to all message types (Blocks 1, 2, 3, and 5). This is used to enforce the message standard format during conversion of the (entire) file. There is a separate XSD for each SDK version.

### To install SDK document formats

To install the IS document types and IS schemas, complete the following steps:

1. Execute the wm.sdk.docgenerator:createMTISDocFromSchema service to create IS document types and IS schemas for MT FIN messages.
2. Execute the wm.sdk.docgenerator:createMXISDocFromSchema service to create IS document types and IS schemas for MX FIN messages.

The corresponding MT and MX IS document types and IS schemas are created in the wm.sdk.rec folder inside the WmFIN package.

## SWIFT Module SDK Folder Organization

The supporting SDK documents and services are organized using the following folder structure:

- wm.sdk.fin—Contains all the Java services related to the SDK feature.
- wm.sdk.rec.mtxsd.Vyear—Placeholder for the IS document types created from MT XSDs that correspond to the supported MT XML message version.

- wm.sdk.rec.mxxsd—Placeholder for all the IS document types created from MX (ISO20022) XSDs that correspond to the supported MX XML message version.
- *Software AG_directory*\Integration Server\packages\WmFIN\pub\resources\SDK_MT_XML_Schema_Library—Contains the pre-bundled MT XSDs.
- *Software AG_directory*\Integration Server\packages\WmFIN\pub\resources\SDK_MT_XML_Schema_Library—Contains the pre-bundled XSDs.

For information about using SWIFT Module SDK samples, see *webMethods SWIFT Module Samples Guide*.

# 11 Configuring SWIFT Interfaces

# Overview

The SWIFT FIN component of SWIFT Module provides integration support for the following SWIFT Alliance Access (SAA) interfaces that you can use to send and receive messages to SWIFT:

- MQHA. If you are connecting to SAA through MQHA, you must install the webMethods WebSphere MQ Adapter on Integration Server. For more information about configuring and using the adapter with the SWIFT FIN component, see "Using WebSphere MQ Adapter to Communicate with SWIFT" on page 110.
- CASmf. If you are connecting to SAA through CASmf, you must install the WmFIN package that contains the WmCASmf services on Integration Server. For more information about configuring and using CASmf with the SWIFT FIN component, see "Using the CASmf Services to Communicate with SWIFT" on page 112.
- AFT. If you are using Automated File Transfer (AFT), you must configure the File Polling Listener and AFT settings in the message TPA. For more information about configuring and using the File Polling Listener with the SWIFT FIN component as well as using File Drop for outbound messages, see "Using AFT to Communicate with SWIFT" on page 116.

# Using WebSphere MQ Adapter to Communicate with SWIFT

The WebSphere MQ Adapter enables Integration Server to exchange information with other systems through an IBM WebSphere MQ message queue. This capability lets you route documents or any piece of information from Integration Server to systems that use WebSphere MQ message queuing as their information interface. The WebSphere MQ Adapter enables you to connect to SAA through MQHA. For detailed information about using the WebSphere MQ Adapter, see *webMethods WebSphere MQ Adapter Installation and User's Guide*.

**Important!** The following procedure assumes that you have already configured your WebSphere MQ system and SAA to communicate with one another, and have installed the WebSphere MQ Adapter. For more information about installing the WebSphere MQ Adapter, see *webMethods WebSphere MQ Adapter Installation and User's Guide*.

## Configuring the WebSphere MQ Adapter

### To configure the WebSphere MQ Adapter for the SWIFT FIN component

1 Configure WebSphere MQ Adapter connections. You will need to configure at least two connections:

   - One connection to send messages to MQ Series.
   - One connection to receive messages from MQ Series.

For information about configuring WebSphere MQ Adapter connections, see*webMethods WebSphere MQ Adapter Installation and User's Guide*.

2 Configure settings to deliver outbound messages from the SWIFT FIN component to SWIFT via WebSphere MQ Adapter.

a Configure a WebSphere MQ Adapter Put service to deliver SWIFT messages to a MQ Series queue. Use the connection that you configured in step 1 for sending messages to MQ Series.

For information about configuring the WebSphere MQ Adapter Put service, see *webMethods WebSphere MQ Adapter Installation and User's Guide*.

b Create a flow service or Java service that invokes the Put service you just configured in step a.

When you create the service:

- The input variables must include the variable *msgBody* with data type byte[] (or Object for a flow service).
- Before invoking the Put service that you configured in step a, the logic of the service must map the data in the *msgBody* input variable to the input of the Put service; that is, map the value to the *putServiceInput/msgBody* variable of the generated Put adapter service.
- The logic must invoke the Put service that you configured in step 2a.

c Update the TPA for SWIFT messages to identify the correct method to deliver outbound SWIFT messages. To do so, update the following in the TPA:

| In this section of the TPA... | Set this parameter... | To... |
|---|---|---|
| FINProcessInfo | *Transport* | MQ |
| MQSeriesInfo | *putMessageHandlerService* | The name of the service that you created in step 2b. |

For more information about TPAs, see Chapter 8, "Customizing Trading Partner Agreements".

3 Define configurations for receiving inbound messages from SWIFT via WebSphere MQ Adapter.

a Configure WebSphere MQ Adapter listeners and listener notifications. When configuring the listener, identify the connection that receives messages from MQ Series that you configured in step 1.

When you create a listener notification, you must specify a service to invoke when WebSphere MQ Adapter retrieves a message from MQ Series. For more information about this service, see the next step.

For information about configuring the WebSphere MQ Adapter listeners and listener notifications, see *webMethods WebSphere MQ Adapter Installation and User's Guide*.

b Create a flow service or Java service that is invoked when the WebSphere MQ Adapter retrieves a message from MQ Series. The last action that the flow or Java service takes must be to invoke the wm.fin.transport.MQSeries:getMQSeriesListenerService service. The service getListenerService processes the received FIN message. For more information about this service, see wm.fin.transport.MQSeries:getMQSeriesListenerService.

How the service is invoked depends upon the type of listener notification you created in step a:

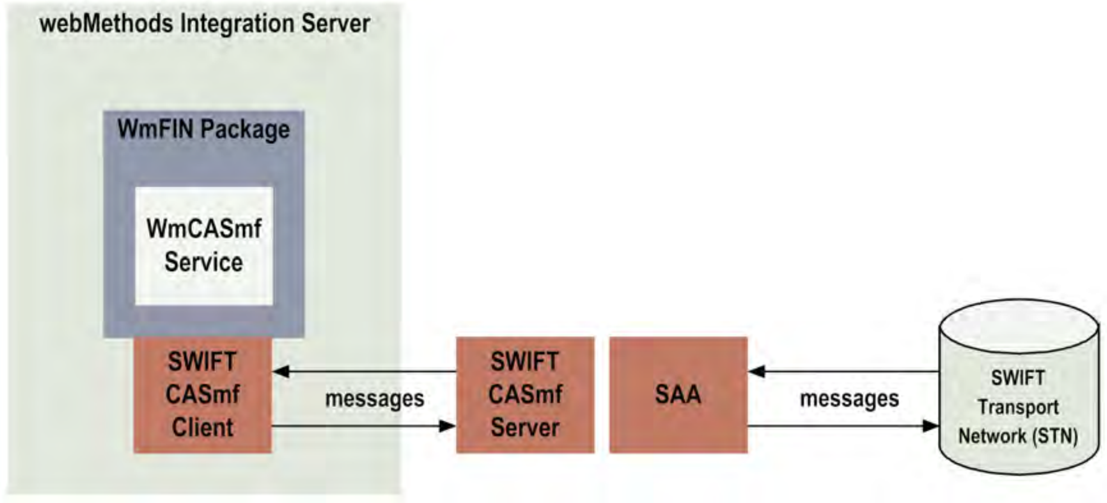
- If you create an asynchronous listener notification, you must create a trigger that subscribes to the notification. When you create the trigger, set it up to invoke this service.
- If you create a synchronous listener notification, you must specify the name of this service when you configure the synchronous listener notification.

# Using the CASmf Services to Communicate with SWIFT

The SWIFT Module WmFIN package contains the CASmf services that enable Integration Server to send and receive messages through the CASmf interface. The WmFIN package also contains services that enable you to connect to SAA, and to route messages from Integration Server to SAA over the CASmf interface.

## webMethods CASmf Services

The webMethods CASmf services are part of the WmFIN package and the following diagram illustrates how the webMethods CASmf services interact with the other webMethods product suite components.



The following table describes the components that interact with the webMethods CASmf services.

| Component | Description |
|---|---|
| WmFIN package | The package that contains the webMethods CASmf services. |
| WmCASmf service | The service that transfers SWIFT FIN messages to and from the SAA system using CASmf. |
| CASmf Client | The client enables the CASmf services to interface with the CASmf server. |
| CASmf Server | The server enables communication between the CASmf client and SAA. |
| SAA | SWIFT software configured to access the SWIFT Transport Network (STN), SWIFT's original network accessed using x.25 transport technologies. |
| SWIFT Transport Network (STN) | The existing SWIFT interface, a computer system provided and operated by the user, which enables communication with the SWIFT network. |

## Configuring the CASmf Interface

Important! The following procedure assumes that you have already configured your CASmf server and SAA to communicate with one another. For more information, see your CASmf server and SWIFT documentation.

The configuration settings for the CASmf services are stored in the CASmf configuration file (wmcasmf.cnf). This file resides in the*Integration Server_directory* \packages\WmFIN\config directory and contains parameters that determine how the services operate. Edit the file directly with a text editor.

### To configure the CASmf Interface for SWIFT Module

1 Install a CASmf client on the same machine as Integration Server. For more information, see your CASmf documentation.

Important! Start the SWIFT CASmf Client before starting Integration Server.

2 Configure the CASmf services to work with SWIFT Module:

a Open the *Integration Server_directory*\packages\WmFIN\config\wmcasmf.cnf file in a text editor.

b Edit the following properties as needed:

| Property | Description |
|---|---|
| wm.casmf.send.mapid | The sending and receiving mapIDs that you have defined for SAA. The default value is `CASmfInput`. |
| wm.casmf.receive.mapid | The mapIDs must match exactly the two `l_mapid` fields in the CASmf client dmapid.dat file. Usually, this file is located in the directory where the CASmf client is installed: *$CASmfInstallationFolder*\dat. You can also locate this file using the folder listed in your *DATTOP* environment variable. The default value is `CASmfOutput`. |
| wm.casmf.send.message.folder | The default folder in which all outbound SWIFT FIN messages are queued before being sent to SWIFT via the CASmf Interface: *Integration Server_directory*\packages\WmFIN\config\outboundMessages. You can change this location if desired. |
| wm.casmf.authentication.type | The type of authentication that you want the webMethods CASmf services to perform with SAA. Specify one of the following:<br>■ `AUTH_ACCESS`—Performs session authentication.<br>■ `AUTH_DATA`—Performs data authentication.<br>■ `AUTH_BOTH`-Performs session and data authentication.<br>■ `AUTH_NONE`—Performs no authentication. |
| wm.casmf.authentication.sendKey | The keys used for authentication of the session:<br>■ *sendKey*—The receive key that you defined for the CASmf input message partner on SAA.<br>■ *receiveKey*—The send key that you defined for the CASmf output message partner on SAA.<br>**Important!** Reverse the keys appropriately when defining this property. For example, use the value you defined for the *sendKey* in SAA for the wm.casmf.authentication.receiveKey property. |
| wm.casmf.authentication.receiveKey | |

| Property | Description |
|---|---|
| wm.casmf.authentication.local Send Key | The keys used for local authentication of the data sent and received over a CASmf session on SAA:<br>■ *localSendKey*—The receive key that you defined for the CASmf input message partner.<br>■ *localReceiveKey*—The send key that you defined for the CASmf output message partner.<br>**Important!** Reverse the keys appropriately when defining this property. For example, use the value you defined for the *localSendKey* in SAA for the wm.casmf.authentication.local.ReceiveKey property. |
| wm.casmf.authentication.local ReceiveKey | |
| wm.casmf.receive.timeout | The number of seconds that CASmf services should maintain an active connection with SAA for receiving messages. If no messages are received within the specified time, the connection is closed. The default is 300 seconds.<br>For example, if the timeout value is 300 seconds and there are 10 messages that take 20 seconds to receive, the connection remains open for the 20 seconds it takes to receive the 10 messages, then remain idles for the next 280 seconds before being closed. |

Save and close the file.

3 In the TPA for each type of SWIFT message that you will send and receive using CASmf, set the following field:

| Set this field to... | To... |
|---|---|
| Transport | CASmf |

For more information about TPAs, see "Modifying the TPA" on page 94.

4 In the Integration Server Administrator, create a scheduling service to run the wm.casmf.trp:casmfSendReceiveSchedule service at intervals:

a In the Server menuof the Navigation panel, click **Scheduler**.

b Click **Create a scheduled task**.

c To set the fields in the Service Information section, follow the instructions in Integration Server administration guide for your release. See "About this Guide" for specific document titles.

d In the Schedule Type and Details section, under **Repeating Tasks With a Simple Interval**, set the fields as follows:

- Select **Repeating**.
- In the **Start Date** and **Start Time** fields, enter the date and time of the first execution of the service. These fields are optional.
- In the **End Date** and **End Time** fields, enter the date and time of the last execution of the service. These fields are optional.
- Select the **Repeat after completion** check box.
- In the **Interval** field, Software AG recommends that you set this service to run at intervals of at least 15-20 minutes.

e Click **Save Tasks**.

For more information about this service, see wm.casmf.trp:casmfSendReceiveSchedule.

# Using AFT to Communicate with SWIFT

**Important!** To use Automated File Transfer (AFT), you must have the WmFlatFile package installed. This package is installed by default with Integration Server.

AFT enables Integration Server to exchange information with other systems. If you are using AFT to receive inbound SWIFT FIN messages through the File Polling Listener, and File Drop capabilities to send outbound SWIFT FIN messages, you must properly configure the File Polling Listener and the message TPA.

## Configuring AFT for Inbound Messages

### To configure the webMethods File Polling Listener for SWIFT Module

1. In the Integration Server Administrator, click **Security > Ports > Add Port**.
2. Select **webMethods/FilePolling** and click **Submit**.
3. Configure the File Polling Listener's general fields as described in the “Configuring Ports” section of the “Configuring the Server” chapter of the Integration Server administration guide for your release. See “About this Guide” for specific document titles.
4. Define the following fields so that the File Polling Listener can properly handle SWIFT FIN messages.

| Set this field... | To... |
|---|---|
| Content Type | application/x-wmflatfile |

| Set this field... | To... |
|---|---|
| Folder location | The fully qualified path of the folder from which SAA will send SWIFT FIN messages. The folder must be accessible to both SAA and Integration Server. |
| Processing Service | wm.fin.transport.AFT:processIncomingFile |

## Configuring AFT for Outbound Messages

Complete the following procedure to configure AFT using File Drop capabilities to send outbound SWIFT FIN messages using SWIFT Module.

### To configure File Drop for SWIFT Module

1. Map a network directory in which you want to drop files for SAA. This can be any directory in Integration Server.
2. In the TPA for the SWIFT message, set the following parameters:

| Set this field... | To... |
|---|---|
| Transport | `AFT` |
| Folder location | The fully qualified path of the folder in which Integration Server will drop SWIFT FIN messages. The folder must be accessible to both SAA and Integration Server. |
| FileExtension | `.inp` |

3. For more information about TPAs, see "Modifying the TPA" on page 94.

# 12 Configuring Notifications for Messages in XML v2 Format

# Overview

SWIFT Module handles all levels of notification that SWIFT Alliance Access (SAA) sends in response to requests from SWIFT Module:

- Information notification: When SAA receives MT messages from SWIFT Module, SAA validates the structure and the security signature of the message against SWIFT standards. If the message fails validation, SAA routes the message to a specific routing point and sends an information notification and message status to SWIFT Module.
- Transmission notification: SWIFT Central Services validates messages for FIN, InterAct, and FileAct services, and returns either a Positive (ACK) or Negative (NAK) acknowledgement. If SWIFT accepts the message, it returns ACK. If the message validation fails, SWIFT rejects the message and returns NAK. SAA maps these acknowledgements into transmission notifications and sends them to SWIFT Module.
- Delivery notification: When the message is delivered to the receiver, SWIFT Central Services sends a Delivery Notification (DeIN) acknowledgement. SAA maps DeIN into a delivery report or delivery notification and sends it to SWIFT Module.

SWIFT Module handles notifications using site-specific processing, configured through Trading Networks processing rules. SWIFT Module provides you with the TN documents required to recognize the notification messages. Trading Networks saves incoming notifications and associates them with the original messages using the corresponding processing rule. You can see related messages by viewing the documents in Trading Networks.

Important! SWIFT Module handles notifications only for messages exchanged over SAA in XML v2 format.

# Configuring SWIFT Module to Handle Notifications

## Step 1: Import Trading Networks Information for Notifications

To enable SWIFT Module to handle notifications received from SAA, import the following Trading Networks assets:

| Trading Networks Asset | Attribute | Description |
|---|---|---|
| Processing Rules | HandleDeliveryNotifications | Processes delivery notifications from SAA and relates them to the original document using the 108: tag of the message. |
| | Related SWIFT documents | Reconciles all notifications from SAA and relates notifications to the original document using the *SenderReference* attribute. |
| Document Attributes | Format | Identifies the format of the message (MT, MX, or any XML type). |
| | SenderReference | Extracted from the outbound document (being sent to) SAA and uniquely identifies any document sent to SAA. |
| | ReconciliationInfo | Used to reconcile the MX Delivery Notification message to the corresponding Transmission report. (This attribute is extracted from the inbound Delivery Notification.) |
| Document Types | DeliveryNotification | Corresponds to the delivery notification sent by SWIFT to SWIFT Module for successful delivery of a message to a counterparty. |

### To import Trading Networks information for handling notifications

- Using the file located in the *Integration Server_directory*\packages\WmFIN \config\xmlv2\ xmlv2TNItems.dat directory, import Trading Networks assets using the instructions described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

## Step 2: Configure SWIFT Module to Handle Notifications

### To configure SWIFT Module to handle notifications

1. Configure routing rules in SAA using the Routing Application to send different notifications to specific end points (AFT or MQHA). For more information about configuring routing rules in SAA, see *SWIFT Alliance Access Administration Guide*.
2. Configure the routing on SAA to send notifications to:
   - AFT - Create a new service to process notifications and submit them to Trading Networks for further processing, or modify the sample service wm.xmlv2:recieveFromAFT from the SWIFT Module samples. For more information about the SWIFT Module sample services, see *webMethods SWIFT Module Samples Guide*.
   - MQHA - Create a listener and notification for this connection:
     - Create a listener to wait for a notification document and a trigger that subscribes to the notification document. The notification document is published when a message is put into the MQHA queue by SAA.
     - Create a new service to handle notifications or modify the sample service, wm.xmlv2:receiveFromMQ. This service is called from the above trigger which subscribes to the Notification document.

   Sample services for the connection, listener, notification, trigger, and receive services are available in the SWIFT Module samples. For more information about the SWIFT Module sample services, see *webMethods SWIFT Module Samples Guide*.
3. Configure the connection details for SAA as described in:
   - For AFT, see "Using AFT to Communicate with SWIFT" on page 116.
   - For MQHA, see "Using WebSphere MQ Adapter to Communicate with SWIFT" on page 110.

## Step 3: View Notifications and Related Messages

You can view the different types of notifications and the messages to which they relate using the Trading Networks pages in My webMethods.

Search for the notification you want to view in Trading Networks by following the instructions described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

### Notification Details Displayed in the Transaction Details Panel

In the Transaction Details panel there are several tabs that display information about the notifications. The following table describes the most important details for the different types of notifications that you can find on the Activity Log and Content tabs. For information about all tabs in the Transaction Details panel, see the Trading Networks administration guide for your release. See “About this Guide” for specific document titles..

| Tab | Notification | Details |
|---|---|---|
| Activity Log | | The **Brief Message** column contains the Document ID of the original message or TN document to which the notification or report is related: |
| | Delivery Notification | The TN document related to this notification, for example:<br>`Related to 53o6d600488qm9n8000004ov (Delivery Notification).` |
| | Delivery Report | ID of the message related to this report, for example:<br>`Related to 53o6d600488qm65j000004nm (Delivery Report).` |
| | History Report | ID of the message related to this report, for example:<br>`Related to 50k2ga0049dnl10r000000cs(History Report).` |
| | Transmission Report | ID of the message related to this report, for example:<br>`Related to 50k2ga0049dnl10r000000cs (NetworkAcked).` |
| Content | | **Details** displays the contents of the selected item: |
| | Delivery Notification | The delivery notification contents of the bizdoc, which is in either XML or MT format:<br>■ For `xmlData`—The Data PDU content in XML format. The body tag contains the MT or MX message data:<br>■ MT message—Base 64 encoded format.<br>■ MX message—The delivery information.<br>■ For `finMsg`—The MT content. The `108:` tag in the decoded MT message contains the *SenderReference* of the original document, for example:<br>`{175:1049}{106:090624PTSAUSA0AXXX0077000422}{108:MT910586328}{175:1059}{107:090624PTSAUSA0AXXX0078000882}` |

| Tab | Notification | Details |
|---|---|---|
| | Delivery Report | The delivery report for the selected item:<br>■ For `xmlData`—The Data PDU in XML format.<br>■ For `SAA`—The SAA content. The `108:` tag contains the *SenderReference* of the original document, for example:<br>`{175:1049}{106:090624PTSAUSA0AXXX0077000419}{108:MT210244895}{175:1049}{107:090624PTSAUSA0AXXX0077000872}` |
| | History Report | For `xmlData`, the actual Data PDU content of the history report in XML format. |
| | Transmission Report | The content of the transmission report:<br>■ For `xmlData`—The Data PDU in XML format.<br>■ For `SAA`—The SAA content. The `451`: tag provides the processing status of the original document by SWIFT Network, for example:<br>`{1:F21PTSAUSA0AXXX0090000441}{4:{177:0907081124}{451:0}{108:MT199704775}}`<br>The value of the tag in the example is 0, indicating that the original message has been successfully acknowledged by SWIFT Network. |

For examples of the Data PDU content of the different types of notifications, see "Examples of Data PDU Content of Documents" on page 313.

# 13 Using SAA to Exchange XML v2 Wrapped MT and MX Messages

# Overview

SWIFT Module supports the exchange of MX and MT messages over the SWIFT Network through SWIFT Alliance Access (SAA). SWIFT Module, also supports FileAct and InterAct messaging services for transporting MX messages. MT and MX messages exchanged through SAA are wrapped in XML v2 data format. The root tag for this data format is `Data PDU`. The body tag of the `Data PDU` may contain an MX message or a base 64-encoded MT message.

SWIFT Module also handles inbound messages from SAA. You can configure custom services to be triggered for these messages.

SWIFT Module validates MX messages against the SWIFT standards.

# Exchanging MT Messages in XMLv2 Format

To exchange MT messages using XML v2 format, you must do the following:

- "Step 1: Configure Trading Partners for Message Exchange" on page 126
- "Step 2: Create Trading Networks Items" on page 127
- "Step 3: Send the MT Message to SAA" on page 133
- "Step 4: Reconcile the Notification from SWIFT with the Original MT Message" on page 133

## Step 1: Configure Trading Partners for Message Exchange

To exchange XML v2 wrapped MT messages over SAA, configure trading partner profiles. For information about how to configure trading partner profiles, see "About Trading Partner Profiles" on page 80.

### To view trading partner information

1 Follow the instructions for viewing Trading partner information in Trading Networks as described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

2 In the Partner Details, you can view the following information that Trading Networks uses to identify a trading partner for an MT message exchange:

| Field | Description |
|---|---|
| Corporation Name | Name of the partner's corporation, for example, `Software AG`. |
| Partner Type | This specifies that the corporation uses Trading Networks. The default value is `webMethods Trading Networks`. |

| Field | Description |
|---|---|
| External ID | Identifies the partner as a sender or receiver of the MT message.The default value is `BIC`, for example, PTSAUSA0XXX. |

3 For information about the other fields in the Partner Details panel of the Partner Profile page, see the Trading Networks administration guide for your release. See “About this Guide” for specific document titles.

## Step 2: Create Trading Networks Items

In Designer, run the wm.xmlv2.dev:createSWIFTItems service to create the following Trading Networks assets for an MT message: a TN document type, a processing rule, and a TPA. The following table lists the parameters that you should define for MT messages:

| Parameter | Value |
|---|---|
| *msgTypeName* | The MT message type for which a TN document type must be created, for example, `fin.535`. |
| *format* | MT |
| *finFormat* | Required. Defines the format of the IS document that is generated for the MT message type. The default value is `TAG_BIZNAME`. |
| *version* | Required. The version of the SWIFT specification, for example, nov10. |
| *subfieldFlag* | Required. Specifies whether the fields generated in the IS document type are parsed to the subfield level. The default value is `true`. |
| *createProcessingRule* | Creates a default processing rule for the specified document type. The default is `false`. |
| *createTPA* | Creates a Trading Networks TPA for the message that specifies variables used in WmFIN for processing and validation. The default is `true`. |
| *createDocType* | Creates and inserts a TN document type for the message. The TN document type is used to recognize an incoming message. The default is `true`. |

The parameters *finFormat*, *version*, and *subFieldFlag* are required for an MT message type because the service uses these to generate an IS document type for the corresponding message type (an MT message in this case). This service internally invokes wm.fin.dev:importFINItems to import the relevant DFDs and SWIFT message templates required for the validation and parsing of the MT message. For more information, see wm.xmlv2.dev:createSWIFTItems.

## Viewing Trading Networks Assets for an MT Message

You can view the Trading Networks assets that wm.xmlv2.dev:createSWIFTItems generates for an MT message in Trading Networks. For more information about searching for and viewing items in Trading Networks, follow the instructions described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

### About TN Document Types

TN document types are created when the *createDocType* parameter in the wm.xmlv2.dev:createSWIFTItems service is set to `true`. You can view the following parameters for TN document types for MT messages:

- **Identification Information.** The information that Trading Networks uses to determine whether a document matches a defined TN XML document type.
- **Extraction Information.** The attributes that you want Trading Networks to extract from the XML document. The *SenderID* and *ReceiverID* attributes should be selected from the Data PDU XML document for use in processing rules, transaction analysis, and process management.

### To view details for TN document types for MT messages

1. Follow the instructions for viewing TN document types as described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.
2. From the **Document Types** list, click the MT message document type that you want to view (for example, `fin.535`).
3. In the Document Type Details screen, on the Identify tab, the following identification information is available for the MT message TN document type:

| Field | Description |
|---|---|
| Name | The name of the document type, for example, `fin.535`. |
| Description | Indicates that the XML format of the document corresponds to the Data PDU XML format (for example, DataPDU for doc type = fin.535). |
| Root Tag | The root tag that identifies the message as a Data PDU XML document when it is submitted to Trading Networks. The default value is `DataPDU`. |

| Field | Description |
|---|---|
| Identifying Query | Uniquely identifies the MT message, indicating the location of the MT message element in the Data PDU XML document. You can specify which value Trading Networks evaluates to determine if it matches the TN XML document type.<br>For example, if the Identifying Query is: `*:DataPDU/*:Header/*:Message/ *:MessageIdentifier`, and you define the value as `fin.535`, Trading Networks does the following for any `fin.535` document submitted:<br>■ Runs the above query against the XML structure of the document and extracts the value at the specified location.<br>■ Identifies the document as a fin.535 message.<br>■ Continues with the next action defined for this message type.<br>To view or edit Identifying Queries, follow the instructions described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles. |

4 To extract information about the *SenderID* and *ReceiverID* attributes, do the following:

a Follow the instructions for extracting information about attributes as described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

b From the **Attributes to Extract** list, select `SenderID` or `ReceiverID` to edit.

c On the Edit Attribute screen, you can edit or view the following details:

| Field | Description |
|---|---|
| Name | The name of the attribute, for example, `SenderID` or `ReceiverID` |
| Query | Indicates to Trading Networks how to extract the attribute for the Data PDU XML document type, for example:<br>■ SenderID Query: `/*:DataPDU[0]/*:Header[0]/*:*/*:Sender[0]/*:*/*:X1[0]`<br>■ ReceiverID Query: `/*:DataPDU[0]/*:Header[0]/*:*/*:Receiver[0]/*:*/*:X1[0]` |
| Built-in Transformation | Specifies the external ID type associated with the information in the document for the sender or receiver. For MT messages the external ID type for the sender and the receiver is `BIC`. |

### About Processing Rules

Processing rules are created when the *createProcessingRule* parameter in the wm.xmlv2.dev:createSWIFTItems service is set to `true`.

### To view details for processing rules for MT messages

1. Follow the instructions for viewing processing rules as described in theTrading Networks administration guide for your release. See "About this Guide" for specific document titles.

2. From the **Processing Rules** list, click the MT message document type that you want to view, for example, `fin.535`.

3. On the Criteria tab in the Processing Rule Details screen, you can view the processing rule criteria. Trading Networks uses these parameters to identify the documents that trigger the rule execution.

| Field | Description |
|---|---|
| Name | The name of the processing rule, for example, `fin.535`. |
| Sender | The sender. The default value is `Any`. |
| Receiver | The receiver. The default value is `Any`. |
| Document Type | TN document type specified for the document. The default value is `Selected`. For example, if the TN document type is `fin.535`, the processing rule is triggered only for `fin.535` messages. |
| User Status | For Trading Networks to determine which processing rule to invoke, the processing rule must have this additional **User Status** value for outgoing documents. The default value is `AwaitingDelivery`.<br>This value is very important to the processing rule. All MT/MX messages are submitted to Trading Networks before they are sent to SAA from SWIFT Module. Trading Networks uses the information in the TPA to determine whether to send a document to SAA, and invokes the corresponding processing rule accordingly. The processing rule must have this additional **User Status** value. Messages received from SAA are processed similarly. |
| Recognition Errors | Indicates the possibility that the document has errors. The default value is `May have errors`. |

4. On the Action tab in the Processing Rule Details screen, you can view the actions in the processing rule that Trading Networks performs to process the document:

   - The action selected for the MT message processing rule is `Execute a service`.
   - The Execute A Service panel shows the name of the service that Trading Networks executes. The default service is wm.xmlv2.process:outbound, which sends the document to SAA based on the TPA information. Trading Networks invokes the service synchronously.

## About Trading Partner Agreements

### To view details for TPAs for MT messages

1 Follow the instructions for viewing Agreements as described in the Trading Networks administration guide for your release. See “About this Guide” for specific document titles.

2 From the **Agreements** list, click the MT message document type that you want to view (for example, `fin.535`).

3 In the Agreement Details screen, view the following important details for the default TPA, generated by the wm.xmlv2.dev:createSWIFTItemsservice when the *createTPA* parameter in this service is set to `true`:

| Field | Description |
|---|---|
| Sender | Specifies the partner that has the sending role in the TPA.The default value is `Unknown`. |
| Receiver | Specifies the partner that has the receiving role in the TPA. The default value is `Unknown`. |
| Agreement ID | Uniquely identifies the type of agreement between two partners, for example, `fin.535`. |
| IS document type | Specifies the data that you define in the TPA for processing the MT message document type. The default value is wm.xmlv2.doc:XMLV2Params. This IS document contains three main sections: |

| Field | Description | |
|---|---|---|
| | IS Doc Section | Description |
| | ProcessInfo | This section consists of the following parameters:<br>■ *Transport*. The transport to use for sending the Data PDU document to SAA. Select `MQ` or `AFT`.<br>■ *MTProcessInfo*. Supports backward compatibility for MT messaging in a SWIFT FIN 6.1 TPA. For more information, see *webMethods SWIFT FIN Module Installation and User's Guide 6.1*. |
| | | ■ *MXProcessInfo*. The validation to perform on the MX message:<br>■ `Validate`. Validate the MX message.<br>■ `SchemaValidation`. Perform a schema validation.<br>■ `NonSchemaValidation`. Perform a non-schema (extended) validation.<br>For details about these parameters, see "Process Information Section of the XMLv2 Parameters Document" on page 257. |
| | | ■ *ns:Message*. Contains SWIFT specific information required for sending the message to SAA. For more information, see sample service, wm.xmlv2.MT.maps:mapDataPDU. |
| | MQSeriesInfo | For more information, see "Modifying the TPA" on page 94. |
| | AFT | |
| Description | Indicates that the TPA is for an MT message document type, for example, `TPA for fin.535`. | |

For information about the other fields in the Agreement Details screen of the TPA, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

## Step 3: Send the MT Message to SAA

To send the MT message to SAA, create your own custom service. Use the sample service, wm.xmlv2.MT:sampleMTExchangeUseTPA located in the SWIFT Module sample package as an example.

**To view detailed information for the MT message that SWIFT Module submits to Trading Networks**

1 Follow the instructions to view Transactions as described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

2 In the Transaction Details panel, view detailed information for the MT message that SWIFT Module submits to Trading Networks.

- The Attributes tab provides details about the MT message. The attribute User Status is very important. When the value is "`Waiting SWIFT Network`," it indicates that Trading Networks has successfully submitted the message to SAA and is waiting for an acknowledgement from SWIFT Network. If a problem occurs while sending the document, the User Status is updated to `SentFailed`.

- The Content tab provides the following information:

  - The Data PDU content of the MT message. For an example of the Data PDU content, see "Examples of Data PDU Content of Documents" on page 313.

  - The decoded MT Data of the MT message.

For information about how to view detailed information for the MT message, see "Step 3: View Notifications and Related Messages" on page 122.

## Step 4: Reconcile the Notification from SWIFT with the Original MT Message

After SWIFT Module submits the MT message to SAA, the Trading Networks bizdoc waits for the transmission notification from SWIFT. When the MT message bizdoc receives the transmission notification from SWIFT, SWIFT Module reconciles the notification with the original MT message.

The complete processing of an MT message involves reconciliation of all types of notification documents (history report, delivery report, delivery notification and transmission report) with the original document.

For information about how to handle notifications, seeChapter 12, "Configuring Notifications for Messages in XML v2 Format".

# Exchanging MX Messages through SAA

SWIFT Module supports the exchange of MX messages through SAA using FileAct and InterAct messaging services. To send MX messages through SAA, you must do the following:

- "Step 1: Configure Trading Partners for Message Exchange" on page 134
- "Step 2: Create Trading Networks Assets" on page 134
- "Step 3: Create IS Schema and IS Document Type" on page 137
- "Step 4: Send the MX Message to SAA" on page 138
- "Step 5: Receive an MX Document from SAA" on page 139

## Step 1: Configure Trading Partners for Message Exchange

You must configure trading partner profiles to exchange MX messages over SAA. For information about configuring trading partner profiles, see .

## Step 2: Create Trading Networks Assets

### To create Trading Networks assets for MX message types

1 In Designer, run the wm.xmlv2.dev:createSWIFTItems service to create a TN document type, processing rule, and TPA for MX messages in Trading Networks.

2 Define the following parameters:

| Parameter | Value |
|---|---|
| *msgTypeName* | Specify the MX message type for which a TN document type must be created, for example, `camt.029.001.01`. |
| *format* | MX |
| *createProcessingRule* | Creates a default processing rule for the specified document type. The default is `false`. |
| *createTPA* | Creates a Trading Networks TPA for this message that specifies the variables used in WmFIN for processing and validation. The default is `true`. |
| *createDocType* | Creates and inserts a TN document type for this message. The default is `true`. |

**Important!** The *finFormat*, *version*, and *subfieldFlag* fields are not required for an MX message. Use the default values for these fields.

For more information, see the wm.xmlv2.dev:createSWIFTItems service.

## Viewing or Modifying Trading Networks Assets for an MX Message

### To view or modify Trading Networks assets for an MX Message

1. To view or modify the Trading Networks assets that the wm.xmlv2.dev:createSWIFTItems service created for an MX message, follow the instructions described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.
2. Click the **Identify** tab on the Document Type Details screen. The following table highlights the most important details for the MX TN document type:

| Field | Description |
|---|---|
| Name | The name of the document type, for example, `camt.029.001.01`. |
| Description | Indicates that the document is in an XML format that corresponds to the Data PDU XML, for example, `camt.029.001.01`. |
| Root Tag | The root tag that serves to identify the message as a Data PDU XML document when it is submitted to Trading Networks. The default value is `DataPDU`. |
| Identifying Query | Uniquely identifies the MX message, indicating the location of the MX message element in the Data PDU XML document. You can specify which value Trading Networks evaluates to determine if it matches the TN XML document type.<br>For example, if the Identifying Query is `*:DataPDU/*:Header/*:Message/*:MessageIdentifier`, and you define the value as `camt.029.001.01`, Trading Networks does the following for any `camt.029.001.01` document submitted to it:<br>■ Runs the above query against the XML structure of the document and extracts the value at the specified location.<br>■ Identifies the document as a `camt.029.001.01` message.<br>■ Continues with the next action defined for this message type.<br>To view or edit Identifying Queries, follow the instructions described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles. |

3 Click the Edit Attribute screen to view or edit the following attributes:

| Field | Description |
|---|---|
| Name | The name of the attribute. You can extract the following attributes for an MX message:<br>■ SenderID<br>■ ReceiverID<br>■ SenderReference<br>■ Format<br>■ ReconciliationInfo |
| Query | Instructs Trading Networks how to extract the attribute for the Data PDU XML document type. For example:<br>■ SenderID: `/*:DataPDU[0]/*:Header[0]/*:*/*:Sender[0]/*:*/*:X1[0]`<br>■ ReceiverID: `/*:DataPDU[0]/*:Header[0]/*:*/*:Receiver[0]/*:*/*:X1[0]`<br>■ SenderReference: `/*:DataPDU[0]/*:Header[0]/*:*/*:SenderReference[0]`<br>■ Format: `/*:DataPDU[0]/*:Header[0]/*:*/*:Format[0]`<br>■ ReconciliationInfo: `/*:DataPDU[0]/*:Header[0]/*:*/*:ReconciliationInfo[0]` |
| Built-in Transformation | Specifies the external ID type associated with the information in the document for the sender or receiver. For MX messages the external ID type for the sender and the receiver is `BIC`. |

**Note:** To extract attributes in the TN document type, follow step 4 in the "To view details for TN document types for MT messages" procedure.

4 On the Processing Rule Details screen:

- Click the Criteria tab, to view the following processing rule criteria that Trading Networks uses to identify the documents:

| Field | Description |
|---|---|
| Name | The name of the processing rule, for example, camt.029.001.01. |
| Sender | The sender. The default value is `Any`. |
| Receiver | The receiver. The default value is `Any`. |
| Document Type | The default value is `Selected`, and a TN document type is specified for the document. For example, if the TN document type is camt.029.001.01, the processing rule is triggered only for the routed camt.029.001.01 message. |
| User Status | For Trading Networks to determine which processing rule to invoke, the processing rule must define this additional User Status value for outgoing documents. The default value is `AwaitingDelivery`.<br>This value is very important to the processing rule. All MT/MX messages are submitted to Trading Networks before they are sent to SAA from SWIFT Module. Trading Networks uses the information in the TPA to determine whether to send a document to SAA, and invokes the corresponding processing rule accordingly. The processing rule must have this additional User Status value. Messages received from SAA are processed similarly. |
| Recognition Errors | Indicates whether to specify if the document has errors. The default value is `May have errors` |

- Click the Action tab to view the processing rule action details. (These details are the same as for an MT message. For information, see step 4 in the “To view details for processing rules for MT messages” procedure.)

5 To view the MX TPA, follow the procedure described in “To view details for TPAs for MT messages.” The details are the same as in the MT TPA, essage, except for the Agreement ID, which shows the type of message that the TPA represents (MX message). For example, the Agreement ID for an MX message may have the value `camt.029.001.01`.

## Step 3: Create IS Schema and IS Document Type

The MX message structure is defined by an XML schema. You must create an XML schema in Integration Server that corresponds to the MX message structure to populate the data values in the pre-defined placeholders for that particular MX message type.

### To create an IS schema and IS document type for an MX message type

1. Start Designer.
2. Navigate to the package and folder of the MX message type for which you want to create an IS document type (for example, `camt.029.001.01`).
3. Follow the instructions for creating document types as described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.
4. In the **Name** field in the New Document Type dialog box, type a name for the IS document type using any combination of letters, numbers, and/or the underscore character (for example, `camt_029_001_01`) and click **Next**.
5. Under **Select a source**, select **XML Schema** and click **Next**.
6. In the **Enter the URL or select a local file** box and browse to the location of the XML schema to be used for creating this message type (for example, *Integration Server_directory*/packages/WmFIN/config/schemas/camt_029_001_01.xsd). Click **Next**.
7. In the New Document Type dialog box do the following:
   a. Under **Select the root node**, specify the root element of the document (for example, `Document#urn:iso:std:iso:20022:tech:xsd:camt.029.001.01`).
   b. Under **Select schema type processing**, select **Expand complex types inline**. Designer processes complex types by expanding them inline in the editor.
   c. Click **Finish**.
8. Designer generates the IS document type and IS schema and saves the IS document type on Integration Server. You can view the IS document type in the editor and the IS schema in the Navigation panel. For information about creating IS document types and IS document schemas, see the Designer online help for your release. See "About this Guide" for specific document titles.
9. The WmSWIFTSamples package contains IS schemas and IS document types already created for camt.029.001.01 and camt.007.002.01 messages in the folders wm.xmlv2.doc.camt_029_001_01 and wm.xmlv2.doc.camt_029_001_01, respectively. For information about the samples package, see *webMethods SWIFT Module Samples Guide*.

## Step 4: Send the MX Message to SAA

SWIFT Module supports the use of FileAct and InterAct messaging services with SAA. To send the MX message to SAA, create and run your own service in Designer. You can use the sample wm.xmlv2.MX:sample_camt029_001_01 service and wm.xmlv2.fileact.camt007_002_01:sendFile as examples. Follow the instructions for "Viewing Transactions" as described in theTrading Networks administration guide for your release. See "About this Guide" for specific document titles.

In the Transaction Details panel, you can view detailed information for the MX message that SWIFT Module submits to Trading Networks:

- The Attributes tab provides details about the MX message bizdoc. The **User Status** attribute provides important information about the status of the message transmission to SAA. The status, `Waiting SWIFT Network`, shows that Trading Networks has submitted the MT message to SAA successfully, and is waiting for an acknowledgment from the SWIFT Network. Trading Networks updates the status to reflect the state of the message transmission. If a problem occurs while sending the document, the **User Status** is updated to `SentFailed`.
- The Content tab provides the following information:
    - The Data PDU content of the MX message bizdoc
    - The MX Header
    - The MX Document

  For examples of the Data PDU content, the MX Header, and the MX Document, see "Examples of Data PDU Content of Documents" on page 313.
- The Activity Log tab shows the activity log entry for the MX message type. In the Details section, you can view the full message for the entry. The message contains the name of the default processing service for this document.

For information about how to view detailed information for the MX message, see "Step 3: View Notifications and Related Messages" on page 122.

### Step 5: Receive an MX Document from SAA

SWIFT Module reconciles MX message transmission notifications with the original MX message. SWIFT Module updates the **User Status** of the message to `NetworkAck` or `NetworkNack` based on the response from the SWIFT Network.

SWIFT Module receives MX documents from SAA and uses the transport setting for the outbound traffic from SAA to determine how to receive them. The process for receiving any MX message from SAA is the same as that described in Chapter 12, "Configuring Notifications for Messages in XML v2 Format".

The following services are included in the SWIFT Module samples. You can use these services as is or as a model to create your own services:

- For MQHA, use the sample service, wm.xmlv2:recieveFromMQ.
- For AFT, use the sample service, wm.xmlv2:recieveFromAFT.

## Validating MX Messages Conform to SWIFT Standards

With SWIFT Module, you can perform two types of MX message validation:

- Schema validation

- Extended validation

## Schema Validation of MX Messages

Schema validations perform two tests on a message: that the MX message is a well-formed XML document, and that the MX message is valid. To perform schema validations, use the services provided in the WmPublic package. For information about this package, see the Integration Server built-in services reference guide for your release. See "About this Guide" for specific document titles.

## Extended Validation of MX Messages

Extended, or non-schema validation, ensures that SWIFT MT and MX messages comply with SWIFT Standards. Use SWIFT Module to perform extended validation when the XML schema does not validate the document completely. To perform non-schema validation on an MX message, run the wm.unifi.validation:validateMXMsg service in Designer.

You can also perform individual validation tasks, such as BIC validation, or IBAN validation only.

### To perform an individual validation task

1 Start Integration Server and Designer.

2 Configure the XMLV2Params document to trigger validation only for the fields that you want to validate separately. For information about how to configure this document, see "Process Information Section of the XMLv2 Parameters Document" on page 257.

3 In Designer, run the SWIFT Module built-in service that performs the individual validation task you want as follows:

| Run this service... | To validate that... |
|---|---|
| wm.unifi.validation:validateBIC | The following data types exist in the BIC directory (ISO 9362) on SWIFTNet:<br>■ XML elements of type BIC (data type: `BICIdentifier`)<br>■ XML elements of type BIC or BEI (data type: `AnyBICIdentifier`). |
| wm.unifi.validation:validateBEI | XML elements of type BEI (data type: `BEIIdentifier`) exist in the BEI list on SWIFTNet. |

| Run this service... | To validate that... |
|---|---|
| wm.unifi.validation:validateCurrencyCode | The following data types exist in Currency Code ISO 4217:<br>■ XML elements of type ActiveCurrency (data type: `ActiveCurrencyCode`)<br>■ XML elements of type ActiveOrHistoricCurrency (Data type: `ActiveOrHistoricCurrencyCode`).<br>■ XML elements containing an amount and a currency (Data type: `ActiveCurrencyAndAmount` and `ActiveOrHistoricCurrencyAndAmount`). Also verifies that the number of digits in the amount is as specified by ISO 4217 for that specific currency. |
| wm.unifi.validation:validateCountryCode | Country codes (data type: `CountryCode`) exist in ISO 3166. |
| wm.unifi.validation:validateIBAN | IBAN identifiers (data type: `IBANIdentifier`) match the IBAN structure as specified by ISO 13616 (which contains the country code, the specified number of digits, and the basic bank account number). |

# 14 Working with Market Practices

# Overview

Market Practices are specific requirements for individual markets. Using Trading Partner Agreements (TPAs), SWIFT Module supports customization of SWIFT FIN messages based on specific trading partner sender-receiver pairs. For example, two partners trading within France might have different processing requirements for their SWIFT FIN messages than two trading partners within Austria.

SWIFT FIN messages that are exchanged between two partners may have additional fields and/or a subset of key words. SWIFT Module enables you to maintain multiple versions of a given message that conform to different Market Practices.

# Creating Market Practices

Create a Market Practice by creating an alternate version of the SWIFT message based on an original message record. In this way you maintain the original content of the message record.

### To create a Market Practice

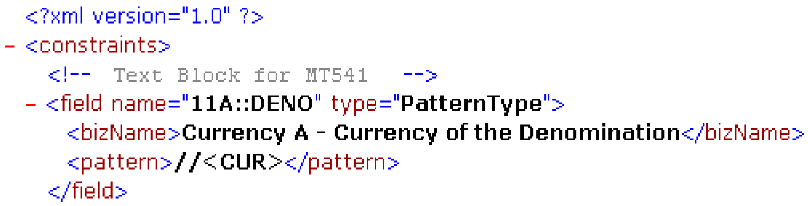
1 On your file system, create identically named folders (for example, `FrenchMarket`) in the following directories:

- *Integration Server_directory*\packages\WmFIN\import
- *Integration Server_directory*\packages\WmFIN\config\dfd

2 Copy `dfd000.xml` from WmFIN\config\dfd\`nov10` to WmFIN\config\dfd\`FrenchMarket`, where `nov10` is the SWIFT message version.

3 Copy the `dfd*.xml` file (for example, `dfd541.xml`) for your message from WmFIN\import\`nov10` to WmFIN\import\`FrenchMarket`.

4 Open your `French Market\dfd*.xml` and edit it as necessary.



```
  <?xml version="1.0" ?>
- <constraints>
    <!--  Text Block for MT541   -->
  - <field name="11A::DENO" type="PatternType">
      <bizName>Currency A - Currency of the Denomination</bizName>
      <pattern>//<CUR></pattern>
    </field>
```

5 In Designer, run the wm.fin.dev:importFINItems service for the message.

6 On the Input screen for the importFINItems service, set the version field to the name of the new folder (for example, `FrenchMarket`). Set the remaining fields as desired.

7 In Trading Networks, open the TPA and define the following parameters:

- Set **ISDocumentName** to the location the new message record (for example, `wm.fin.doc.FrenchMarket.cat1:MT103`).
- Set **Version** to a new Market Practice version name (for example, `FrenchMarket`).
- Set **MarketPracticeRulesService** to the Market Practice rule for this SWIFT message.

For more information about TPAs, see Chapter 8, "Customizing Trading Partner Agreements".

## Creating Market Practice Rules

SWIFT Module provides sixteen common Market Practice rules for Category 5 SWIFT FIN messages. You can create additional Market Practice rules by writing services based on message documentation (.pdf) provided by SWIFT.

To use a new Market Practice rule, you must specify the service you created in the **MarketPracticeRulesService** parameter in the TPA for the particular SWIFT message. For more information about TPAs, see Chapter 8, "Customizing Trading Partner Agreements".

# III Configuring SWIFT Module for FileAct and InterAct Message Exchange Over SAG

# 15 Configuration Steps for InterAct and FileAct Messaging Services over SAG RAHA

## Overview

The SWIFTNet component of SWIFT Module provides client-side and server-side support for the InterAct and FileAct messaging services. You can use these services to exchange messages and files with SWIFT Alliance Gateway (SAG).

The SWIFTNet component supports two types of transport: the Remote API Host Adapter (RAHA) and the MQ Host Adapter (MQHA). This chapter describes how to prepare your application server or client to exchange messages and files over SAG using the RAHA transport. RAHA uses the Remote API (RA) client on your Integration Server to enable message and file exchange.

For information about the SWIFT messaging services and the two types of transport, see "SWIFTNet Component" on page 36. For more information about how to configure the MQHA transport, see Chapter 16, "Configuration Steps for InterAct and FileAct Messaging Services over SAG MQHA".

**Important!** The following steps assume that you have already installed Integration Server, Trading Networks, and SWIFT Module, and have imported the SWIFT lists that you need in order to use the BICPlusIBAN and SEPA directories. For steps to install SWIFT Module, see Chapter 2, "Installing webMethods SWIFT Module". For more information about BICs and IBANs, see Chapter 4, "Importing BICPlusIBAN and SEPA Routing Directories". For more information about the SWIFT software you need, work with SWIFT to determine your software needs.

## Step 1: Prepare the Server to Handle Requests

To prepare your server application to receive and respond to requests using the SWIFTNet component of SWIFT Module, you must complete three configuration tasks:

| Task | Description |
|---|---|
| 1 | Configure SAG to communicate with your RA client, SWIFT Module, and Integration Server. |
| 2 | Configure the SWIFTNet component. |
| 3 | Configure Trading Networks information. |

## Configuring SWIFT Alliance Gateway

### Task 1 Configure SWIFT Alliance Gateway (SAG)

1 **Install RAHA.** Install RAHA on the same machine as SAG. RAHA enables SAG to exchange messages and files with the RA client on the same machine as your Integration Server. To obtain an appropriate RAHA, contact SWIFT.

2 **Configure Message Partners and Endpoints.** Configure the server message partners for the server module, and the client message partners for the client module.

**Important!** If at any time SAG restarts, you must reload the WmSWIFTNetServer and WmSWIFTNetClient packages.

For more information about completing these steps, see *SWIFT Alliance Gateway File Transfer Interface Guide*, *SWIFT Alliance Gateway Operations Guide*, and *Remote API for SWIFT Alliance Gateway Operations Guide*.

## Configuring the SWIFTNet Component

### Task 2 Configure the SWIFTNet component

1 Install a RA client on the same machine as Integration Server. The RA client enables the SWIFTNet component to communicate with your SAG and SNL through RAHA. To obtain an RA client, contact SWIFT.

2 Set the server application remote process connection settings.

a From Integration Server Administrator, select **Adapters > SWIFT**.

b On the SWIFT Module home page, select **SWIFTNet Server Configuration** > **Edit.**

c On the SWIFTNet Server Configuration screen, in the SWIFTNet Remote Process Connection Configuration section, define how the remote process listeners connect to Integration Server for an incoming request from the SWIFT Network as follows:

| For this property... | Specify... |
|---|---|
| User Name | The Integration Server user that has permissions to execute the required services that the remote process listeners will invoke for incoming requests from the SWIFT Network. |
| Password | The user password. |
| Host IP | The IP address of the machine on which Integration Server is running. |
| Host Port | The port of the machine on which Integration Server is running. |

d Test the remote process connection settings by clicking **Test Connection Settings**. SWIFT Module attempts to simulate opening a connection to Integration Server in the same way as the SWIFTNet process handlers to establish a connection in real-time for an incoming request from SWIFT Network.

  If the test connection fails, an error message indicates which input values are incorrect. For example, if the Host IP address value is incorrect, the error message includes this value. You can enter the correct value and test again, if required.

3 Set the server application environment variables.

  In the **SWIFTNet Server Environment Info** section on the SWIFTNet Server Configuration screen, set the following environment properties for the RA client.

| For this property... | Specify... |
|---|---|
| SWNET_CFG_PATH | The cfg folder of the RA instance in your system, for example, c:\SWIFTAlliance\RA\Ra1\cfg\. |
| SystemRoot | The system root folder. The value of this parameter depends on your operating system, for example, c:\windows. |
| SWNET_BIN_PATH | The lib folder of the RA instance in your system, for example, c:\SWIFTAlliance\RA\lib. |
| SWNET_HOME | The RA Home folder, for example, c:\SWIFTAlliance\RA. |
| PROCESS_INSTANCES | The maximum number of processes the SWIFTNet component will process at the same time. The default is `3`.<br>**Note:** There is no maximum value for the number of processes. However, the number of processes specified should not exceed the concurrent server processes configured on SAG as specified in the SAG documentation. |
| RMI Port | The Remote Method Invocation (RMI) port where the remote process listeners will be bound in the Java Naming and Directory Interface (JNDI) tree, for example, `10985`. |

4 Set the server application connection properties in the **SWIFTNet Server SAG Connection Properties** section of the SWIFTNet Server Configuration screen as described in the table below.

  When Integration Server starts, SWIFT Module automatically registers itself as the server module with the SNL libraries on your SAG, and exchanges a series of pre-defined SNL primitives in sequence with your SNL libraries using your RA client.

  You define the properties that SWIFT Module uses to populate these primitives. For the SWIFTNet component to exchange information with your SAG, you must set these properties using the information you used to configure your SAG in "Configuring SWIFT Alliance Gateway" on page 151.

Set the following properties to define how to connect to SAG using the RAHA transport:

| For this property... | Specify... |
|---|---|
| SAGMessagePartner | The Server message partner defined in SAG. |
| server_pki_profile | The user name of the configured profile that will be used for opening a security context with SAG for sending the request to SWIFT Network. |
| server_pki_password | The password associated with the user name of the server PKI profile defined in your SAG. This password is used to unlock the Server PKI profile. |
| userDN | The Distinguished Name to be used for sign, encryption, and authorization operations. Valid values are: `cn=`*encryptCN*`, o=`*bic*`, o=swift`. |
| encryptDN | The Distinguished Name to be used for encryption. Value values are: `cn=`*encryptCN*`, o=`*bic*`, o=swift` |
| Sign, Decrypt, and Authorization | `True` or `False` for each to specify whether the security context opened during server initialization should be used.<br>**Note:** At least one field must be set to `True`. |
| AllFileEvents | `True` or `False` to populate the Sw:SubscribeFileEventRequest primitive exchanged during server initialization.<br>■ `True`—Default. SWIFT Module receives all events generated by the SAG File subsystem during file transfer.<br>■ `False`—SWIFT Module receives only state transition events. |
| FullFileStatus | `True` or `False` to populate the Sw:SubscribeFileEventRequest primitive exchanged during server initialization.<br>■ `True`—Default. SWIFT Module receives full details for each event report generated by the SAG system.<br>■ `False`—SWIFT Module does not receive the detailed status for the file transfer. |
| SwEventEP | The file transfer event endpoint to which file transfer events are posted by your SAG during FileAct operations. This value is used to populate the primitive Sw:SubscribeFileEventRequest exchanged during server application initialization. |

| For this property... | Specify... |
|---|---|
| ReceptionFolder | The default folder to receive incoming files. When this field is blank, the folder is created in the following location:<br>*Integration Server_directory*\packages\WmSWIFTNetServer \pub\SWIFTNetReceptionFolder.<br>If the specified folder does not exist, it is created. |
| SwTransferEP | The default transfer endpoint of the remote file handler. This property is optional on Windows systems and required on UNIX systems. If this field is specified, the value must match a remote file handler endpoint running on the same machine as Integration Server.<br>For information about invoking the remote file handler, see "Step 3: Invoke the Remote File Handler" on page 159. |
| cryptoMode | Specifies how your SAG performs encryption operations. Valid values are: `Automatic` or `Manual`. |
| Transport | The transport type that SWIFT Module uses to initialize the server application to handle incoming responses from the SWIFT Network. In this case, specify `RAHA`. |

**Note:** The values specified for all properties in the table, except for the SAGMessagePartner and Transport properties, are used by the sample services. For information about the sample services, see *webMethods SWIFT Module Samples Guide*.

5 Click **Save** to save all configuration settings.

## Configuring Trading Networks Information

### Task 3 Configure Trading Networks information

SWIFT Module integrates with Trading Networks to process incoming requests. Therefore, you must define certain properties within Trading Networks.

1 **Define Trading Partner Profiles.** In Trading Networks, define trading partner profiles for yourself and for the all financial institutions with whom you want to exchange messages and files. For more information about defining trading partner profiles for

use with SWIFT Module, see "About Trading Partner Profiles" on page 80.

2 **Define TN Document Types.** TN document types are definitions that tell Trading Networks how to identify the incoming SNL request primitives and which processing rules to apply when processing the document. You must create a TN document type for each type of request that you handle.

When SWIFT Module receives a request, it invokes a Trading Networks service to recognize the incoming requests (SWIFTNet primitives) for server applications. Trading Networks matches the incoming request to one of the TN document types that you created and extracts the information indicated by the attributes specified in the TN document type.

When you define a TN document type, specify which root tag in the SNL primitive that the TN document type must match.

To create an internal TN document type do the following:

a Follow the instructions for creating document types in Trading Networks as described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

b Select **XML**.

c In the **Name** field on the Document Type Details screen, type the name you want to assign to the internal TN document type.

d In the **Description** field, type a description for the internal TN document type.

e On the Identify tab, in the **Root Tag** field, type the value of the root tag of your internal document, for example, `HandleFileRequest`.

**Note:** You can modify one of the sample TN document types included in the SWIFT Module samples. For information about using samples, see *webMethods SWIFT Module Samples Guide*.

3 **Create Mapping Services.** A mapping service defines the response to return to the client. You must create a mapping service for each type of request you handle. SWIFT Module provides sample services in the SWIFT Module samples that you can use as is or as a model for creating new services. For information about SWIFT Module sample services, see *webMethods SWIFT Module Samples Guide*.

4 **Define Processing Rules.** Processing rules enable you to process incoming requests (SNL primitives) for SWIFTNet server applications by invoking the mapping services that you create. Assign a processing rule to the TN document type to specify how Trading Networks processes requests, including invoking specified processing services. You must create a processing rule for each type of request you handle. To create a processing rule do the following:

a Follow the instructions for creating processing rules in Trading Networks as described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

b In the **Name** field on the Processing Rule Details screen, type a name for the processing rule.

c In the **Description** field, type a description for the processing rule.

d On the Criteria tab, select the TN document type associated with the processing rule.

e While adding a new action to the processing rule, select the **Execute a Service** check box and specify the service that you want this processing rule to execute.

To function properly, this service must meet the following requirements:

- The input/output signature must conform to the specification wm.swiftnet.server.doc:SWIFTNetServerSideProcessingRule in the WmSWIFTNetServer package.
- The output must contain a string *xmlResponse*. This is the response to the incoming request that the application server returns to the client.

Using Designer, verify that your processing rule complies with this criteria:

On the Input/Output tab of the mapping service, in the **Specification Reference** field, specify the SWIFTNetServerSideProcessingRule document.

f Follow the instructions to complete the creation of a processing rule as described in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

You can use or modify one of the sample server processing rules included in the SWIFT Module samples. For information about samples, see *webMethods SWIFT Module Samples Guide*. You can also create your own processing rules.

5 After completing the previous three configuration tasks, your server application is ready to receive requests and send responses.

## Step 2: Prepare the Client to Handle Requests

To prepare your client application to receive and respond to requests using the SWIFTNet component of SWIFT Module, you must complete the following steps:

1 Install an RA Client. Install an RA client on the same machine as Integration Server. The RA client enables the SWIFTNet component to communicate with SAG and SNL through RAHA. To obtain an RA client, contact SWIFT.

2 Configure the client environment information.

a From Integration Server Administrator, select **Adapters> SWIFT**.

b On the SWIFT Module home page, select **SWIFTNet Client Configuration > Edit**.

c On the Edit SWIFTNet Client Configuration screen, in the SWIFTNet Client Environment Information section, set the following environment properties according to the properties set for the RA client:

| For this property... | Specify... |
|---|---|
| SWNET_CFG_PATH | The cfg folder of the RA instance in your system, for example, c:\SWIFTAlliance\RA\Ra1\cfg\ |
| SystemRoot | The system root folder. The value of this parameter depends on your operating system, for example, c:\windows. |
| SWNET_BIN_PATH | The lib folder of the RA instance in your system, for example, c:\SWIFTAlliance\RA\lib. |
| SWNET_HOME | The RA home folder, for example, c:\SWIFTAlliance\RA. |
| RMI Port | The Remote Method Invocation (RMI) port where the remote process listeners will be bound in the Java Naming and Directory Interface (JNDI) tree, for example, `10985`. |

d In the SWIFTNet Client SAG Connection Configuration section, set the following properties to define how to connect to SAG using the RAHA transport:

| For this property... | Specify... |
|---|---|
| SAGMessagePartner | The Client message partner defined in SAG. |
| client_pki_profile | The user name of the configured profile used in opening a security context with SAG for sending the request to SWIFT Network. |
| client_pki_password | The password associated with the user name of the client PKI profile defined in your SAG. This password is used to unlock the Client PKI profile. |
| cryptoMode | `Automatic` or `Manual` to specify how your SAG performs encryption operations. The default value is `Automatic`. |
| requestor | The SWIFT BIC of the partner from where the message originates. Valid values are: `o=`*bic*`, o=swift` |
| responder | The SWIFT BIC of the partner that serves the request sent by the SWIFTNet client. Valid values are: `o=`*bic*`, o=swift` |
| service | The default service to be used by SAG for processing the request from the client. |
| encryptDN | The Distinguished Name to be used for encryption. Valid values are: `cn=`*encryptCN*`, o=`*bic*`, o=swift.` |

| For this property... | Specify... |
|---|---|
| userDN | The Distinguished Name to be used for sign, encryption, and authorization operations. Valid values are: `cn=`*encryptCN*`, o=`*bic*`, o=swift` |
| Sign, Decrypt, and Authorization | Defines whether to use the security context opened during server initialization. Valid values: `True` or `False`. |
| ReceptionFolder | The default folder to receive incoming files. If the specified folder does not exist, it is created. When this field is blank, the default value is: *Integration Server_directory*\packages\WmSWIFTNetServer\pub\SWIFTNetReceptionFolder. |
| physicalName | The absolute path of the file to be transferred on the machine on which SAG is running, for example, c:/temp/log.log |
| logicalName | The logical name of the file to be used during the transfer, for example, `sample`. |
| SwTransferEP | Default transfer endpoint of the remote file handler. If this field is specified, the value must match a remote file handler endpoint running on the same machine as Integration Server. For information about invoking the remote file handler, see "Step 3: Invoke the Remote File Handler" on page 159. |
| SwEventEP | The file transfer event endpoint where file transfer events (associated with file transfers) should be sent, for example, `File_Status_Event_EP`.<br>**Note:** When SnF Pull or Push is used to send the file transfer events to the application server host Integration Server, the value specified for this field must be the same on both the client and the application server. |
| Transport | `RAHA` |

e Click **Save** to save all configuration settings.

**Note:** The values specified for all properties in the table, except for the **SAGMessagePartner** and **Transport** properties, are used by the sample services. For information about the sample services, see *webMethodsSWIFT Module Samples Guide*.

3 Invoke the wm.swiftnet.client.services:swArguments service.

Before exchanging any primitives with SAG using the SWIFTNet component, the client application must invoke the wm.swiftnet.client.services:swArguments service in theWmSWIFTNetClient package at least once. The only parameter required to be passed to this service is SAGMessagePartner, which is the message partner defined as the "Client" in your SAG during configuration.

## Step 3: Invoke the Remote File Handler

You must invoke the Remote File Handler to transfer the files that reside in your system.

### To run the Remote File Handler

1 Run the following command from the RA installation bin directory:

```
RA_Installation_Directory\RA\bin\swiftnet.bat init -S
ra_instance
```

where *ra_instance* is the instance of the Remote API on your system, for example, `RA1`.

2 Start the swfa_handler with the command line arguments as follows:

```
swfa_handler hostName:portNumber:[ssl] transferEndpoint
[Process ID]
```

The *hostName* is the name of the host where SAG/SNL is installed as in the following examples:

```
swfa_handler snlhost:48003:ssl MyUniqueEndpoint 23450
swfa_handler snlhost:48003 MyUniqueEndpoint 23450
swfa_handler snlhost:48003 MyUniqueEndpoint
```

**Note:** The `swfa_handler` is present in the *RA_HOME*\bin directory.

# 16 Configuration Steps for InterAct and FileAct Messaging Services over SAG MQHA

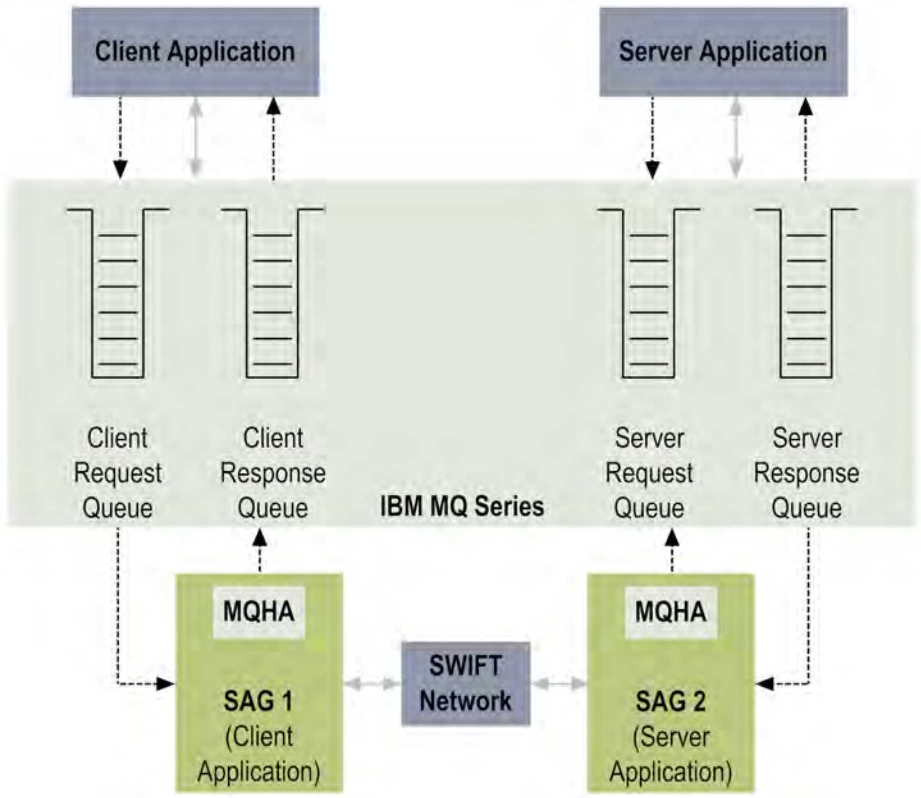
# Overview

The SWIFTNet component of SWIFT Module supports two types of transport for InterAct and FileAct messaging services over SWIFT Alliance Gateway (SAG): the Remote API Host Adapter (RAHA) and the MQ Host Adapter (MQHA).

The MQHA enables your SWIFTNet component client and server applications to communicate with SAG through IBM WebSphere MQ.

This chapter describes how to prepare your server or client application to exchange messages and files over SAG using the MQHA transport. For more information about the SWIFT messaging services and the two types of transport, see "SWIFTNet Component" on page 36. For information about how to configure the RAHA transport, see Chapter 15, "Configuration Steps for InterAct and FileAct Messaging Services over SAG RAHA".

The following diagram shows the response-request interaction between the client, server, and SAG when you use the MQHA transport.



The client and server must first register with the SWIFT Network. SAG must be installed on both the client and the server. In the diagram above, the applications are designated as SAG 1 for the client and SAG 2 for the server.

1 The client registers itself with the SWIFT Network through SAG 1.

2 The client application puts a message in the client request queue.

3 SAG 1 takes the message from this queue and determines the destination for the message is the server.

4 SAG 1 sends this message across the SWIFT Network to SAG 2.

5 On receipt of this message, SAG 2 puts the message in the server request queue of the server application.

6 The server application takes the message from the server request queue and processes the message as required. Then, the server application puts the reply message into the server response queue.

7 SAG 2 takes this message, determines its destination, and sends it across the SWIFT Network to SAG 1.

8 On receipt of this message, SAG 1 puts it in the client response queue.

9 The client takes this message from this queue.

To prepare your server application or your client application for the response-request interaction with SAG, you must complete the steps described in the following sections.

# Step 1: Prepare the Server to Handle Requests

To prepare your server application to exchange messages and files over SAG using MQHA, you must complete the following stages of configuration:

| Task | Description |
|---|---|
| 1 | Configure SAG to communicate with your SWIFT Module through the IBM MQ request and reply queues. |
| 2 | Configure the SWIFTNet component. |
| 3 | Configure Trading Networks information. |

## Configuring SWIFT Alliance Gateway

### Task 1 Configure SWIFT Alliance Gateway

1 **Define MQ queues as per SWIFT Alliance Gateway MQHA configuration.** To handle the exchange of requests between your server application and SAG, you must configure the following types of queues on the SAG side: client request queue, client reply queue, server request queue, and server reply queue.

- The client request and reply queues are used by the client application.

  The server request and reply queues are used by the server application, as described in the “Overview” on page 94.

  In SAG, you must associate these queues with the applications that will process messages from these queues.

2 **Configure How to Handle Messages for Client Queues.** Configure how SAG retrieves messages from the client request queue and how it puts response messages into the client reply queue.

3 **Configure How to Handle Messages for Server Queues.** Configure how SAG puts messages into the server request queue and how it gets response messages from the server reply queue.

For more information about completing these steps, see *SWIFT Alliance Gateway MQ Host Adapter Configuration Guide* and *SWIFT Alliance Gateway Operations Guide*.

## Configuring the SWIFTNet Component

### Task 2 Configure the SWIFTNet component

1 Install **webMethods WebSphere MQ Adapter**. If you have not already installed WebSphere MQ Adapter, see *webMethods WebSphere MQ Adapter Installation and User's Guide* for installation instructions.

2 **Set up the MQ queues.** In WebSphere MQ Adapter, configure the adapter connection properties for the MQ server request and server reply queues to handle incoming requests from the SWIFT Network. For complete information about configuring adapter connections, see *webMethods WebSphere MQ Adapter Installation and User's Guide*.

On the adapter's Configure Connection Type screen, specify the MQHA-specific information for connection settings for the queues as follows:

a Specify the settings from SWIFT Module to the IBM MQ Server Request Queue:

| For this property... | Specify... |
|---|---|
| Queue Manager Name | `MQHA.SAG.QM` |
| Host Name | The name of the server on which IBM WebSphere MQSeries is running. |
| Server Connection Channel | `MQHA.CHANNEL` |
| Queue Name | `MQHA.SERVER.REQUEST` |

b Specify the settings from IBM MQ Server Reply Queue to SWIFT Module:

| For this property... | Specify... |
|---|---|
| Queue Manager Name | `MQHA.SAG.QM` |
| Host Name | The name of the server on which IBM WebSphere MQSeries is running. |
| Server Connection Channel | `MQHA.CHANNEL` |
| Queue Name | `MQHA.SERVER.REPLY` |

3 **Create the MQ listener and configure the listener notification.** You must create a single-queue listener to listen to all incoming requests from the SWIFT Network. This listener is associated with the connection configured for the IBM MQ Server Request Queue.

You must also configure an asynchronous WebSphere MQ Adapter notification for the MQ listener and define a document trigger for the notification document. This trigger calls the wm.swiftnet.server.mq.inbound.handleSWIFTRequest service that extracts the required values from the notification document and publishes the SNL primitive to Trading Networks. Configure the MQ listener notification document as described in step 6.

For instructions on how to create the listener and configure the listener notification, see *webMethods WebSphere MQ Adapter Installation and User's Guide*. For more information about triggers, see *Publish-Subscribe Developer's Guide*.

4 **Create the MQ request/reply client service.** This service sends and receives messages from the SAG client queues configured in MQ. This service is also used to start and stop the server application, since the primitives required for starting and stopping a server application are communicated to SAG through client queues. For detailed instructions on creating the request/reply service, see *webMethods WebSphere MQ Adapter Installation and User's Guide*.

You must specify the following SWIFT-specific properties for this service:

a The **Wait Interval** must have a value greater than 90 seconds. For more information, see the SWIFT documentation.

b Define the **msgHeader** properties for the **MQMD Header**:

| For this property... | Specify... |
|---|---|
| ReplyToQueueMgr | `MQHA.SAG.QM` (default value for `MQHA`) |
| ReplyToQ | `MQHA.CLIENT.REPLY` |
| MsgType | `Datagram` |
| Format | `MQSTR` |

5 **Create the MQ put server service.** This service sends messages to the SAG response queue configured in MQ. For instructions on how to create the put service, see *webMethods WebSphere MQ Adapter Installation and User's Guide*.

Define the **msgHeader** properties for the **MQMD Header**:

| For this property... | Specify... |
|---|---|
| Format | `MQSTR` |
| MsgType | `REPLY` |
| Feedback | `MQFB_APPL_FIRST` (or `65536`) |

6 **Configure the server application.** Define how the server application connects to SAG using the MQHA transport:

a From Integration Server Administrator, click **Adapters> SWIFT**.

b In the SWIFT Module home page, click **SWIFTNet Server Configuration > Edit.**

c In the **SWIFTNet Server SAG Connection Properties** section on the SWIFTNet Server Configuration screen, set the following properties.

**Important!** When using MQHA, leave the default values in the fields in the **Remote Process Connection Configuration** and **Server Environment Information** sections. You must configure the fields in these sections only when you use the RAHA transport.

| For this property... | Specify... |
|---|---|
| SAGMessagePartner | The Server message partner defined in SAG. |
| server_pki_profile | The user name of the configured profile that is used when opening a security context with SAG for sending the request to the SWIFT Network. |
| server_pki_password | The password associated with the user name of the server PKI profile defined in your SAG, used to unlock the Server PKI profile. |
| userDN | The Distinguished Name to use for sign, encryption, and authorization operations, for example, `cn=`*encryptCN*`, o=`*bic*`, o=swift` |
| encryptDN | The Distinguished Name used for encryption, for example, `cn=`*encryptCN*`, o=`*bic*`, o=swift` |
| Sign, Decrypt, and Authorization | Defines if the security context opened during server initialization should be used for sign on, decryption, or authorization. Valid values are `True` and `False`.<br>At least one field must be set to `True`. |
| AllFileEvents | Indicates whether to populate Sw:SubscribeFileEventRequest primitive exchanged during server initialization.<br>■ `True`—Default. SWIFT Module receives all events generated by the SAG file sub-system during file transfer.<br>■ `False`—SWIFT Module receives only state transition events. |

| For this property... | Specify... |
|---|---|
| FullFileStatus | Indicates whether to populate Sw:SubscribeFileEventRequest primitive exchanged during server initialization.<br>■ `True`—Default. SWIFT Module receives full details for each event report generated by the SAG system.<br>■ `False`—SWIFT Module does not receive full details for each event report generated. |
| SwEventEP | The file transfer event end point to which file transfer events are posted by your SAG during FileAct operations. This value is used to populate the `Sw:SubscribeFileEventRequest` primitive exchanged during server initialization. |
| ReceptionFolder | The default folder for incoming files. If the specified folder does not exist, it is created. When this field is blank, the reception folder is created in: *Integration Server_directory*\packages\WmSWIFTNetServer\pub\SWIFTNetReceptionFolder. |
| SwTransferEP | The default transfer endpoint of the remote file handler. This property is optional on Windows systems and required on UNIX systems. If SwTransferEP is specified, the value must match a remote file handler endpoint running on the same machine as Integration Server. For information on invoking the remote file handler, see "Step 3: Invoke the Remote File Handler" on page 159. |
| cryptoMode | `Automatic` or `Manual` to specify how your SAG performs encryption operations. |
| Transport | The transport type that SWIFT Module uses to initialize the application server to handle incoming responses from the SWIFT Network. The default value is `MQHA`. |
| MQ Request Reply Client Service | You can accept the default service or specify your own. For more information about this property, see step 4.<br>The wm.swiftnet.config.sample.mq.services:getAndRecieveService sample service is the default. For information about this service, see *webMethods SWIFT Module Samples Guide*.<br>If you accept the default service, the default client queue and queue manager settings are used. For more information on the queue settings, see *SWIFT Alliance Gateway MQ Host Adapter Configuration Guide*. |

| For this property... | Specify... |
|---|---|
| MQ Put Server Service | You can accept the default service or specify your own. For more information about this property, see step 5.<br>The wm.swiftnet.config.sample.mq.services:serverResponse1 sample service is set as the default and is included with the SWIFT Module samples. For information about this service, see *webMethods SWIFT Module Samples Guide*.<br>The default server queue and queue manager settings are used if you accept the default service. For more information on server queue settings, see *SWIFT Alliance Gateway MQ Host Adapter Configuration Guide*. |
| MQ Listener Notification Document | The MQ listener notification you created in step 3.<br>This notification document will be published for an incoming request at the MQ server request queue. |

Note: The values specified for all properties in the table, except for the Transport, MQ Request Reply Client Service, MQ Put Server Service, and MQ Listener Notification Document properties, are used by the sample services. For information about the sample services, see *webMethods SWIFT Module Samples Guide*.

7 Click Save to save all configuration settings.

## Configuring Trading Networks Information

### Task 3 Configure Trading Networks information

1 Define Trading Partner Profiles. Define trading partner profiles for your enterprise and each partner with whom you exchange files or messages. For more information about defining trading partner profiles for use with SWIFT Module, see "About Trading Partner Profiles" on page 80.

2 Define TN Document Types. TN document types are definitions that tell Trading Networks how to identify the incoming SNL request primitives and which processing rules to apply when processing the document. You must create a TN document type for each type of request that you handle.

For information about creating TN document types, see step 2 in "Configuring Trading Networks Information" on page 154. For general information about using TN document types, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

3 **Create Mapping Services.** A mapping service defines the response to return to the client. You must create a mapping service for each type of request that you handle. SWIFT Module provides sample services in the SWIFT Module samples that you can use as is or as a model to build your own services. For information about the sample services, see *webMethods SWIFT Module Samples Guide*.

4 **Define Processing Rules.** Processing rules enable you to process incoming requests. Assign a processing rule to a TN document type so that Trading Networks knows how to process the request, including which processing service to invoke. You must create a processing rule for each type of request you handle.

For information about processing rules, see step 4 in "Configuring Trading Networks Information" on page 154. For general information about using processing rules, see the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.

## Step 2: Prepare the Client to Handle Requests

To prepare your client application to receive and respond to requests from SAG over the IBM WebSphere MQ transport, you must complete the following steps:

1 **Set up the MQ Queues.** In WebSphere MQ Adapter, configure the adapter connection properties for the MQ server request queue and server reply queue to handle incoming requests from the SWIFT Network. For complete information about configuring adapter connections, see *webMethods WebSphere MQ Adapter Installation and User's Guide*.

- Specify the MQHA-specific information for the connection settings from SWIFT Module to IBM MQ Client Request Queue:

| For this property... | Specify... |
|---|---|
| Queue Manager Name | `MQHA.SAG.QM` |
| Host Name | The name of the server on which IBM WebSphere MQSeries is running. |
| Server Connection Channel | `MQHA.CHANNEL` |
| Queue Name | `MQHA.CLIENT.REQUEST` |

- Specify the MQHA-specific information for the connection settings from IBM MQ Client Reply Queue to SWIFT Module:

| For this property... | Specify... |
|---|---|
| Queue Manager Name | `MQHA.SAG.QM` |
| Host Name | The name of the server on which IBM WebSphere MQSeries is running. |
| Server Connection Channel | `MQHA.CHANNEL` |

| For this property... | Specify... |
|---|---|
| Queue Name | `MQHA.CLIENT.REPLY` |

2 Define how the client application connects to SAG using MQHA transport as follows:

a In Integration Server Administrator, click **Adapters> SWIFT**.

b In the SWIFT Module home page, click **SWIFTNet Client Configuration** > **Edit**.

c In the **SWIFTNet Client SAG Connection Configuration** section, set the following properties to define how to connect to SAG using MQHA transport:

**Important!** When using MQHA, leave the default values in the fields in the **SWIFTNet Client Environment Information** section. Configure the fields in this section only when using RAHA transport.

| For this property... | Specify... |
|---|---|
| SAGMessagePartner | The Client message partner defined in SAG. |
| client_pki_profile | The user name of the configured profile that is used for opening a security context with SAG (to send the request to SWIFT Network). |
| client_pki_password | The password associated with the user name of the client PKI profile defined in your SAG. This is used to unlock the Client PKI profile. |
| cryptoMode | `Automatic` or `Manual` to specify how your SAG performs encryption operations. The default value is `Automatic`. |
| requestor | The SWIFT BIC of the partner from where the message originates, for example, `o=`*bic*`, o=swift` |
| responder | The SWIFT BIC of the partner that will serve the request sent by the SWIFTNet client, for example, `o=`*bic*`, o=swift` |
| service | The default service to be used by SAG for processing the request from the client. |
| encryptDN | The Distinguished Name to be used for encryption, for example, `cn=`*encryptCN*`, o=`*bic*`, o=swift` |
| userDN | The Distinguished Name to be used for sign, encryption, and authorization operations, for example, `cn=`*encryptCN*`, o=`*bic*`, o=swift` |
| Sign, Decrypt, and Authorization | `True` or `False` to define if the security context opened during server initialization should be used for sign on, decryption or authorization. |

| For this property... | Specify... |
|---|---|
| ReceptionFolder | The default folder to receive incoming files. When this field is blank, the folder is created in the following path:<br>*Integration Server_directory*\packages\WmSWIFTNetClient \pub\SWIFTNetReceptionFolder.<br>If the specified folder does not exist, it is created. |
| physicalName | The absolute path of the file (to be transferred), for example, c:/temp/log.log.<br>For more information, see *SWIFTNet Service Design Guide* and *SWIFTNet Link Interface Specification*. |
| logicalName | The logical name of the file that should be used during the transfer, for example, sample. |
| SwTransferEP | The default transfer endpoint of the remote file handler. If SwTransferEP is specified, the value must match a remote file handler endpoint running on the same machine as Integration Server.<br>For information on invoking the remote file handler, see "Step 3: Invoke the Remote File Handler" on page 159. |
| SwEventEP | The file transfer event end point where file transfer events should be sent, for example, `File_Status_Event_EP`.<br>**Note:** When SnF Pull or Push is used to send the file transfer events to the server application host Integration Server, the value specified for this parameter must be the same on both the client and the server application. |
| Transport | `MQHA`. This is the default value. |
| MQ Request Reply Client Service | You can accept the default service or specify your own. For more information about the MQ request/reply client service, see step 4 in "Configuring the SWIFTNet Component" on page 164.<br>The default service is wm.swiftnet.config.sample.mq.services: getAndRecieveService, included in the SWIFT Module samples. For information about the sample services, see *webMethods SWIFT Module Samples Guide*.<br>If you accept the default service, the default client queue and queue manager settings are used. For more information on the queue settings, see *SWIFT Alliance Gateway MQ Host Adapter Configuration Guide*. |

**Note:** The values specified for all properties in the table, except for the **Transport** and **MQ Request Reply Client Service** properties, are used by the sample services. For information about the sample services, see *webMethods SWIFT Module Samples Guide*.

d Click **Save** to save all configuration settings.

# Step 3: Initialization and Request-Time Operations for Your Client or Server Application

For an application to communicate with SAG, it must first register itself either as a client or a server. Then the application can be initialized to interact with SAG and perform the request-time operations required for the message exchange.

## Initializing the Client or Server Application

To initialize the client and the server application, first configure the input fields for the initialization request in SWIFT Module, as described in "Step 1: Prepare the Server to Handle Requests" on page 163 and "Step 2: Prepare the Client to Handle Requests" on page 169.

Initializing the client and server applications involves the exchange of primitives, as required by SWIFT. You can find sample services that demonstrate this primitive exchange for each type of application in the SWIFT Module samples.

During initialization, a security context is established between the client and server applications with the specified **MessagePartner**. The security context for the **userDN** parameter is stored in the shared cache with **MessagePartner** as its key value pair. The security context is fetched from the shared cache when sending or receiving requests from SWIFT Network. Once the server and client initialization is complete, the server can handle all incoming requests and the client can send requests to SWIFT Network.

## Request-Time Operations

### Client Application

Once the security context has been established, the client application is ready to send an InterAct request to SWIFT.

**To send an InterAct request to SWIFT:**

1. Create an exchange request and an associated envelope. The exchange request consists of two content parts:
   - *xmldata*—The payload of the request.
   - *sagenv*—The envelope required by SAG for processing the request.
2. Call the request/reply service for the MQ transport with the request.
3. After receiving the response from SWIFT Network, the client submits the response message to Trading Networks. The ExchangeResponse consists of two content parts:
   - *xmldata*—The payload of the request.
   - *sagenv*—The envelope required by SAG for processing the request.

SWIFT Module provides sample services that demonstrate how to send an InterAct request to SWIFT. For information about the SWIFT Module sample services, see *webMethods SWIFT Module Samples Guide*.

## Server Application

Once the Server application is initialized with the identified MessagePartner, it handles all incoming requests as described in the following procedure:

1. SAG puts all requests for this server application message partner in the server request queue. Configure an asynchronous listener notification for the server request queue as described in step 6 in "Configuring the SWIFTNet Component" on page 164. Once the message is received by the listener, the notification is triggered and invokes a handleRequest service that extracts the incoming request from the notification document.
2. The handleRequest service creates a HandleRequest message and submits this message to Trading Networks. The HandleRequest message consists of the following content parts:
   - *xmldata*—The payload of the request.
   - *sagenv*—The envelope required by SAG for processing the request.
   - *msgId*—The message ID from the message queue, used for associating the response message with the request message that will be returned to SAG.
3. After submitting the request to Trading Networks, the configured processing rule is triggered based on the document type. When the mapping of the request/reply service is completed, the reply primitive is submitted to Trading Networks.

4 The wm.swiftnet.server.mq.trp.respond service puts the response document in the server reply queue. MQHA on SAG takes the server application response and routes the response to the requesting client through the SWIFT Network.

This service uses the MQ Put service that you configured for **MQ Put Server Service** as described in step 5 in "Configuring the SWIFTNet Component" on page 164. The HandleResponse message consists of the following content parts:

- *xmldata*—The payload of the request.
- *sagenv*—The envelope required by SAG for processing the request.
- *correlationId*—The correlation ID for the response message. This is set as MQDM headers and sent back to SAG. SAG uses the correlation ID to associate the reply message with the request message with the same *msgId*, and then returns the response to the client.
- *msgId*—The message ID from the message queue, used to associate the response message with the request message that is returned to SAG.

## Termination

After completing the request/response message exchange, you can invoke the termination request service to stop the client application and the server application. After terminating the server application, the security context for the server message partner is removed from the shared cache. For sample services that demonstrate how to close the client and server applications, see *webMethods SWIFT Module Samples Guide*.

For more information about the primitives exchanged during application termination, see the SWIFT documentation.

# 17 Using FTA to Transfer Files over SWIFTNet

# Overview

The SWIFT File Transfer Adapter (FTA) automates file transfer from SWIFT Module to other parties over SWIFTNet. The SWIFT File Transfer Adapter (FTA) manages the SWIFT Alliance Gateway (SAG) directories for file transfer related tasks:

- Places the data file in the SAG output directory for transfer over SWIFTNet.
- Creates a companion file to the data file and sends it to the SAG output directory.
- Monitors the SAG input directory for processing status reports generated by FTA.

# Placing a Data File in the SAG Output Directory

SWIFT Module allows you to use a custom utility service to place data files that you want to transfer over SWIFTNet into the SAG output directory. The FT-Interface provided by SWIFT uses the FTA configuration parameters to pick up the file from the SAG output directory and send it over SWIFTNet. You can configure the FTA configuration parameters using the SWIFT FT-Interface. For information how to configure FTA, see *SWIFT Alliance Access 6.1 File Transfer Interface Guide*.

# Creating a Companion File

The FTA allows you to create a companion parameter file and transfer it with the data file from SWIFT Module to the SAG host that processes the data files. You can override the FTA configuration parameters with your user-defined parameters by running the wm.swiftnet.client.transport.FTA:generateCompanionFile service in Designer. The service generates a companion parameter file with the same file name as the related data file, followed by the .par extension.

## Companion Parameter File Data Structure

The data in the companion parameter file is structured according to the ParametersSchema.xsd XML schema file. After you install SAG, this file is created in the *SAG_HOME* \data directory, where *SAG_HOME* is the SAG installation directory.

After you install SWIFT Module, you can find the corresponding TN document types for the companion parameter files in the following directory:
*Integration Server_directory*\packages\WmSWIFTNetClient\config\FTADocTypes.dat

# Generating Data File Processing Status Reports

You can generate reports about the processing status of data files by configuring FTA emission and reception profiles. SWIFT Module scans the SAG input directory for report files and then submits the reports to Trading Networks in XML format.

### To scan the SAG input directory for report files and submit the reports to Trading Networks

1. Import the document types for report XML files in Trading Networks (follow the instructions for importing document types in the Trading Networks administration guide for your release. See "About this Guide" for specific document titles.)
2. Select the following file: *Integration Server_directory*\packages\WmSWIFTNetClient \config\FTADocTypes.dat.

   The TN document types are imported and listed on the Available Items screen.
3. In Designer, run wm.swiftnet.client.transport.FTA:scanForReports. This service does the following:
   - Scans the SAG input directory for report files.
   - Invokes wm.swiftnet.client.transport.FTA:submitToTN to submit the report to Trading Networks.

## Report File Data Structure

The content of a report file depends on the reason why it is generated. Content within a report file is structured using XML according to the ReportSchema.xsd XML schema file. This schema file is located in the SAG data directory after installing SAG. FTA generates the following report files:

| Type | File Extension | XML Element | Reason |
|---|---|---|---|
| Success | .ok | `<Success>` | A file was successfully sent or received. |
| Delivery Notification | .dlv | `<Delivery>` | FTA received a delivery notification for a file. |
| Error | .err | `<Error>` | An error occurred during the incoming or outgoing file transfer. |
| Authorization or Refusal | .arn | `<AuthNotif >` | FTA received an authorization or refusal notification, which is also included in the file. |

# A Services

## WmFIN Package

The WmFIN package contains services used to implement and support the SWIFT FIN-compliant functionality of webMethods SWIFT Module. This package provides core services for processing and transporting MX and MT messages, as well as services for handling inbound notifications from SWIFT Alliance Access to webMethods SWIFT Module. This package contains the following folders:

| Folder | Contains services to... |
|---|---|
| wm.casmf.init Folder | Perform initialization routines for CASmf. |
| wm.casmf.trp Folder | Send and receive messages using CASmf. |
| wm.casmf.util Folder | Retrieve property values specified in the *Integration Server_directory*\packages\WmFIN\config\wmcasmf.cnf file. |
| wm.fin.bic Folder | Derive or validate BICPlusIBAN information. |
| wm.fin.dev Folder | Install and configure new SWIFT FIN messages during design-time. |
| wm.fin.dfd Folder | Load and use the FIN Data Field Dictionary (DFD). |
| wm.fin.doc Folder | Define the document structures that represent particular sections of SWIFT FIN messages, such as the header and trailer structures and their fields, and the generic structure definitions for incoming and outgoing SWIFT FIN messages. |
| wm.fin.format Folder | Define the record definitions that describe the record structures for the trailer section (block 5) of a SWIFT FIN message. |
| wm.fin.init Folder | Initialize or de-initialize FIN packages on startup and shutdown of Integration Server. |
| wm.fin.map Folder | Provide easy frameworks for creating the header and trailer sections of SWIFT FIN messages for outbound (to be sent to SWIFT) messages. |
| wm.fin.marketPractice Folder | Support Market Practices for some Category 5 messages. The services in this package are for internal use only. |
| wm.fin.rules Folder | Use utility functions to implement network validation rules. |
| wm.fin.sepa Folder | Derive or validate data against the SEPA Routing directory. |
| wm.fin.transport Folder | Exchange messages with SWIFT using Automated File Transfer (AFT) and MQSeries. |

| Folder | Contains services to... |
|---|---|
| wm.fin.trp Folder | Provide single-point access to send and receive SWIFT FIN messages. |
| wm.fin.utils Folder | Use generic utility services providing various functionality. |
| wm.fin.validation Folder | Facilitate the validation of a SWIFT FIN message. |
| wm.sdk.fin Folder | Support the conversion of MT messages into a flat file or XML format, as needed using SDK related Java services, XSDs, and IS document types. |
| wm.unifi Folder | Validate an MX message against the SWIFT generic rule book. |
| wm.xmlv2.dev Folder | Create Trading Networks data for a particular message type. |
| wm.xmlv2.doc Folder | Define a TN document type that is used as the TPA document. |
| wm.xmlv2.notifications Folder | Handle incoming delivery notifications. |
| wm.xmlv2.process Folder | Apply processing rules to documents exchanged over SAA. |
| wm.xmlv2.transport Folder | Submit DataPDU input in XML format to Trading Networks for further processing of the bizdoc. |
| wm.xmlv2.utils Folder | Contains utility services for message encoding. |

## wm.casmf.init Folder

The services in this folder perform initialization routines for CASmf.

### wm.casmf.init:shutdown

Unregisters the application with CASmf.

### wm.casmf.init:startup

Registers the application with CASmf *Input Name.*

## wm.casmf.trp Folder

The services in this folder send and receive messages using CASmf.

## wm.casmf.trp:casmfSendReceiveSchedule

Run as a scheduled job. This service does the following:

1. Sends all the outbound messages to CASmf. It uses the value specified for *wm.casmf.send.mapid* in the wmcasmf.cnf file (located in the folder, *Integration Server_directory*\packages\WmFIN\config).
2. Retrieves the incoming messages from CASmf using the value specified for *wm.casmf.receive.mapid* in the wmcasmf.cnf file.
3. Publishes the received messages to Integration Server and webMethods Broker for processing by the WmFIN package service, wm.fin.trp:receive.

## wm.casmf.trp:processOutboundMessage

This service is invoked during publishing an outbound message, when the *transport* parameter in the message TPA is set to `CASmf`. It writes the SWIFT message with a unique file name to the directory specified by the *wm.casmf.send message.folder* property in the wmcasmf.cnf file. (The wmcasmf.cnf file is located in the directory *Integration Server_directory*\packages\WmFIN\config\.)

### Input Parameters

| | |
|---|---|
| *wm.fin.doc:FINOutboundMessage* | **Document** Document subscribed by this service when the *transport* parameter in the message TPA is set to `CASmf`. |

### Output Parameters

None.

## wm.casmf.trp:sendAndReceive

The wm.casmf.trp:casmfSendReceiveSchedule service invokes this service, after it is done its processing. This service sends and receives messages from CASmf. It accumulates messages received from CASmf into a String.

### Input Parameters

None.

### Output Parameters

| | |
|---|---|
| *receivedFINMessages* | **String List** List of messages received from CASmf. |

## wm.casmf.trp:CASmfOutboundTrigger

This trigger processes outbound SWIFT FIN messages that must be sent via CASmf. It then invokes wm.casmf.trp:processOutboundMessage to process the outbound SWIFT message.

## wm.casmf.util Folder

This folder contains utility services for retrieving property values specified in the *Integration Server_directory*\packages\WmFIN\config\wmcasmf.cnf.

## wm.casmf.util:getOutboundMessageFolder

This service retrieves the value for the wm.casmf.send.message.folder property specified in *Integration Server_directory*\packages\WmFIN\config\wmcasmf.cnf file. This is the folder in which all outbound SWIFT FIN messages to CASmf are stored prior to sending them to CASmf.

### Input Parameters

None.

### Output Parameters

| | |
|---|---|
| *folder* | String Value specified for the wm.casmf.send.message.folder property in the wmcasmf.cnf file. |

## wm.fin.bic Folder

This folder contains BICPlusIBAN-related services used to derive or validate BICPlusIBAN information.

## wm.fin.bic:deriveBICfromIBAN

This services uses the IBAN provided as input and derives a valid BIC code based on the logic specified by SWIFT. The service does the following:

1 Retrieves the Country Code and the National Code from the IBAN and validates as follows:

   a Retrieves the country code (the first two characters of the IBAN).

   b Finds the record with the corresponding *IBAN Country_Code* in the IS list.

   c Uses the bank identifier position and IBAN national ID length fields to establish the start position and the length of the data to extract within the IBAN.

   d Applies these parameters to the IBAN to extract the *IBAN_National_Id*.

2 Retrieves the BIC issued with the IBAN, as follows:

   a Searches the *IBAN_Country_Code* and *Unique_IBAN_National_Id* fields that contain the values from step 1.

   b The record retrieved from the *IBAN_BIC_Code* and *IBAN_ Branch_Code* fields contains the BIC to be used together with the IBAN.

### Input Parameters

| | |
|---|---|
| *IBAN* | String IBAN of the financial institution. |

### Output Parameters

| | |
|---|---|
| *Country_Code* | String Country code of the financial institution retrieved from the IBAN (the first two characters of the IBAN). |
| *IBAN_National_Id* | String The national identifier of the financial institution retrieved from the IBAN. |
| *IBAN_Country_Code* | String Country code prefix of the IBAN of the financial institution. |
| *Unique_IBAN_National_Id* | String IBAN National ID. For search purposes, the value is unique in the data file per *IBAN_Country_Code*. |
| *BIC_Code* | String BIC code of the financial institution, country, or location code). |
| *branchCode* | String Branch code of the financial institution that corresponds to the *BIC_Code*. |
| *IBAN_BIC_Code* | String The BIC code that corresponds to the IBAN. |
| *IBAN_Branch_Code* | String Branch code of the financial institution that corresponds to the *IBAN_BIC_Code*. |
| *Routing_BIC_Code* | String The routing or processing BIC to which the payment must be sent. |
| *Routing_Branch_Code* | String Branch code of the financial institution that corresponds to the *Routing_BIC_Code*. |
| *error* | String Specifies whether an error occurred. Valid values: `yes` and `no`. |
| *errorMessage* | String Specifies the error message if an error occurs. |

## wm.fin.bic:generateIBAN

This service generates the IBAN from the input parameters, based on the following logic:

1. In the IS list, it finds the IBAN structure for the country.
2. In the BI list, it finds the bank's national code and the BIC corresponding to the IBAN.
3. It constructs the BBAN (Basic Bank Account Number).
4. It constructs the IBAN by adding the country and check digits.

### Input Parameters

| | |
|---|---|
| *Country_Code* | String Country code of the financial institution retrieved from the IBAN (the first two characters of the IBAN). |
| *Institution_Name* | String Name of the financial institution. |
| *City_Heading* | String City in which the financial institution is located. |
| *Account_Number* | String Account number of the financial institution. |

### Output Parameters

| | |
|---|---|
| *IBAN* | String IBAN of the financial institution. |
| *BBAN* | String BBAN of the financial institution. |
| *errorMessage* | String Specifies the error message if an error occurs. |
| *error* | String Specifies whether an error occurred. Valid values: `yes` and `no`. |

## wm.fin.bic:getBICInfo

This service retrieves BIC information from the database based on the specified criteria.

### Input Parameters

| | |
|---|---|
| *code* | String BIC code of the financial institution. Specify a partial string using `%partial string%`. |
| *bicKey* | String BIC key of the financial institution. |
| *institution* | String Name of the financial institution. |
| *branch* | String Name of the financial institution's branch. |
| *city* | String City in which the financial institution is located. |

*modFlag* String BIC modifier flag for the financial institution, which identifies the BIC records added, updated, and deleted since the last update. Valid values:

- `A`. Addition.
- `U`. Unchanged.
- `M`. Modified.

*location* String Location of the financial institution.

*countryName* String Country in which the financial institution is located.

### Output Parameters

*count* String Specifies the number of BIC records returned in the search.

*bicInfo* Document Reference List BIC records specifying the search criteria.

*errorMessage* String Specifies the error message, if an error occurs.

*error* String Specifies whether an error occurred. Valid values: `yes` and `no`.

## wm.fin.bic:getBICPlusInfo

This service retrieves BICPlus information from the database, based on the specified criteria.

### Input Parameters

*code* String BIC code of the financial institution. Specify a partial string using `%partial string%`.

*bicKey* String BIC key of the financial institution.

*institution* String Name of the financial institution.

*branch* String Name of the financial institution's branch.

*city* String City in which the financial institution is located.

*modFlag* String BIC modifier flag for the financial institution, which identifies the BIC records added, updated, and deleted since the last update. Valid values:

- `A`. Addition.
- `U`. Unchanged.
- `M`. Modified.

*location* String Location of the financial institution.

*countryName* **String** Country in which the financial institution is located.

### Output Parameters

*count* **String** Specifies the number of BIC records returned in the search.

*IBANInfo* **Document Reference List** BIC records specifying the search criteria.

*errorMessage* **String** Specifies the error message if an error occurs.

*error* **String** Specifies whether an error occurred. Valid values: `yes` and `no`.

## wm.fin.bic:insertIBANList

This service imports the BICPlusIBAN list into the database.

### Input Parameters

*filename* **String** Fully qualified path and file name of the IBAN list to import, for example, c:\bic\sample\BIDELTA_20100202.txt.

### Output Parameters

*errorMessage* **String** Specifies the error message if an error occurs.

*error* **String** Specifies whether an error occurred. Valid values: `yes` and `no`.

## wm.fin.bic:insertISList

This service imports the IS list into the database.

### Input Parameters

*filename* **String** Fully qualified path and file name of the IS list you want to import, for example, c:\bic\sample\IS_20071103.txt.

### Output Parameters

*errorMessage* **String** Specifies the error message if an error occurs.

*error* **String** Specifies whether an error occurred. Valid values: `yes` and `no`.

## wm.fin.bic:insertSRList

This service imports a SEPA Routing list into the database.

### Input Parameters

| | |
|---|---|
| *filename* | **String** Fully qualified path and file name of the SR list you want to import, for example, c:\bic\sample\SR_20990101.txt |

### Output Parameters

| | |
|---|---|
| *errorMessage* | **String** Specifies the error message if an error occurs. |
| *error* | **String** Specifies whether an error occurred. Valid values: `yes` and `no`. |

## wm.fin.bic:validateBankID

This service validates the National Code for a financial institution, as follows:

1. Retrieves the country code (the first two characters) from the IBAN.
2. Retrieves the *Unique_ IBAN_National_Id* from the IBAN using the IS list.
3. In the BI (BICPlusIBAN) list, searches the row using the *IBAN_Country_Code* and the *Unique_IBAN_National_Id* as search criteria.
4. If the row exists, the national code is valid.

### Input Parameters

| | |
|---|---|
| *IBAN* | **String** IBAN of the financial institution. |

### Output Parameters

| | |
|---|---|
| *output* | **String** Specifies whether the validation succeeded. Valid values: `true` and `false`. |
| *errorMessage* | **String** Specifies the error message, if an error occurs. |
| *error* | **String** Specifies whether an error occurred. Valid values: `yes` and `no`. |

# wm.fin.bic:validateBICCode

This service validates the BIC code for a financial institution, as follows:

1. Splits the BIC into a *BIC_Code* (the first 8 characters) and a *branchCode* (characters 9 to 11). If the branch code is empty, it substitutes it with `XXX`.
2. In the BI (BICPlusIBAN) list, searches for the *BIC_Code* and *branchCode* in the data file.
3. If a record is found, the BIC is valid.

## Input Parameters

| | |
|---|---|
| *bicCode* | **String** BIC code of the financial institution. |

## Output Parameters

| | |
|---|---|
| *output* | **String** Specifies if the validation succeeded. Valid values: `true` and `false`. |
| *errorMessage* | **String** Specifies the error message if an error occurs. |
| *error* | **String** Specifies whether an error occurred. Valid values: `yes` and `no`. |

# wm.fin.bic:validateBICIBAN

This service validates the BIC code and IBAN combination for a financial institution, as follows:

1. Finds the *PARENT BANK CODE* from the IBAN:
   a. Retrieves the *Country_Code* from the IBAN (the first two characters).
   b. Retrieves the *Unique_IBAN_National_Id* from the IBAN using the IS list.
   c. Searches the BI (BICPlusIBAN) list using the *IBAN Country_Code* and the *Unique_IBAN_National_Id* as search criteria.
   d. Retrieves the *PARENT BANK CODE* from the row found it the file.
2. Finds the *PARENT BANK CODE* from the BI (BICPlusIBAN) list:
   a. Splits the BIC into a *bicCode* (the first 8 characters) and a *branchCode* (characters 9 to 11). If the *branchCode* is empty, substitutes it with `XXX`.
   b. In the BI list, finds the *bicCode* and the *branchCode*.
   c. Retrieves the *PARENT BANK CODE* from the row found in the file.
3. Compares the parent bank codes found in the first two steps. If the *PARENT BIC CODEs* are the same, then the BIC and the IBAN belong to the same institution.

**Note:** If no matching record is found, it does not necessarily mean that the IBAN/BIC combination is invalid. Determine if any of the following situations apply:

- The account servicing institution issues IBANs along with the BIC of another institution.
- The account servicing institution has multiple BIC codes with different bank codes (first 4 characters).

### Input Parameters

| | |
|---|---|
| *bicCode* | **String** BIC code of the financial institution. |
| *IBAN* | **String** IBAN of the financial institution. |

### Output Parameters

| | |
|---|---|
| *output* | **String** Specifies whether the validation succeeded. Valid values: `true` and `false`. |
| *errorMessage* | **String** Specifies the error message if an error occurs. |
| *error* | **String** Specifies whether an error occurred. Valid values: `yes` and `no`. |

## wm.fin.bic:BICInfo

Record structure identifying the BIC record retrieved from the database. The document specifies the result from executing the wm.fin.bic:getBICInfo service.

## wm.fin.dev Folder

This folder contains design-time services used for the install and configuration of new SWIFT FIN messages.

## wm.fin.dev:importFINItems

This service imports, configures, and creates all items needed in a SWIFT message transaction. This includes the IS document, DFD, parse template, TN document type, TN processing rule, and TN TPA.

### Input Parameters

| | |
|---|---|
| *msgType* | **String** SWIFT message type, for example, 502. |
| *version* | **String** FIN version (for example, nov10). |

*format*

**String** Optional. The format of the generated blocks and fields for the input SWIFT message. Valid values are:

- `TAG_BIZNAME` (default). SWIFT message tag followed by the business name specified in the message DFD, for example, `23G_Function of the Message`.
- `TAGONLY`. SWIFT message tag only. for example, `23G:`.
- `BIZNAMEONLY`. Business name specified in the message DFD, for example, `Function of the Message`.
- `XMLTAG`. XML-compatible tag name. This format cannot contain colons or tags that begin with a number, for example, `F23G`.

*subfieldFlag*

**String** Specifies whether to parse fields to the subfield level in the IS Document Type generated for this SWIFT FIN message. Valid values:

- `true` (default). Parse to the subfield level:
  - For inbound messages, removes the SWIFT delimiter (/) between subfields.
  - For outbound messages, adds the SWIFT delimiter (/) between subfields.
- `false`. Parse to the field level.

*createDocType*

**String** Optional. Indicates whether to create and insert a TN document type for this message. The TN document type is used to recognize an incoming message. Valid values: `true` or `false`.

*createProcessing*

*Rule*

**String** Optional. Indicates whether to create a Trading Networks processing rule for this message. After the message is recognized, the processing rule specifies how the message should be processed. Valid values: `true` or `false`.

*createTPA*

**String** Optional. Indicates whether to create a Trading Networks TPA for this message. This specifies specific variables used in WmFIN for processing and validation. Valid values: `true` or `false`.

### Output Parameters

None.

## wm.fin.dfd Folder

This folder contains services related to the loading and use of the FIN Data Field Dictionary (DFD).

## wm.fin.dfd:convertBizNameFormat

This service converts FIN IData from a specified format to `TAGONLY` format and merges subfields into a FIN field.

### Input Parameters

*finIData* **Document** FIN IData in the format specified in the *fromFormat* input string.

*msgType* **String** SWIFT message type, for example, `502`.

*version* **String** Version number of the SWIFT message record, for example, `nov10`.

*fromFormat* **String** The format of the generated blocks and fields for the input SWIFT message. Valid values:

- `TAG_BIZNAME` (default). SWIFT message tag followed by the business name specified in the message DFD, for example, `23G_Function of the Message`.
- `TAGONLY`. SWIFT message tag only, for example, `23G:`.
- `BIZNAMEONLY`. Business name specified in the message DFD, for example, `Function of the Message`.
- `XMLTAG`. XML-compatible tag name. This format cannot contain colons or tags that begin with a number, for example, `F23G`.

*subfieldFlag* **String** Optional. Specifies whether to parse the fields in the input *finIData* to the subfield level. Valid values:

- `true` (default). Parse to the subfield level:
  - For inbound messages, removes the SWIFT delimiter (/) between subfields.
  - For outbound messages, adds the SWIFT delimiter (/) between subfields.
- `false`. Parse to the field level.

### Output Parameters

*convertedFinIData* **Document** Converted FIN IData in the format of `TAGONLY`.

## wm.fin.dfd:convertTagFormat

This service converts FIN IData from TAGONLY to the specified format and parses whole fields into subfields.

### Input Parameters

*finIData* — **Document** FIN IData in the format specified in the *fromFormat* input string.

*msgType* — **String** SWIFT message type, for example, 502.

*version* — **String** Version number of the SWIFT message record being used, for example, nov10.

*toFormat* — **String** The format of the generated blocks and fields for the input SWIFT message. Valid values:

- TAG_BIZNAME (default). SWIFT message tag followed by the business name specified in the message DFD, for example, 23G_Function of the Message.
- TAGONLY. SWIFT message tag only. for example, 23G:.
- BIZNAMEONLY. Business name specified in the message DFD, for example, Function of the Message.
- XMLTAG. XML-compatible tag name. This format cannot contain colons or tags that begin with a number, for example, F23G.

*subfieldFlag* — **String** Optional. Specifies whether to parse the fields in the input *finIData* to the subfield level. Valid values:

- true (default). Parse to the subfield level.
  - For inbound messages, removes the SWIFT delimiter (/) between subfields.
  - For outbound messages, adds the SWIFT delimiter (/) between subfields.
- false. Parse to the field level.

*userParameters* — **Document** Optional. User parameters providing configuration information for the message.

### Output Parameters

*convertedFinIData* — **Document** Converted FIN IData in the format specified in the *fromFormat* input string.

# wm.fin.dfd:getDFDList

This service displays a list of DFDs loaded into the system.

### Input Parameters

None.

### Output Parameters

| | |
|---|---|
| *dfdList* | **StringList** Conditional. List of DFDs loaded into the system in <dfd name>_<dfd version> format, for example, `541_nov10`. |

# wm.fin.dfd:loadDFD

This service loads a FIN DFD into memory.

### Input Parameters

| | |
|---|---|
| *msgType* | **String** SWIFT message type, for example, `541`. |
| *version* | **String** Optional. Version number of the SWIFT message record being used, for example, `nov10`. |

### Output Parameters

None.

# wm.fin.dfd:unloadDFD

This service unloads a FIN DFD from memory.

### Input Parameters

| | |
|---|---|
| *msgType* | **String** SWIFT message type, for example, `541`. |
| *version* | **String** Optional. Version number of the SWIFT message record being used, for example, `nov10`. |

### Output Parameters

None.

## wm.fin.dfd:unloadDFDs

This service unloads all FIN DFDs from memory.

### Input Parameters

None.

### Output Parameters

None.

## wm.fin.doc Folder

This folder contains the document structures used to represent particular sections of SWIFT FIN messages, such as the header and trailer structures, and their fields. Also within this folder are the generic structure definitions for incoming and outgoing SWIFT FIN messages, where the data record structure (known as block 4 in SWIFT FIN messages) is left as an open record. The wm.fin.dev:importFINItems service (in the wm.fin.dev folder) generates these items. This folder also includes publishable IS document types that are used to send and receive SWIFT FIN messages, and IS document types that are used to populate values for a given TPA.

## wm.fin.doc:FINIData_Input

Record structure defining the fields of an incoming SWIFT message. B4 is left as an open record and can be created based on the particular message type and version.

## wm.fin.doc:FINIData_Output

Record structure defining the fields of an outgoing SWIFT message. B4 is left as open record and can be created based on the particular message type and version.

## wm.fin.doc:FINInboundMessage

Deprecated. Publishable document. SWIFT FIN messages received via AFT or MQ Series are mapped into this document. This document is then published to webMethods Broker or Integration Server where it is processed by the wm.fin.trp:FINInboundMessageTrigger and wm.fin.trp:receive services.

### Usage Notes

This document is deprecated.

## wm.fin.doc:FINOutboundMessage

Deprecated. Publishable document. SWIFT FIN messages sent via AFT or MQSeries are mapped into this document and published to webMethods Broker or Integration Server.

- If *Transport* = `MQ`, wm.fin.transport.MQSeries:MQSeriesPutTrigger subscribes to and processes this document. This trigger is deprecated.
- If *Transport* = `AFT`, wm.fin.transport.AFT:AFTOutboundTrigger subscribes to and processes this document. This trigger is deprecated.

### Usage Notes

This document is deprecated.

Note: Please see *webMethods SWIFT Module Samples Guide* for information about sending SWIFT FIN messages using the AFT or MQSeries.

## wm.fin.doc:MessageHeader

Non-publishable document. Data used to populate to header blocks (B1,B2,B3 and B5) in the outgoing SWIFT message.

## wm.fin.doc:UserParameters

Non- publishable document. TPA information to be used while sending and receiving SWIFT FIN messages.

### wm.fin.doc.catF Folder

The record definitions in this folder describe the record structures that represent the body of the FIN acknowledgement.

## wm.fin.doc.catF:MTF21

Record structure defining the fields of the body of the FIN Acknowledgement (F21).

## wm.fin.doc.header Folder

The record definitions within this folder describe the record structures used to represent the three header sections of a SWIFT message; the Basic Header (known as block 1 in SWIFT FIN messages), the Application Header (block 2, in both incoming and outgoing message format), and User Header (block 3).

### wm.fin.doc.header:ApplicationHeader_Input

Record structure defining the fields of the Application Header (block 2) on an incoming SWIFT message.

### wm.fin.doc.header:ApplicationHeader_Output

Record structure defining the fields of the Application Header (block 2) on an outgoing SWIFT message.

### wm.fin.doc.header:BasicHeader

Record structure defining the fields of the Basic Header (block 1) of a SWIFT message.

### wm.fin.doc.header:UserHeader

Record structure defining the fields of the User Header (block 3) of a SWIFT message.

## wm.fin.doc.trailer Folder

The record definitions within this folder describe the record structures used to represent the trailer section of a SWIFT message, known as block 5.

## wm.fin.doc.trailer:Trailer

Record structure defining the fields representing the trailer section (block 5) of a SWIFT message.

# wm.fin.format Folder

This folder contains format-related services. They are used in converting formats, such as a SWIFT message format, into a FIN IData.

## wm.fin.format:conformFINIData

This service conforms (rearranges) FIN IData into the correct structure based on the B4 IS document.

### Input Parameters

| | |
|---|---|
| *inputFINIData* | **Document** Input bound FIN IData (must include B4 block). |
| *isDocument* | **String** Fully-qualified IS document name to which the *inputFINIData* B4 block conforms, for example, `wm.fin.doc.nov10.cat5:MT502`. |

### Output Parameters

| | |
|---|---|
| *outputFINIData* | **Document** Output bound conformed FIN IData. |

## wm.fin.format:conformIData

This service conforms (rearranges) FIN IData into the correct structure based on the B4 IS document.

### Input Parameters

| | |
|---|---|
| *finIData* | **Document** Input bound FIN IData (must include B4 block). |
| *isDocument* | **String** Fully-qualified IS document name to which the *finIData* B4 block conforms, for example, `wm.fin.doc.nov10.cat5:MT502`. |

### Output Parameters

*finIData* Document Conditional. Output bound conformed IData.

## wm.fin.format:convertFINBlock4ToISDoc

This service maps the contents of a formatted block 4 MT message from the back-end MT IS document type.

### Input Parameters

*finMsgBlock4* String Block 4 of a FIN message according to the SWIFT specification.

*version* String Optional. Version number of the SWIFT message record being used, for example, `nov10`.

*msgType* String Optional. SWIFT message type identifier, for example, `199`.

*relaxed* String Optional. Formats the new line or line feed characters to SWIFT specific control characters "\r\n".

> Note: As per SWIFT specification the path separator for all the lines in block 4 is "\r\n". "\r"- is specified as the line feed character and "\n"- as the new line character.

### Output Parameters

*MTISDoc* Document Back-end MT IS document type.

## wm.fin.format:convertFINToIData

This service converts a SWIFT message into FIN IData. It loads a "parse" template into memory to create the correct structure.

### Input Parameters

*finMsg* String Valid SWIFT message.

*version* String Optional. Version number of the SWIFT message record being used, for example, `nov10`.

> Note: When using this service to convert a SWIFT acknowledgement (ACK) or negative acknowledgement (NACK) to an IData object, do not specify the version input variable because ACKs and NACKs are version neutral.

*msgType* **String** SWIFT message type identifier, for example, `541`.

*relaxed* **String** Optional. Valid values are `true` and `false`. Formats the new line or line feed characters to SWIFT specific control characters “\r\n”.

**Note:** As per SWIFT specification the path separator for all the lines in block 4 is “\r\n”. “\r”- is specified as the line feed character and “\n”- as the new line character.

### Output Parameters

*finIData* **Document** Conditional. FIN IData in the format specified.

## wm.fin.format:convertIDataToFIN

This service converts FIN IData into a SWIFT message.

### Input Parameters

*finIData* **Document** FIN IData in the format specified.

### Output Parameters

*FINmsg* **String** Conditional. Output SWIFT message.

## wm.fin.format:convertISMTDocToFINFormat

This service creates block 4 of an MT message in FIN format from the back-end MT IS document type.

### Input Parameters

*ISMTDoc* **Document** An instance of an MT FIN document, for example, `wm.fin.doc.nov10.cat1:MT199`.

### Output Parameters

*finFormattedBlock4* **String** Conditional. Flat file structure of block 4 according to the SWIFT specification.

## wm.fin.format:flushTemplateCache

This service clears “parse” templates from memory.

### Input Parameters

None.

### Output Parameters

None.

## wm.fin.format:xmlToIData

This service converts an XML-formatted SWIFT message into IData.

### Input Parameters

| | |
|---|---|
| *xmlString* | String XML string. |

### Output Parameters

| | |
|---|---|
| *outputIData* | Document Conditional. Output FIN IData. |

## wm.fin.init Folder

The services in this folder either initialize or de-initialize FIN packages on startup and shutdown of webMethods Integration Server.

## wm.fin.init:startup

This service initializes DSP user interface and resource bundles and configures the wmFIN package for run-time.

### Input Parameters

None.

### Output Parameters

None.

## wm.fin.init:shutdown

This service de-initializes the WmFIN package when Integration Server shuts down.

### Input Parameters

None.

### Output Parameters

None.

# wm.fin.map Folder

The services in this folder provide an easy framework for creating the header and trailer sections of SWIFT FIN messages for outbound messages (to be sent to SWIFT). The input for all services is the mandatory header or trailer section.

## wm.fin.map:mapApplicationBlockHeader

This service maps the input variables into a default FIN application header.

### Input Parameters

| | |
|---|---|
| *userParameters* | Document Reference FIN transport user variables. |

### Output Parameters

| | |
|---|---|
| *B2* | Document Reference Application header IData. |

### Usage Notes

This service replaces the deprecated servicewm.fin.map:mapApplicationHeader.

## wm.fin.map:mapApplicationHeader

Deprecated. Maps the input variables into a default FIN application header.

### Input Parameters

| | |
|---|---|
| *userParameters* | Document Reference FIN transport user variables. |
| *bizEnv* | Document Reference TN business envelope. This service derives the branch information and uses it to populate the receiver field in the address. |

### Output Parameters

| | |
|---|---|
| *B2* | Document Reference Application header IData. |

### Usage Notes

This service is deprecated and is replaced by wm.fin.map:mapApplicationBlockHeader.

## wm.fin.map:mapBasicBlockHeader

This service maps the input variables into a default FIN application header.

### Input Parameters

| | |
|---|---|
| *userParameters* | Document Reference FIN transport user variables. |

### Output Parameters

| | |
|---|---|
| *B1* | Document Reference Basic header IData. |

### Usage Notes

This service replaces the deprecated service wm.fin.map:mapBasicHeader.

## wm.fin.map:mapBasicHeader

Deprecated. This service maps the input variables into a default FIN application header.

### Input Parameters

| | |
|---|---|
| *userParameters* | Document Reference FIN transport user variables. |
| *bizEnv* | Document Reference TN business envelope. This service extracts the sender information and uses it to populate the LT information in the header. |

### Output Parameters

| | |
|---|---|
| *B1* | Document Reference Basic header IData. |

### Usage Notes

This service is deprecated and is replaced by wm.fin.map:mapBasicBlockHeader.

# wm.fin.map:mapOutbound

Deprecated. This service maps the input variables into *finIData*.

### Input Parameters

| | |
|---|---|
| *finText* | Document Block 4 content of the FIN message. |
| *userParameters* | Document Reference FIN transport user variables. |

### Output Parameters

| | |
|---|---|
| *finIData* | Document Reference FIN IData in the format specified. |

### Usage Notes

This service is deprecated and is replaced by wm.fin.map:mapOutboundMessage.

# wm.fin.map:mapOutboundMessage

This service maps the input variables into *finIData*.

### Input Parameters

| | |
|---|---|
| *finText* | Document Block 4 content of the FIN message. |
| *userParameters* | Document Reference FIN transport user variables. |

### Output Parameters

| | |
|---|---|
| *finIData* | Document Reference FIN IData in the format specified. |

### Usage Notes

This service replaces the deprecated service wm.fin.map:mapOutbound.

# wm.fin.map:mapTrailer

This service creates a trailer record. For outbound SWIFT messages, this record does not need to be populated, so this service currently creates an empty trailer record.

### Input Parameters

| | |
|---|---|
| *userParameters* | Document Reference FIN transport user variables. |

### Output Parameters

| | |
|---|---|
| *B5* | Document Reference Trailer IData. |

## wm.fin.map:mapUACK

This service maps the SWIFT acknowledgement to *finIData*.

### Input Parameters

| | |
|---|---|
| *accept-reject* | String Acknowledgement status of the message. |
| *rejection-reason* | String Optional. Reason for a negative acknowledgement. |
| *mur* | String Optional. Unique identifier of the FIN message for which this acknowledgement is received. |

### Output Parameters

| | |
|---|---|
| *finIData* | Document Reference FIN IData in the format specified. |

## wm.fin.map:mapUserBlockHeader

This service maps the input variables into a default FIN application header.

### Input Parameters

| | |
|---|---|
| *userParameters* | Document Reference FIN transport user variables. |

### Output Parameters

| | |
|---|---|
| *B3* | Document Reference User header IData. |

### Usage Notes

This service replaces the deprecated service wm.fin.map:mapUserHeader.

## wm.fin.map:mapUserHeader

Deprecated. Maps the input variables into a default FIN application header.

### Input Parameters

| | |
|---|---|
| *userParameters* | Document Reference FIN transport user variables. |
| *bizEnv* | Document Reference Optional. TN business envelope. This service extracts the conversation ID (if provided) and uses it to populate the MUR for the FIN message. |

### Output Parameters

| | |
|---|---|
| *B3* | Document Reference User header IData. |

### Usage Notes

This service is deprecated and is replaced by wm.fin.map:mapUserBlockHeader.

# wm.fin.marketPractice Folder

This folder contains common services that support Market Practices for some Category 5 messages. The services in this package are for internal use only.

# wm.fin.rules Folder

The services in this folder provide utility functions that are used in the implementation of network validation rules.

## wm.fin.rules:checkCodeOrder

This service specifies whether codes are in the correct order.

### Input Parameters

| | |
|---|---|
| *codeList* | String List The input code list. |
| *codeOrder* | String List The correct order of codes. |

### Output Parameters

| | |
|---|---|
| *isCodeOrderValid* | String Specifies whether the code list is valid. Valid values: `true` or `false`. |

## wm.fin.rules:contains

This service specifies whether a key is contained in the code list.

### Input Parameters

| | |
|---|---|
| *codeList* | String List The input code list. |
| *key* | String Key that may be in the code list. |

### Output Parameters

| | |
|---|---|
| *keyExists* | String Specifies whether the key exists in the code list. Valid values: `true` or `false`. |

## wm.fin.rules:getDuplicateCodeList

This service returns all codes that are duplicates in the code list.

### Input Parameters

| | |
|---|---|
| *codeList* | String List Optional. The input code list. |

### Output Parameters

| | |
|---|---|
| *duplicateCodeList* | String List Conditional. All duplicate codes in the input *codeList*. |

## wm.fin.rules:setErrorDocument

This service returns the appropriate error document from the specified variables.

### Input Parameters

| | |
|---|---|
| *key* | String Error message key. This is usually a FIN error message code. |
| *path* | String Path of the error in the message, for example, `B4/SBB/57D:`. |
| *data* | String Data where the error occurs. |

### Output Parameters

| | |
|---|---|
| *errors* | Document List Error array with the error appended to the end. |

## wm.fin.sepa Folder

This folder contains SEPA-related services used to derive or validate data against the SEPA Routing Directory.

## wm.fin.sepa:checkOperationalReadiness

This service validates a BIC's operational readiness to ensure that a BIC is ready to receive SEPA payment instructions for a particular scheme, as follows:

1. Splits the BIC into a BIC code (the first 8 characters) and a branch code (characters 9 to 11). If the branch code is empty, substitutes it with `XXX`.
2. Searches the data file with the BIC code, branch code, service level (for example, SEPA), and scheme instrument.
3. If no record is found for a specific branch code, it repeats the search with `XXX` in the branch code.

4 If at least one record is found with an operational readiness date older than the current date, and with an active validity period, the BIC is ready to accept payment instructions for the service level/scheme instrument and can receive payment instructions through the payment channel(s).

If no record is found, you must investigate manually to validate that the counterpart bank is ready to accept SEPA payment instructions.

### Input Parameters

| | |
|---|---|
| *bicCode* | String BIC code of the financial institution. |
| *serviceLevel* | String The SEPA service level. |
| *schemaIns* | String The scheme instruments within the SEPA service level for which data is collected and published. |

### Output Parameters

| | |
|---|---|
| isReady | String Specifies if the input BIC code is operationally ready for the input scheme. Valid values are `true` and `false`. |
| *message* | String Specifies the reason if *isReady* is false. |
| *errorMessage* | String Specifies the error message if an error occurs. |
| *error* | String Specifies whether an error occurred. Valid values: `yes` and `no`. |

## wm.fin.sepa:getAvailablePaymentChannels

This service identifies the available payment channels for a BIC code, as follows:

1 Splits the BIC into a BIC code (the first 8 characters) and a branch code (characters 9 to 11). If the branch code is empty, substitutes it with `XXX`.

2 Searches the data file with the BIC code, branch code, service level (for example, SEPA), and scheme instrument.

3 If no record is found for a specific branch code, then repeats the search with `XXX` in the branch code.

4 The search may return multiple rows if the counterpart bank is reachable through multiple payment channels.

- The *PaymentChannelID* field in each retrieved record provides the list of possible payment channels.
- The *ValidFrom* and *ValidTo* date fields may restrict the validity of the entries.

If no record is found, you must investigate manually to validate that the counterpart bank is ready to accept SEPA payment instructions, and determine the payment channels through which you can reach the bank.

**Note:** If no available payment channels are found, the service returns the message: "No Available Payment Channel found."

### Input Parameters

| | |
|---|---|
| *bicCode* | **String** BIC code of the financial institution. |
| *serviceLevel* | **String** The SEPA service level. |
| *schemaIns* | **String** The scheme instruments within the SEPA service level for which data is collected and published. |

### Output Parameters

| | |
|---|---|
| *message* | **String** Conditional. Indicates that there is no available payment channel. |
| *PaymentChannelInfo* | **Document List** The available payment channel. |
| *PaymentChannelID* | **String** The ID of the available payment channel. |
| *ValidFrom, ValidTo* | **String** Show the validity of the available payment channel. |
| *errorMessage* | **String** Specifies the error message, if an error occurs. |
| *error* | **String** Specifies whether an error occurred. Valid values: `yes` and `no`. |

## wm.fin.sepa:getOtherPaymentChannel

This service identifies the payment channels available to the financial institution with the input BIC code, using an intermediary institution, as follows:

1. Retrieves the BIC from the intermediary institution BIC field.
2. Splits the intermediary institution BIC into a BIC code (the first 8 characters) and a branch code (characters 9 to 11). If the branch code is empty, substitutes it with `XXX`.
3. Searches the data file with the BIC code, branch code, service level (for example, SEPA), and scheme instrument.
4. If no record is found for a specific branch code, repeats the search with `XXX` in the branch code.

   The search may return multiple rows if the intermediary institution is reachable through multiple payment channels.

   - The *PaymentChannelID* field in each retrieved record provides the list of possible payment channels through which the intermediary institution BIC is reachable.
   - The *ValidFrom* and *ValidTo* date fields restrict the validity of the entries.

### Input Parameters

*bicCode* String BIC code of the financial institution.

*serviceLevel* String The SEPA service level.

*schemaIns* String The scheme instruments within the SEPA service level for which data is collected and published.

### Output Parameters

*message* String Conditional. Indicates that there is no available payment channel.

*PaymentChannelInfo* Document List The available payment channel.

*PaymentChannelID* String The ID of the available payment channel.

*ValidFrom, ValidTo* String Indicates the validity of the available payment channels.

*errorMessage* String Specifies the error message, if an error occurs.

*error* String Specifies whether an error occurred. Valid values: `yes` and `no`.

## wm.fin.sepa:getPreferredPaymentChannel

This service determines whether the counterpart financial institution has specified a preferred payment channel for receiving payment instructions, as follows:

1 Splits the BIC into a BIC code (the first 8 characters) and a branch code (characters 9 to 11). If the branch code is empty, substitutes it with `XXX`.

2 Searches the data file with the BIC code, branch code, service level (for example, SEPA), scheme instrument, and preferred channel flag set to `P`.

3 If no record is found for a specific branch code, the service repeats the search with `XXX` in the branch code.

If a record is found, then the *PaymentChannelID* field in the retrieved record provides the preferred payment channel. The *ValidFrom* and *ValidTo* date fields restrict the validity of the entry.

If no record is found, then there is no preferred channel.

Note: If no available payment channels are found the output for the service contains the following message: “No Preferred Payment Channel found.”

### Input Parameters

*bicCode* String BIC code of the financial institution.

*serviceLevel* String The SEPA service level.

| | |
|---|---|
| *schemaIns* | String The scheme instruments within the SEPA service level for which data is collected and published. |

### Output Parameters

| | |
|---|---|
| *message* | String Conditional. Indicates that there is no preferred payment channel. |
| *PaymentChannelInfo* | Document List The preferred payment channel. |
| *PaymentChannelID* | String The ID of the preferred payment channel. |
| *ValidFrom, ValidTo* | String Show the validity of the preferred payment channel. |
| *errorMessage* | String Specifies the error message, if an error occurs. |
| *error* | String Indicates whether an error occurred. Valid values: `yes` and `no`. |

## wm.fin.sepa:validateAdherenceStatus

This service validates a BIC's adherence status to confirm that an institution has signed an adherence agreement for a particular scheme and is published in the EPC Register of Participants (that is, the adherence database).

1. Splits the BIC into a BIC code (the first 8 characters) and a branch code (characters 9 to 11). If the branch code is empty, substitutes it with `XXX`.
2. Searches the data file with the BIC code, branch code, service level (for example, SEPA), and scheme instrument.
3. If no record is found for a specific branch code, repeats the search with `XXX` in the branch code.

   If at least one record is found in which the Adherent Institution Flag has the value `P` (Published Institution), then the institution has been published in the EPC Register.

   If no record is found, you may consult the EPC Register of Participants directly. If the bank is listed as adherent to the scheme, then the bank may not have registered its routing information in the SEPA Routing Directory or may not have provided to SWIFT the institutions' reference BIC as provided on the scheme Adherence Agreement Schedule form.

### Input Parameters

| | |
|---|---|
| *bicCode* | String BIC code of the financial institution. |
| *serviceLevel* | String The SEPA service level. |
| *schemaIns* | String The scheme instruments within the SEPA service level for which data is collected and published. |

### Output Parameters

| | |
|---|---|
| isAdherent | **String** Specifies if the input BIC code adheres to the input scheme. Valid values are `true` and `false`. |
| *message* | **String** Specifies the reason if *isAdherent* is false. |
| *errorMessage* | **String** Specifies the error message if an error occurs. |
| *error* | **String** Specifies whether an error occurred. Valid values: `yes` and `no`. |

# wm.fin.transport Folder

This folder contains the services needed to exchange messages with SWIFT using Automated File Transfer (AFT) and MQSeries.

## wm.fin.transport.AFT Folder

The services in this folder send and receive messages using Automated File Transfer.

## wm.fin.transport.AFT:AFTOutboundTrigger

Deprecated. This trigger subscribes to outbound SWIFT FIN messages when the *transport* parameter in the message TPA is set to `AFT`. The trigger invokes the wm.fin.transport.AFT:processOutboundFile service to process the outbound SWIFT message.

## wm.fin.transport.AFT:generateUniqueFileName

This service generates a unique file name.

### Input Parameters

| | |
|---|---|
| *folder* | **String** The folder in which the file needs to be created. |
| *extension* | **String** Optional. The extension to use for the generated file name. If no extension has been specified, the default extension is `.inp`. |

### Output Parameters

| | |
|---|---|
| *fileName* | **String** Generated unique file name. |

## wm.fin.transport.AFT:processInboundFile

Deprecated. The flat file listener invokes this service to process incoming SWIFT FIN messages received via AFT. It then invokes wm.fin.trp:receiveto process each of the SWIFT FIN messages in the inbound batch file.

### Input Parameters

| | |
|---|---|
| *ffdata* | java.io.InputStream Input stream to the file received via AFT. |

### Output Parameters

None.

### Usage Notes

This service is deprecated and is replaced by wm.fin.transport.AFT:processIncomingFile.

## wm.fin.transport.AFT:processIncomingFile

This service is invoked to process the SWIFT FIN flat file message. This service breaks the batch input file into the individual SWIFT FIN messages. It then invokes wm.fin.trp:receiveMessageto process each of the SWIFT FIN messages in the inbound batch file.

### Input Parameters

| | |
|---|---|
| *ffdata* | java.io.InputStream Input stream to the file received via AFT. |

### Output Parameters

None.

### Usage Notes

This service replaces the deprecated service, wm.fin.transport.AFT:processInboundFile.

## wm.fin.transport.AFT:processIncomingMessage

This service parses incoming SWIFT FIN messages separated with special characters and outputs the SWIFT FIN messages as a string array with the special characters stripped.

### Input Parameters

| | |
|---|---|
| *ffdata* | java.io.InputStream Input stream to the file received via AFT. |

### Output Parameters

| | |
|---|---|
| *finMessage* | **String** A string list containing the individual SWIFT FIN messages with the special characters stripped. |

## wm.fin.transport.AFT:processOutboundFile

Deprecated. This service generates a unique file name and writes the outbound SWIFT message to the file in the folder specified in the TPA.

### Input Parameters

| | |
|---|---|
| *ffdata* | **java.io.InputStream** Input stream to the file received via AFT. |
| *wm.fin.doc:FINOutbound Message* | **Document Reference** The document to which this service subscribes when the *transport* parameter in the message TPA is set to `AFT`. |

### Output Parameters

None.

### Usage Notes

This service is deprecated and is replaced by wm.fin.transport.AFT:processOutgoingFile.

## wm.fin.transport.AFT:processOutgoingFile

This service generates a unique file name and writes the outbound SWIFT message to the file in the folder specified in the TPA.

### Input Parameters

| | |
|---|---|
| *finMsg* | **String** The SWIFT FIN message in flat file format. |
| *ProcessUser Parameters* | **Document Reference** Configuration document providing the required information to process the outbound file. |

### Output Parameters

None.

### Usage Notes

This service replaces the deprecated service, wm.fin.transport.AFT:processOutboundFile.

## wm.fin.transport.MQSeries

This folder contains services to send and receive SWIFT FIN messages from MQ Series.

# wm.fin.transport.MQSeries:getListenerService

Deprecated. This service retrieves SWIFT FIN messages from a specified MQSeries queue. This service strips out extraneous information in the SWIFT message and publishes the actual SWIFT message, to be processed further by wm.fin.trp:receive. The wm.fin.trp:receive service subscribes to, processes, and validates the message, after which the service either passes the resulting TN document type to the Process Engine or to the specified Trading Networks processing rule. The user must specify this service as the Message Service when creating the WebSphere MQ-to-IS message handler.

## Input Parameters

| | |
|---|---|
| *msgbody* | String SWIFT message retrieved off the specified WebSphere MQ queue. |

## Output Parameters

None.

## Usage Notes

This service is deprecated and is replaced by wm.fin.transport.MQSeries:getMQSeriesListenerService.

# wm.fin.transport.MQSeries:getMQSeriesListenerService

This service retrieves SWIFT FIN messages from a specified MQSeries queue, removes extraneous information in the SWIFT message, and publishes the actual SWIFT message for further processing by the wm.fin.trp:receiveMessage service.

## Input Parameters

| | |
|---|---|
| *msgbody* | String SWIFT message retrieved from the specified WebSphere MQ queue. |

## Output Parameters

None.

## Usage Notes

This service replaces the deprecated service, wm.fin.transport.MQSeries:getListenerService.

## wm.fin.transport.MQSeries:MQSeriesPutTrigger

Deprecated. This trigger subscribes to outbound SWIFT FIN messages when the *transport* parameter in the message TPA is set to `MQ`. The trigger then invokes the service, wm.fin.transport.MQSeries:put, to put the outbound SWIFT message into the specified WebSphere MQ queue.

### Input Parameters

None.

### Output Parameters

None.

### Usage Notes

This service is deprecated.

## wm.fin.transport.MQSeries:put

Deprecated. This service puts the outbound SWIFT message in a MQ Series queue by invoking the "put" message handler service created by the user and specified in the message TPA.

### Input Parameters

| | |
|---|---|
| *FINOutboundMessage* | **Document Reference** The document to which this service subscribes when the *transport* parameter in the message TPA is set to `MQ`. |

### Output Parameters

None.

### Usage Notes

This service is deprecated and is replaced by wm.fin.transport.MQSeries:putMessage.

## wm.fin.transport.MQSeries:putMessage

This service invokes the "put" message handler service (the user-created service specified in the corresponding message TPA), and puts the outbound SWIFT message in a MQ Series queue.

### Input Parameters

| | |
|---|---|
| *finMsg* | **String** SWIFT FIN message in the flat file format. |

| | |
|---|---|
| *ProcessUser Parameters* | Document Reference Configuration document providing the required information to process the outbound file. |

### Output Parameters

None.

### Usage Notes

This service replaces the deprecated service, wm.fin.transport.MQSeries:put.

## wm.fin.transport.property

This folder contains services to retrieve properties defined for publishing SWIFT FIN messages.

## wm.fin.transport.property:getProperty

This service returns the property value specified in *Integration Server_directory*\packages\WmFIN\config\finTransport.cnf file.

### Input Parameters

| | |
|---|---|
| *propertyName* | String Property name specified in the finTransport.cnf file. |

### Output Parameters

| | |
|---|---|
| *value* | String Value of the property name specified in the finTransport.cnf file. |

## wm.fin.transport.property:listProperties

This service returns all the properties specified in *IntegrationServer_directory*\packages\WmFIN\config\finTransport.cnf file.

### Input Parameters

None.

### Output Parameters

| | |
|---|---|
| *properties* | Document List of all the properties and their values specified in the finTransport.cnf file. |

# wm.fin.transport.Test

This folder contains services, triggers and publishable documents to be used with the SWIFT Module samples. For information about the SWIFT Module samples, see *webMethods SWIFT Module Samples Guide*.

# wm.fin.transport.Test:FINSampleInboundMessage

Deprecated. A publishable IS document that represents an inbound SWIFT message, used with the SWIFT Module samples.

### Input Parameters

None.

### Output Parameters

None.

### Usage Notes

This service is deprecated.

# wm.fin.transport.Test:FINSampleInboundMessageTrigger

Deprecated. This service subscribes to the FINSampleInboundMessage document that the wm.fin.transport.Test:processFinMsg service publishes. This trigger invokes the sample service, wm.fin.sample:receive, to process the incoming SWIFT message.

### Input Parameters

*wm.fin.doc:FINSampleInboundMessage* **Document** The document to which this service subscribes when the *transport* parameter in the message TPA is set to `Test`.

### Output Parameters

*wm.fin.doc:FINSample OutboundMessage* **Document** The document to which this service subscribes when the *transport* parameter in the message TPA is set to `Test`.

### Usage Notes

This service is deprecated.

## wm.fin.transport.Test:FINSampleOutboundMessageTrigger

Deprecated. This service subscribes to outbound SWIFT FIN messages when the *transport* parameter in the message TPA is set to `Test`. This trigger invokes wm.fin.transport.Test:processFinMsg to process outbound SWIFT messages.

### Input Parameters

| | |
|---|---|
| *wm.fin.doc:FINSampleInboundMessage* | Document The document to which this service subscribes when the *transport* parameter in the message TPA is set to `Test`. |

### Output Parameters

| | |
|---|---|
| *wm.fin.doc:FINSampleOutboundMessage* | Document The document to which this service subscribes when the *transport* parameter in the message TPA is set to `Test`. |

### Usage Notes

This service is deprecated.

## wm.fin.transport.Test:processFinMsg

Deprecated. This service receives an outbound SWIFT message and simulates a round-trip by publishing the same message as an inbound SWIFT message. wm.fin.transport.Test:FINSampleOutboundMessageTrigger invokes this service when the *transport* parameter in the message TPA is set to `Test`.

### Input Parameters

| | |
|---|---|
| *wm.fin.doc:FIN OutboundMessage* | Document The document to which this service subscribes when the *transport* parameter in the message TPA is set to `Test`. |

### Output Parameters

None.

### Usage Notes

This service is deprecated.

## wm.fin.trp Folder

This folder contains two core services that are used in conjunction with Trading Networks to provide single-point access to send and receive SWIFT FIN messages.

## wm.fin.trp:FINInboundMessageTrigger

Deprecated. This trigger subscribes to the wm.fin.doc:FINInboundMessage service. When a document is received, this trigger invokes the wm.fin.trp:receive service.

### Usage Notes

This service is depricated.

## wm.fin.trp:receive

Deprecated. The wm.fin.trp:FINInboundMessageTrigger service triggers this service. This service receives an incoming FINInboundMessage IData, parses it into a record, and sends it to Trading Networks for further processing.

### Input Parameters

| | |
|---|---|
| *wm.fin.doc:FINInboundMessage* | Document IData containing the raw SWIFT FIN message to be processed. |

### Output Parameters

None.

### Usage Notes

This service is deprecated and is replaced by wm.fin.trp:receiveMessage.

## wm.fin.trp:receiveMessage

This service receives an incoming FINInboundMessage IData, parses it into a record, and sends it to Trading Networks for further processing. The wm.fin.trp:FINInboundMessageTrigger service triggers this service to run.

### Input Parameters

| | |
|---|---|
| *rawFINMessage* | String Flat file inbound FIN message. |

### Output Parameters

None.

### Usage Notes

This service replaces the deprecated service, wm.fin.trp:receive.

## wm.fin.trp:send

Deprecated. This service formats IData into a SWIFT FIN message, persists the message in Trading Networks, and validates it. Once validated, the service sends the message to SAA, using the transport protocol configured in the TPA for the corresponding message type.

### Input Parameters

| | |
|---|---|
| *finBlock4Doc* | Document Block 4 of flat file FIN message in IS document format, for example, wm.fin.doc.nov11.cat5:MT564. |
| *SenderID* | String Sender ID used to retrieve TPA data from Trading Networks. Default is unknown. |
| *ReceiverID* | String Receiver ID used to retrieve the TPA data from Trading Networks. Default is unknown. |
| *msgType* | String Type of the FIN message, for example `564`. |

### Output Parameters

None.

### Usage Notes

This service is deprecated and is replaced by wm.fin.trp:sendMessage.

## wm.fin.trp:sendMessage

This service formats IData into a SWIFT FIN message, persists the message in Trading Networks, and validates it. Once validated, the service sends the message to SAA, using the transport protocol configured in the TPA for the corresponding message type.

### Input Parameters

| | |
|---|---|
| *finBlock4Doc* | Document Block 4 of flat file FIN message in IS document format, for example, `wm.fin.doc.nov11.cat5:MT564`. |
| *SenderID* | String Sender ID used to retrieve TPA data from Trading Networks. Default is unknown. |
| *ReceiverID* | String Receiver ID used to retrieve the TPA data from Trading Networks. Default is unknown. |
| *msgType* | String Type of the FIN message, for example `564`. |

### Output Parameters

None.

### Usage Notes

This service replaces the deprecated service, wm.fin.trp:sendMessage.

## wm.fin.utils Folder

The services found within this folder are generic utility services providing various functionality.

# wm.fin.utils:generateUniqueIdentifier

This service generates a unique identifier. Use the sample services wm.xmlv2.MT.maps:mapDataPDU and wm.xmlv2.MX.maps:mapDataPDU to populate XMLv2 headers with a unique identifier. SWIFT Module uses this identifier to reconcile notifications from SAA.

### Input Parameters

None.

### Output Parameters

| | |
|---|---|
| *UUID* | String Contains the unique identifier that the service generated. |

# wm.fin.utils:getFINMessageAndIDs

From a raw SWIFT message, this service recognizes and extracts the sender, receiver and message type.

### Input Parameters

| | |
|---|---|
| *rawFINMessage* | String Raw SWIFT message. |

### Output Parameters

| | |
|---|---|
| *finMessage* | Document Contains SWIFT Message and indicates whether message is an acknowledgement. |
| *internalSenderID* | String Trading Networks Internal Sender ID. |
| *internalReceiverID* | String Trading Networks Internal Receiver ID. |
| *msgType* | String SWIFT message type. |

## wm.fin.validation Folder

This folder contains validation-related services. They are used to facilitate the validation of a SWIFT message.

## wm.fin.validation:getErrorMessage

This service returns an appropriate SWIFT message for a key.

### Input Parameters

*key* **String** Error key.

### Output Parameters

*errorMessage* **String** FIN error message.

## wm.fin.validation:validateFinMsg

Parses and validates a SWIFT message.

### Input Parameters

*bizdoc* **Document Reference** Trading Networks BizDocEnvelope containing the SWIFT message.

### Output Parameters

*finIData* **Document** SWIFT message as an IData in `TAGONLY` format.

*convertedFinIData* **Document** SWIFT message as an IData in specified format.

## wm.fin.validation:validateIData

This service provides content validation, network rule validation, market practice rule validation, and usage rule validation of a FIN IData.

### Input Parameters

*finIData* **Document** FIN IData.

*userParameters* **Document Reference** User variables providing configuration information for the message.

### Output Parameters

*isValid* — **String** Specifies whether FIN IData passes validation according to the message configuration.

*errorArray* — **Document List** Errors occurring if FIN IData does not pass validation.

## wm.fin.validation:validateIDataUtil

This service validates content and structure of a FIN IData.

### Input Parameters

*validateHeaders* — **String** Optional. Specifies whether the service should validate the headers. Valid values: `true` or `false`.

### Output Parameters

*isValid* — **String** Specifies whether FIN IData passes validation according to the message configuration.

*errors* — **Document** Conditional. Errors occurring if FIN IData does not pass validation.

## wm.sdk.fin Folder

This folder contains the Java services, XSDs and IS document types to support the SWIFT SDK feature provided within SWIFT Module. SDK services convert MT messages from flat file format into XML or from XML format into flat file format. This folder also contains services to generate current versions of IS document types for MT and MX messages.

| Service | Service Description |
|---|---|
| wm.sdk.fin.converter:convertMTBlock4ToMTXML | This service converts block 4 of the flat file MT message into XML format. |
| wm.sdk.fin.converter:convertMTFlatFileToMTXML | This service converts the entire flat file MT message into XML format. |
| wm.sdk.fin.converter:convertMTXMLblock4ToMTFlatFile | This service converts block 4 of the MT XML message into flat file format. |
| wm.sdk.fin.converter:convertMTXMLToMTFlatFile | This service converts the entire MT XML into MT flat file format. |
| wm.sdk.fin.validator:validateMTXML | This service validates any MT XML message against the SWIFT SDK MT schema. |

## wm.sdk.rec.mtxsd.Vyear

This folder contains the IS document types with names matching the message type, MTxxx where “xxx” represents the message type. IS document types are organized in directories according to the supported version number. For example, the folder wm.sdk.rec.mtxsd.V2009:MTxxx contains the IS document types for version 2009 and wm.sdk.rec.mtxsd.V2010:MTxxx contains the IS document types for version 2010.

## wm.sdk.docgenerator:createMTISDocFromSchema

This service creates IS document types and IS schema for the corresponding MT schemas.

### Input Parameters

*msgType* String Required. The SWIFT FIN MT message type for which the IS document type will be created (for example, 564).

*sdkversion* String Required. The version of SWIFT SDK. This service uses the SDK version to determine which XSD to apply during file conversion. Valid values are any SDK version that SWIFT Module supports (for example, version 2009 or 2010).

### Output Parameters

*isSuccessful* String Required. Indicates if the service executed successfully.

*Warnings* Document Optional. Warnings that result during the creation of the IS document type.

*Errors* Document Optional. Errors that result during the creation of the IS document type.

## wm.sdk.docgenerator:createMXISDocFromSchema

This service creates IS document types and IS schema for the corresponding MX schemas.

### Input Parameters

*msgType* String Required. The SWIFT FIN MX message type for which the IS document type will be created (for example, acmt.018.001.01).

*namespaceURI* String Required. Target namespace of the MX schema (for example, urn:iso:std:iso:20022:tech:xsd:acmt.018.001.01).

### Output Parameters

*isSuccessful* String Required. Indicates if the service executed successfully.

*Warnings* **Document** Optional. Warnings that result during the creation of the message type.

*Errors* **Document** Optional. Errors that result during the creation of the message type.

## wm.sdk.fin.converter:convertMTBlock4ToMTXML

This service converts block 4 of the MT flat file into XML format.

### Input Parameters

*sdkversion* **String** Required. The version of SWIFT SDK. This service uses the SDK version to determine which XSD to apply during file conversion. Valid values are any SDK version that SWIFT Module supports (for example, version 2009 or 2010).

*msgType* **String** Required. The message type corresponding to block 4 of the MT message passed in from the *finMsg* parameter (for example, 199).

*finMsg* **String** Required. Block 4 of the MT flat file. This service converts block 4 data into MT XML format.

*relaxed* **String** Optional. Formats the input with carriage return and line feed characters (“\r\n”) to comply with SWIFT message specifications.

**Note:** The path separator for all the lines in block 4 is “\r\n”. The carriage return (\r) is omitted from the input file when viewed in the message editor. When this parameter is set to `true`, the service inserts the omitted carriage return character.

Valid values are `true` and `false`:

- `true` (Default) The service applies the correct “\r\n” formatting to the input file. This parameter must be `true` (selected) when providing the *finMsg* through the user interface dialog box.
- `false` This parameter is `false` (unselected) when the service processes the *finMsg* input from a file or from the previous step in the pipeline.

### Output Parameters

*xmlData* **String** The MT XML data generated from block 4 of the original MT flat file message.

*errDoc*

**Document Reference** Errors encountered during processing are reported using the standard “wm.sdk.rec.mtxsd:ErrorReport” document type format. The following tags are included within the XML error document:

| XML Tag | Description |
|---|---|
| *errDoc/ConversionErrors* | **Document** The opening XML tag for the errors generated during conversion. |
| *errDoc/ConversionErrors/Error* | **Document List** List of conversion errors generated during processing. |
| *errDoc/ConversionErrors/Error[x]/Code* | **String** Code corresponding to the error in SWIFT format (for example, TC00103). For more information see “SDK Error Descriptions” on page 250. |
| *errDoc/ConversionErrors/Error[x]/Message* | **String** Message corresponding to the error code in SWIFT format (for example, “The message content is invalid Details: MT2XML.002.004”). |
| *errDoc/ConversionErrors/rror[x]/Location* | **Document** Location within the *finMsg* where the conversion failed. |
| *errDoc/ConversionErrors/Error[0]/Location/LineNumber* | **String** Line number within the *finMsg* where the conversion failed. |

## wm.sdk.fin.converter:convertMTFlatFileToMTXML

This service converts the entire flat file MT message into an MT XML message.

### Input Parameters

*sdkversion*

**String** Required. The version of SWIFT SDK. This service uses the SDK version to determine which MT IS document schema to apply during file conversion. Valid values are any SDK version that SWIFT Module supports (for example, version 2009 or 2010).

*finMsg*

**String** Required. The FIN flat file MT message. This service converts the flat file into MT XML format.

*relaxed*

String Optional. Formats the input with carriage return and line feed characters (“\r\n”) to comply with SWIFT message specifications.

**Note:** The path separator for all the lines in block 4 is “\r\n”. The carriage return (\r) is omitted from the input file when viewed in the message editor. When this parameter is set to `true`, the service inserts the omitted carriage return character.

Valid values are `true` and `false`:

- `true` (Default) The service applies the correct “\r\n” formatting to the input file. This parameter must be `true` (selected) when providing the *finMsg* through the user interface dialog box.
- `false` This parameter is `false` (unselected) when the service processes the *finMsg* input from a file or from the previous step in the pipeline.

## Output Parameters

*xmlData*

String The XML message generated from the original *finMsg* flat file.

*errDoc*

Document Reference Errors encountered during processing are reported using the standard “wm.sdk.rec.mtxsd:ErrorReport” document type format. The following tags are included within the XML error document:

| XML Tag | Description |
|---|---|
| *errDoc/ConversionErrors* | Document The opening XML tag for the errors generated during conversion. |
| *errDoc/ConversionErrors/Error* | Document List List of conversion errors generated during processing. |
| *errDoc/ConversionErrors/Error[x]/Code* | String Code corresponding to the error in SWIFT format (for example, TC00103). For more information see “SDK Error Descriptions” on page 250. |
| *errDoc/ConversionErrors/Error[x]/Message* | String Message corresponding to the error code in SWIFT format (for example, “The message content is invalid Details: MT2XML.002.004”). |
| *errDoc/ConversionErrors/Error[x]/Location* | Document Location within the *finMsg* where the conversion failed. |

| | |
|---|---|
| *errDoc/ConversionErrors/Error[0]/Location/LineNumber* | String Line number within the *finMsg* where the conversion failed. |

## wm.sdk.fin.converter:convertMTXMLblock4ToMTFlatFile

This service converts block 4 of the MT XML message into flat file format.

### Input Parameters

| | |
|---|---|
| *sdkversion* | String Required. The version of SWIFT SDK. This service uses the SDK version to determine which XSD to apply during file conversion. Valid values are any SDK version that SWIFT Module supports (for example, version 2009 or 2010). |
| *msgType* | String Required. The message type corresponding to the MT XML message passed in the *finXML* parameter (for example, 199). |
| *finXML* | String Required. The MT XML message. This service converts the XML message into flat file format. |

### Output Parameters

| | |
|---|---|
| *finMsg* | String The flat file generated from block 4 of the MT XML message. |
| *errDoc* | Document Reference Errors encountered during processing are reported using the “wm.sdk.rec.mtxsd:ErrorReport” document type format. The following tags are provided in the XML error document: |

| XML Tag | Description |
|---|---|
| *errDoc/ConversionErrors* | Document The opening XML tag for the errors generated during conversion. |
| *errDoc/ConversionErrors/Error* | Document List List of conversion errors generated during processing. |
| *errDoc/ConversionErrors/Error[x]/Code* | String Code corresponding to the error in SWIFT format (for example, TC00103). For more information see “SDK Error Descriptions” on page 250. |
| *errDoc/ConversionErrors/Error[x]/Message* | String Message corresponding to the error code in SWIFT format (for example, “The message content is invalid Details: MT2XML.002.004”). |

| | |
|---|---|
| *errDoc/ConversionErrors/Error[x]/Location* | Document Location within the *finMsg* where the conversion failed. |
| *errDoc/ConversionErrors/Error[0]/Location/LineNumber* | String Line number within the *finMsg* where the conversion failed. |

# wm.sdk.fin.converter:convertMTXMLToMTFlatFile

This service converts the entire MT XML message into an MT flat file message.

## Input Parameters

*sdkversion* String Required. The version of SWIFT SDK. This service uses the SDK version to determine which XSD to apply during file conversion. Valid values are any SDK version that SWIFT Module supports (for example, version 2009 or 2010).

*msgType* String Required. The message type corresponding to the MT XML message that is passed in the *finXML* parameter (for example, 199).

*finXML* String Required. The MT XML message. This service converts the MT message into flat file format.

## Output Parameters

*finMsg* String The MT flat file message generated from the XML message.

*errDoc* Document Reference Errors encountered during processing are reported using the "wm.sdk.rec.mtxsd:ErrorReport" document type format. The following tags are provided in the XML error document:

| XML Tag | Description |
|---|---|
| *errDoc/ConversionErrors* | Document The opening XML tag for the errors generated during conversion. |
| *errDoc/ConversionErrors/Error* | Document List List of conversion errors generated during processing. |
| *errDoc/ConversionErrors/Error[x]/Code* | String Code corresponding to the error in SWIFT format (for example, TC00103). For more information see "SDK Error Descriptions" on page 250. |

| | |
|---|---|
| *errDoc/ConversionErrors/Error[x]/Message* | **String** Message corresponding to the error code in SWIFT format (for example, “The message content is invalid Details: MT2XML.002.004”). |
| *errDoc/ConversionErrors/Error[x]/Location* | **Document** Location within the *finMsg* where the conversion failed. |
| *errDoc/ConversionErrors/Error[0]/Location/LineNumber* | **String** Line number within the *finMsg* where the conversion failed. |

## wm.sdk.fin.validator:validateMTXML

This service validates any MT XML message against the SWIFT SDK MT schema.

### Input Parameters

*finXML* **String** Required. The MT XML message to be validated.

*sdkversion* **String** Required. The version of SWIFT SDK. This service uses the SDK version to determine which XSD to apply during file conversion. Valid values are any SDK version that SWIFT Module supports (for example, version 2009 or 2010).

*msgType* **String** Required. The message type corresponding to block 4 of the MT message passed in the finMsg parameter (for example, 199).

### Output Parameters

*isValid* **String** The result of the validation. The result s either `true` or `false`.

*errDoc* **Document Reference** Errors encountered during processing are reported using the “wm.sdk.rec.mtxsd:ErrorReport” document type format. The following tags are provided in the XML error document:

| XML Tag | Description |
|---|---|
| *errDoc/ConversionErrors* | **Document** The opening XML tag for the errors generated during conversion. |
| *errDoc/ConversionErrors/Error* | **Document List** List of conversion errors generated during processing. |

*errDoc/ConversionErrors/Error[x]/Code*

String Code corresponding to the error in SWIFT format (for example, TC00103). For more information see “SDK Error Descriptions” on page 250.

*errDoc/ConversionErrors/Error[x]/Message*

String Message corresponding to the error code in SWIFT format (for example, “The message content is invalid Details: MT2XML.002.004”).

*errDoc/ConversionErrors/Error[x]/Location*

Document Location within the *finMsg* where the conversion failed.

*errDoc/ConversionErrors/Error[0]/Location/LineNumber*

String Line number within the *finMsg* where the conversion failed.

## Supported SDK MX Message Types

The following table lists the SWIFT SDK MX schemas bundled with SWIFT Module.

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| acmt | 001.xxx.xx-005.xxx.xx | acmt.001.001.01<br>acmt.001.001.02<br>acmt.002.001.01<br>acmt.002.001.02<br>acmt.003.001.01<br>acmt.003.001.02<br>acmt.004.001.01<br>acmt.004.001.02<br>acmt.005.001.01<br>acmt.005.001.02 |
| acmt | 006.xxx.xx-010.xxx.xx | acmt.006.001.01<br>acmt.006.001.02<br>acmt.007.001.01<br>acmt.008.001.01<br>acmt.009.001.01<br>acmt.010.001.01 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| acmt | 011.xxx.xx-015.xxx.xx | acmt.011.001.01<br>acmt.012.001.01<br>acmt.013.001.01<br>acmt.014.001.01<br>acmt.015.001.01 |
| acmt | 016.xxx.xx-021.xxx.xx | acmt.016.001.01<br>acmt.017.001.01<br>acmt.018.001.01<br>acmt.019.001.01<br>acmt.020.001.01<br>acmt.021.001.01 |
| admi | 001.xxx.xx-.998.xxx.xx | admi.001.001.01<br>admi.002.001.01<br>admi.003.001.01<br>admi.004.001.01<br>admi.998.001.01 |
| camt | .001.xxx.xx-.005.xxx.xx | camt.003.001.03<br>camt.003.001.04<br>camt.004.001.03<br>camt.004.001.04<br>camt.005.001.03<br>camt.005.001.04 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| camt | .006.xxx.xx-<br>.010.xxx.xx | camt.006.001.03<br>camt.006.001.04<br>camt.007.001.03<br>camt.007.001.04<br>camt.007.002.02<br>camt.007.002.03<br>camt.008.001.03<br>camt.008.001.04<br>camt.008.002.02<br>camt.009.001.03<br>camt.009.001.04<br>camt.010.001.03<br>camt.010.001.04 |
| camt | .011.xxx.xx-<br>.015.xxx.xx | camt.011.001.03<br>camt.011.001.04<br>camt.012.001.03<br>camt.012.001.04<br>camt.013.001.02<br>camt.013.001.01<br>camt.014.001.01<br>camt.014.001.02<br>camt.015.001.01<br>camt.015.001.02 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| camt | .016.xxx.xx-<br>.020.xxx.xx | camt.016.001.01<br>camt.016.001.02<br>camt.017.001.01<br>camt.017.001.02<br>camt.018.001.01<br>camt.018.001.02<br>camt.019.001.02<br>camt.019.001.03<br>camt.020.001.01<br>camt.020.001.02 |
| camt | .0216.xxx.xx-<br>.025.xxx.xx | camt.021.001.01<br>camt.021.001.02<br>camt.023.001.02<br>camt.023.001.03<br>camt.024.001.02<br>camt.024.001.03<br>camt.025.001.01<br>camt.025.001.02 |
| camt | .026.xxx.xx-<br>.030.xxx.xx | camt.026.001.02<br>camt.026.001.03<br>camt.027.001.02<br>camt.027.001.03<br>camt.028.001.02<br>camt.028.001.03<br>camt.029.001.02<br>camt.029.001.03<br>camt.030.001.02<br>camt.030.001.03 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| camt | .031.xxx.xx-<br>.035.xxx.xx | camt.031.001.02<br>camt.031.001.03<br>camt.032.001.01<br>camt.032.001.02<br>camt.033.001.02<br>camt.033.001.03<br>camt.034.001.02<br>camt.034.001.03<br>camt.035.001.01<br>camt.035.001.02 |
| camt | .036.xxx.xx-<br>.040.xxx.xx | camt.036.001.01<br>camt.036.001.02<br>camt.037.001.02<br>camt.037.001.03<br>camt.038.001.01<br>camt.038.001.02<br>camt.039.001.02<br>camt.039.001.03<br>camt.040.001.02<br>camt.040.001.03 |
| camt | .041.xxx.xx-<br>.045.xxx.xx | camt.041.001.02<br>camt.041.001.03<br>camt.042.001.02<br>camt.042.001.03<br>camt.043.001.02<br>camt.043.001.03<br>camt.044.001.01<br>camt.044.001.02<br>camt.045.001.01<br>camt.045.001.02 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| camt | .046.xxx.xx-<br>.050.xxx.xx | camt.046.001.01<br>camt.046.001.02<br>camt.047.001.01<br>camt.047.001.02<br>camt.048.001.01<br>camt.048.001.02<br>camt.049.001.01<br>camt.049.001.02<br>camt.050.001.01<br>camt.050.001.02 |
| camt | .051.xxx.xx-<br>.055.xxx.xx | camt.051.001.01<br>camt.051.001.02<br>camt.052.001.01<br>camt.052.001.02<br>camt.053.001.01<br>camt.053.001.02<br>camt.054.001.01<br>camt.054.001.02<br>camt.055.001.01 |
| camt | .056.xxx.xx-<br>.999.xxx.xx | camt.056.001.01<br>camt.057.001.01<br>camt.058.001.01<br>camt.059.001.01<br>camt.060.001.01<br>camt.061.001.01<br>camt.062.001.01<br>camt.998.001.01<br>camt.998.001.02 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| colr | .001.xxx.xx-<br>.999.xxx.xx | colr.003.001.01<br>colr.004.001.01<br>colr.005.001.01<br>colr.006.001.01<br>colr.007.001.01<br>colr.008.001.01<br>colr.009.001.01<br>colr.010.001.01<br>colr.011.001.01<br>colr.012.001.01<br>colr.013.001.01<br>colr.014.001.01<br>colr.015.001.01 |
| fxtr | .001.xxx.xx-<br>.999.xxx.xx | fxtr.014.001.01<br>fxtr.015.001.01<br>fxtr.016.001.01 |
| head | .001.xxx.xx-<br>.999.xxx.xx | head.001.001.01 |
| pacs | .001.xxx.xx-<br>.999.xxx.xx | pacs.002.001.03<br>pacs.003.001.02<br>pacs.004.001.02<br>pacs.007.001.02<br>pacs.008.001.02<br>pacs.009.001.02 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| pain | .001.xxx.xx-<br>.999.xxx.xx | pain.001.001.02 |
| | | pain.001.001.03 |
| | | pain.002.001.02 |
| | | pain.002.001.03 |
| | | pain.007.001.01 |
| | | pain.007.001.02 |
| | | pain.008.001.01 |
| | | pain.008.001.02 |
| | | pain.009.001.01 |
| | | pain.010.001.01 |
| | | pain.011.001.01 |
| | | pain.012.001.01 |
| | | pain.998.001.01 |
| reda | .001.xxx.xx-<br>.999.xxx.xx | reda.001.001.02 |
| | | reda.001.001.03 |
| | | reda.002.001.02 |
| | | reda.002.001.03 |
| | | reda.003.001.02 |
| | | reda.003.001.03 |
| | | reda.004.001.02 |
| | | reda.005.001.02 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| seev | .001.xxx.xx-<br>.005.xxx.xx | seev.001.001.03<br>seev.001.001.04<br>seev.002.001.03<br>seev.002.001.04<br>seev.003.001.03<br>seev.003.001.04<br>seev.004.001.03<br>seev.004.001.04<br>seev.005.001.03<br>seev.005.001.04 |
| seev | .006.xxx.xx-<br>.010.xxx.xx | seev.006.001.03<br>seev.006.001.04<br>seev.007.001.03<br>seev.007.001.04<br>seev.008.001.03<br>seev.008.001.04<br>seev.009.001.01<br>seev.010.001.01 |
| seev | .011.xxx.xx-<br>.015.xxx.xx | seev.011.001.01<br>seev.012.001.01<br>seev.013.001.01<br>seev.014.001.01<br>seev.015.001.01 |
| seev | .016.xxx.xx-<br>.020.xxx.xx | seev.016.001.01<br>seev.017.001.01<br>seev.018.001.01<br>seev.019.001.01<br>seev.020.001.01 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| seev | .021.xxx.xx-<br>.025.xxx.xx | seev.021.001.01<br>seev.022.001.01<br>seev.023.001.01<br>seev.024.001.01<br>seev.025.001.01 |
| seev | .026.xxx.xx-<br>.030.xxx.xx | seev.026.001.01<br>seev.027.001.01<br>seev.028.001.01<br>seev.029.001.01<br>seev.030.001.01 |
| seev | .031.xxx.xx-<br>.035.xxx.xx | seev.031.001.01<br>seev.031.002.01<br>seev.032.001.01<br>seev.032.002.01<br>seev.033.001.01<br>seev.033.002.01<br>seev.034.001.01<br>seev.034.002.01<br>seev.035.001.01<br>seev.035.002.01 |
| seev | .036.xxx.xx-<br>.040.xxx.xx | seev.036.001.01<br>seev.036.002.01<br>seev.037.001.01<br>seev.037.002.01<br>seev.038.001.01<br>seev.038.002.01<br>seev.039.001.01<br>seev.039.002.01<br>seev.040.001.01<br>seev.040.002.01 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| seev | .041.xxx.xx-<br>.045.xxx.xx | seev.041.001.01<br>seev.041.002.01<br>seev.042.001.01<br>seev.042.002.01<br>seev.044.001.01<br>seev.044.002.01 |
| semt | .001.xxx.xx-<br>.005.xxx.xx | semt.001.001.01<br>semt.001.001.02<br>semt.002.001.01<br>semt.002.001.02<br>semt.002.001.03<br>semt.002.002.03<br>semt.003.001.01<br>semt.003.001.02<br>semt.003.001.03<br>semt.003.002.03<br>semt.004.001.01<br>semt.004.001.02<br>semt.005.001.01<br>semt.005.001.02 |
| semt | .006.xxx.xx-<br>.010.xxx.xx | semt.006.001.01<br>semt.006.001.02<br>semt.007.001.01<br>semt.007.001.02<br>semt.008.001.01<br>semt.008.001.02<br>semt.009.001.01<br>semt.009.001.02<br>semt.010.001.01 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| semt | .011.xxx.xx-<br>.015.xxx.xx | semt.011.001.01<br>semt.012.001.01<br>semt.013.001.01<br>semt.013.002.01<br>semt.014.001.01<br>semt.014.002.01<br>semt.015.001.01<br>semt.015.002.01 |
| semt | .016.xxx.xx-<br>.999.xxx.xx | semt.016.001.01<br>semt.016.002.01<br>semt.017.001.01<br>semt.017.002.01<br>semt.018.001.01<br>semt.018.002.01<br>semt.019.001.01<br>semt.019.002.01<br>semt.020.001.01<br>semt.020.002.01<br>semt.021.001.01<br>semt.021.002.01<br>semt.998.001.01 |
| sese | .001.xxx.xx-<br>.005.xxx.xx | sese.001.001.02<br>sese.002.001.02<br>sese.003.001.02<br>sese.004.001.02<br>sese.005.001.02 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| sese | .006.xxx.xx-<br>.010.xxx.xx | sese.006.001.02<br>sese.007.001.02<br>sese.008.001.02<br>sese.009.001.02<br>sese.010.001.02 |
| sese | .011.xxx.xx-<br>.015.xxx.xx | sese.011.001.02<br>sese.012.001.02<br>sese.013.001.02<br>sese.014.001.02 |
| sese | .016.xxx.xx-<br>.020.xxx.xx | sese.018.001.01<br>sese.019.001.01<br>sese.020.001.01<br>sese.020.002.01 |
| sese | .021.xxx.xx-<br>.025.xxx.xx | sese.021.001.01<br>sese.021.002.01<br>sese.022.001.01<br>sese.022.002.01<br>sese.023.001.01<br>sese.023.002.01<br>sese.024.001.01<br>sese.024.002.01<br>sese.025.001.01<br>sese.025.002.01 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| sese | .026.xxx.xx-<br>.030.xxx.xx | sese.026.001.01<br>sese.026.002.01<br>sese.027.001.01<br>sese.027.002.01<br>sese.028.001.01<br>sese.028.002.01<br>sese.029.001.01<br>sese.029.002.01<br>sese.030.001.01<br>sese.030.002.01 |
| sese | .031.xxx.xx-<br>.035.xxx.xx | sese.031.001.01<br>sese.031.002.01<br>sese.032.001.01<br>sese.032.002.01<br>sese.033.001.01<br>sese.033.002.01<br>sese.034.001.01<br>sese.034.002.01<br>sese.035.001.01<br>sese.035.002.01 |
| sese | .036.xxx.xx-<br>.999.xxx.xx | sese.036.001.01<br>sese.036.002.01<br>sese.037.001.01<br>sese.037.002.01 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| setr | .001.xxx.xx-<br>.005.xxx.xx | setr.001.001.02<br>setr.001.001.03<br>setr.002.001.02<br>setr.002.001.03<br>setr.003.001.02<br>setr.003.001.03<br>setr.004.001.02<br>setr.004.001.03<br>setr.005.001.02<br>setr.005.001.03 |
| setr | .006.xxx.xx-<br>.010.xxx.xx | setr.006.001.02<br>setr.006.001.03<br>setr.007.001.02<br>setr.007.001.03<br>setr.008.001.02<br>setr.008.001.03<br>setr.009.001.02<br>setr.009.001.03<br>setr.010.001.02<br>setr.010.001.03 |
| setr | .011.xxx.xx-<br>.015.xxx.xx | setr.011.001.02<br>setr.011.001.03<br>setr.012.001.02<br>setr.012.001.03<br>setr.013.001.02<br>setr.013.001.03<br>setr.014.001.02<br>setr.014.001.03<br>setr.015.001.02<br>setr.015.001.03 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| setr | .016.xxx.xx-<br>.020.xxx.xx | setr.016.001.02<br>setr.016.001.03<br>setr.017.001.02<br>setr.017.001.03<br>setr.018.001.02<br>setr.018.001.03 |
| setr | .045.xxx.xx-<br>.050.xxx.xx | setr.047.001.01<br>setr.048.001.01<br>setr.049.001.01<br>setr.050.001.01 |
| setr | .051.xxx.xx-<br>.055.xxx.xx | setr.051.001.01<br>setr.052.001.01<br>setr.053.001.01<br>setr.054.001.01<br>setr.055.001.01 |
| setr | .056.xxx.xx-<br>.060.xxx.xx | setr.056.001.01<br>setr.057.001.01<br>setr.058.001.01<br>setr.059.001.01<br>setr.060.001.01 |
| setr | .061.xxx.xx-<br>.999.xxx.xx | setr.061.001.01<br>setr.062.001.01<br>setr.064.001.01<br>setr.065.001.01<br>setr.066.001.01 |
| trea | .001.xxx.xx-<br>.005.xxx.xx | trea.001.001.02<br>trea.002.001.02<br>trea.003.001.02<br>trea.004.001.02<br>trea.005.001.02 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| trea | .006.xxx.xx-<br>.010.xxx.xx | trea.006.001.02<br>trea.007.001.02<br>trea.008.001.02<br>trea.009.001.02<br>trea.009.001.03<br>trea.010.001.02<br>trea.010.001.03 |
| trea | .011.xxx.xx-<br>.999.xxx.xx | trea.011.001.02<br>trea.011.001.03<br>trea.012.001.02<br>trea.012.001.03<br>trea.012.001.04<br>trea.013.001.01 |
| tsmt | .001.xxx.xx-<br>.005.xxx.xx | tsmt.001.001.03<br>tsmt.002.001.03<br>tsmt.003.001.03<br>tsmt.004.001.02<br>tsmt.005.001.02 |
| tsmt | .006.xxx.xx-<br>.010.xxx.xx | tsmt.006.001.03<br>tsmt.007.001.02<br>tsmt.008.001.03<br>tsmt.009.001.03<br>tsmt.010.001.03 |
| tsmt | .011.xxx.xx-<br>.015.xxx.xx | tsmt.011.001.03<br>tsmt.012.001.03<br>tsmt.013.001.03<br>tsmt.014.001.03<br>tsmt.015.001.03 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| tsmt | .016.xxx.xx-.020.xxx.xx | tsmt.016.001.03<br>tsmt.017.001.03<br>tsmt.018.001.03<br>tsmt.019.001.03<br>tsmt.020.001.02 |
| tsmt | .021.xxx.xx-.025.xxx.xx | tsmt.021.001.03<br>tsmt.022.001.02<br>tsmt.023.001.03<br>tsmt.024.001.03<br>tsmt.025.001.03 |
| tsmt | .026.xxx.xx-.030.xxx.xx | tsmt.026.001.02<br>tsmt.027.001.02<br>tsmt.028.001.03<br>tsmt.029.001.02<br>tsmt.030.001.03 |
| tsmt | .031.xxx.xx-.035.xxx.xx | tsmt.031.001.03<br>tsmt.032.001.03<br>tsmt.033.001.03<br>tsmt.034.001.03<br>tsmt.035.001.03 |
| tsmt | .036.xxx.xx-.040.xxx.xx | tsmt.036.001.03<br>tsmt.037.001.03<br>tsmt.038.001.03<br>tsmt.040.001.03 |
| tsmt | .041.xxx.xx-.045.xxx.xx | tsmt.041.001.03<br>tsmt.042.001.03<br>tsmt.044.001.01<br>tsmt.045.001.01 |

| Message Category | Range | Supported MX Message Schemas |
|---|---|---|
| tsmt | .046.xxx.xx-<br>.050.xxx.xx | tsmt.046.001.01<br>tsmt.047.001.01<br>tsmt.048.001.01<br>tsmt.049.001.01<br>tsmt.050.001.01 |
| tsmt | .051.xxx.xx-<br>.999.xxx.xx | tsmt.051.001.01<br>tsmt.052.001.01 |

## SDK Error Descriptions

Exceptions encountered during the execution of any of the SDK conversion services are captured in the output parameter *"errDoc."* This parameter contains the error code in either the *errDoc/ConversionErrors/Error[x]/Code* or the *errDoc/ValidationErrors/Error[x]/Code* field.

The following error codes may be generated:

- TC00100—A field was encountered that was not expected at this location.
- TC00101—The end of the MT message was encountered too soon.
- TC00102—There is a problem with the content of the field.
- TC00103—There is a problem with the field.
- TC00200—An element was encountered that was not expected at this location.
- TC00201—The end of the XML MT message was encountered too soon.
- TC00202—There is a problem with the content of the field.
- TC00901—There is a generic problem.

## wm.unifi Folder

**Important!** The following services have been deprecated: wm.unifi.convertXMLtoIData, wm.unifi.transportSAA, and wm.unifi.utils.validateRules. Use the services in the wm.unifi.validation folder instead.

# wm.unifi.convertXMLtoIData

Deprecated. This service converts an XML message into IData and validates it against schema and the generic rule book.

## Input Parameters

| | |
|---|---|
| *xmldata* | **String** XML message. |
| *MXIdentifier* | **String** The fully qualified name of the IS schema name for the specified XML document, for example, `wm.xmlv2.doc.camt_007_002_01:schema_camt_007_002_01`. |
| *validate* | **String** Indicates whether to validate this XML message against an XML schema and the SWIFT generic rule book. Valid values are `true` and `false`. |

## Output Parameters

| | |
|---|---|
| *document* | **IData** IData representing the XML input data. |
| *isValid* | **String** Indicates whether the XML data is a valid MX message. Valid values are `true` and `false`. |
| *errors* | **IData** List of errors, if any. |

# wm.unifi.tranportSAA

Deprecated. This service sends an MX/MT message to SAA in XMLv2 format.

## Input Parameters

| | |
|---|---|
| *msgRef* | **String** Message reference identifier. |
| *payload* | **String** MX/MT message payload. |
| *serviceName* | **String** The name of the SWIFT messaging service. |
| *messageIdentifier* | **String** Message identifier. |
| *requestor* | **String** The DN name of the sender (used for MX messages). |
| *responder* | **String** The DN name of the receiver (used for MX messages). |
| *filename* | **String** The full path of the output file. This file should belong to the directory configured as the inbound AFT in SAA (if using AFT mode). |
| *adapterService* | **String** The name of the WebSphere MQ Adapter service (if MQSA mode of transport is used). |
| *format* | **String** Specifies the payload format. Valid values are `MT` and `MX`. |

| | |
|---|---|
| senderBIC | **String** The BIC code of the sender (used for MT messages). |
| receiverBIC | **String** BIC code of receiver (used for MT messages). |

### Output Parameters

None.

## wm.unifi.utils.validateRules

Deprecated. This service validates an MX message for schema conformance and proper rule definition as outlined in the SWIFT generic rule book.

### Input Parameters

| | |
|---|---|
| *Object* | **Object** An XML element Object instance of an MX message type. |
| *conformsTo* | **String** The fully qualified name of the IS Schema against which the XML string must be validated. |
| *validate* | **String** Indicates whether to validate this XML message against an XML schema, and the SWIFT generic rule book. Valid values are `true` and `false`. |

### Output Parameters

| | |
|---|---|
| *isValid* | **String** Indicates whether the XML data is a valid MX message. Valid values are `true` and `false`. |
| *errors* | **IData** List of errors, if any. |

### wm.unifi.validation Folder

The services in this folder validate an MX message against the SWIFT generic rule book.

## wm.unifi.validation:validateBEI

This service verifies if the BEI code exists in the SWIFT database.

**Note:** By default the service returns `true` when the input *xmlNode* does not contain a node element with QName as the specified *TypeName*. In this case the following optional warning message is returned: “No node with the specified TypeName could be found for the entered xmlNode.”

### Input Parameters

| | |
|---|---|
| *xmlNode* | **Object** An XML element object instance of an MX message type. |

|  |  |
|---|---|
| *TypeName* | String The tag name that identifies the element containing the BEI code. The service extracts the value of the element using the type name to identify it and validate the data value. |
| *ErrorCode* | String The default value for the error code is `Sw.Stds.D0002`. You can replace it with a custom error code. |

### Output Parameters

|  |  |
|---|---|
| *isValid* | String The result of the MX string validation. Valid values are `true` or `false`. |
| *errors* | Document List Conditional. Errors that occur when the MX message does not pass validation. |
| *error/pathName* | String The complete xpath of the element for which validation failed. |
| *error/errorCode* | String The error code for the corresponding element. |
| *error/errorMessage* | String The detailed message for the error that occurred. |
| *error/errorData* | String The field content for which validation failed. |

## wm.unifi.validation:validateBIC

This service verifies if the BIC code exists in the SWIFT database.

Note: By default the service returns `true` when the input *xmlNode* does not contain a node element with QName as the specified *TypeName*. In this case you receive the following optional warning message: “No node with the specified TypeName could be found for the entered xmlNode.”

### Input Parameters

|  |  |
|---|---|
| *xmlNode* | Object An XML element Object instance of an MX message type. |
| *TypeName* | String The tag name identifying the element that contains the BIC code. The service extracts the value of the element using the type name to identify it and validate the data value. |
| *ErrorCode* | String The default value for the error code is `Sw.Stds.D0001`. You can replace it with a custom error code. |

### Output Parameters

|  |  |
|---|---|
| *isValid* | String The result of the MX string validation. Valid values are `true` or `false`. |
| *errors* | Document List Conditional. Errors that occur when the MX message does not pass validation. |

*error/pathName* String The complete xpath of the element for which validation failed.

*error/errorCode* String The error code for the corresponding element.

*error/errorMessage* String The detailed message for the error that occurred.

*error/errorData* String The field content for which validation failed.

## wm.unifi.validation:validateCountryCode

This service verifies if the country code exists in the SWIFT database.

**Note:** By default the service returns `true` when the input *xmlNode* does not contain a node element with QName as the specified *TypeName*. In this case you receive the following optional warning message: “No node with the specified TypeName could be found for the entered xmlNode.”

### Input Parameters

*xmlNode* Object An XML element Object instance of an MX message type.

*TypeName* String The tag name identifying the element that contains the country code. The service extracts and validates this value, using the type name to identify it.

*ErrorCode* String The default value for the error code is `Sw.Stds.D0004`. You can replace it with a custom error code.

### Output Parameters

*isValid* String The result of the MX string validation. Valid values are `true` or `false`.

*errors* Document List Conditional. Errors that occur when the MX message does not pass validation.

*error/pathName* String The complete xpath of the element for which validation failed.

*error/errorCode* String The error code for the corresponding element.

*error/errorMessage* String The detailed message for the error that occurred.

*error/errorData* String The field content for which validation failed.

# wm.unifi.validation:validateCurrencyCode

This service verifies if the currency code exists in the SWIFT database.

**Note:** By default the service returns `true` when the input *xmlNode* does not contain a node element with QName as the specified *TypeName*. In this case you receive the following optional warning message: "No node with the specified TypeName could be found for the entered xmlNode."

## Input Parameters

| | |
|---|---|
| *xmlNode* | **Object** An XML element object instance of an MX message type. |
| *TypeName* | **String** The tag name that identifies the element containing the currency code. The service extracts the value of the element using the type name to identify it and validate the data value. |
| *AttrName* | **String** The attribute name that contains the value for the currency code. |
| *ErrorCode* | **String** The default value for the error code is `Sw.Stds.D0005`. You can replace it with a custom error code. |

## Output Parameters

| | |
|---|---|
| *isValid* | **String** The result of the MX string validation. Valid values are `true` or `false`. |
| *errors* | **Document List** Conditional. Errors that occur when the MX message does not pass validation. |
| *error/pathName* | **String** The complete xpath of the element for which validation failed. |
| *error/errorCode* | **String** The error code for the corresponding element. |
| *error/errorMessage* | **String** The detailed message for the error that occurred. |
| *error/errorData* | **String** The field content for which validation failed. |

# wm.unifi.validation:validateIBAN

This service verifies if the IBAN exists in the SWIFT database.

**Note:** By default the service returns `true` when the input *xmlNode* does not contain a node element with QName as the specified *TypeName*. In this case you receive the following optional warning message: "No node with the specified TypeName could be found for the entered xmlNode."

### Input Parameters

| | |
|---|---|
| *xmlNode* | **Object** An XML element Object instance of an MX message type. |
| *TypeName* | **String** The tag name that identifies the element containing the IBAN. The service extracts the value of the element using the type name to identify it and validate the data value. |
| *ErrorCode* | **String** The default value for the error code is `Sw.Stds.D0003`. You can replace it with a custom error code. |

### Output Parameters

| | |
|---|---|
| *isValid* | **String** The result of the MX string validation. Valid values are `true` or `false`. |
| *errors* | **Document List** Conditional. Errors that occur when the MX message does not pass validation. |
| *error/pathName* | **String** The complete xpath of the element for which validation failed. |
| *error/errorCode* | **String** The error code for the corresponding element. |
| *error/errorMessage* | **String** The detailed message for the error that occurred. |
| *error/errorData* | **String** The field content for which validation failed. |

## wm.unifi.validation:validateMXMsg

This service performs different validations of the MX message.

### Input Parameters

| | |
|---|---|
| *xmlNode* | **Object** An XML element object instance of an MX message type. |
| *XMLV2Params* | **Document Reference** The XMLV2Params document reference. You can configure various parameters in this document to trigger different validations for the input XMLv2 MX message instance. |

### Output Parameters

| | |
|---|---|
| *isValid* | **String** The result of the MX string validation. Valid values are `true` or `false`. |
| *errors* | **Document List** Conditional. Errors that occur when the MX message does not pass validation. |
| *error/pathName* | **String** The complete xpath of the element which failed validation. |
| *error/errorCode* | **String** The error code for the corresponding element. |
| *error/errorMessage* | **String** The detailed message for the error that occurred. |

*error/errorData* **String** The field content for which validation failed.

*Message* **String** Provides additional information during validation, for example, “Skipping non-schema validations” and “Skipping all MX validations.”

## Process Information Section of the XMLv2 Parameters Document

<table>
<tr><th>Parameter Name</th><th>Description</th></tr>
<tr><td><i>Validate</i></td><td>Indicates to the transport service whether to validate the MX message.</td></tr>
<tr><td><i>Schema</i><br><i>Validation</i></td><td>The parameters in this subsection indicate whether to perform schema validation for the MX message instance.
<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td><i>conformTo</i></td><td>Takes the fully qualified name of the IS schema against which schema will be performed.</td></tr>
<tr><td><i>Validate</i></td><td>Indicates whether to perform schema validation. Valid values are <code>true</code> and <code>false</code>.</td></tr>
<tr><td><i>ValidateContent</i></td><td>Indicates whether to perform content validation. Valid values are <code>true</code> and <code>false</code>.</td></tr>
</table></td></tr>
<tr><td><i>NonSchemaValidation</i></td><td>The parameters in this subsection indicate if a non-schema (extended) validation must be performed for the MX message.
<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td><i>Validate/BIC</i></td><td>These parameters indicate whether to perform BIC validation for the MX message.
<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td><i>Validate</i></td><td>Indicates whether to perform BIC validation. Valid values are <code>true</code> and <code>false</code>.</td></tr>
<tr><td><i>ErrorCode</i></td><td>Error code generated if validation fails. Default value is <code>Sw.Stds.D00001</code>.</td></tr>
<tr><td><i>TypeName</i></td><td>Tag ID for the BIC field.</td></tr>
</table></td></tr>
</table></td></tr>
</table>

| Parameter Name | Description | | |
|---|---|---|---|
| | Name | Description | |
| | *Validate/BEI* | These parameters indicate whether to perform BEI validation for the MX message instance. | |
| | | Name | Description |
| | | *Validate* | Indicates whether to perform BEI validation. Valid values are `true` and `false`. |
| | | *ErrorCode* | Error code generated if validation fails. Default value is `Sw.Stds.D00002`. |
| | | *TypeName* | Tag ID for the BEI field. |
| | Name | Description | |
| | *Validate/IBAN* | These parameters indicate whether to perform IBAN validation for the MX message. | |
| | | *Validate* | Indicates whether to perform IBAN validation. Valid values are `true` and `false`. |
| | | *ErrorCode* | Error code generated if validation fails. Default value is `Sw.Stds.D00003`. |
| | | *TypeName* | Tag identifier for the IBAN field. |
| | Name | Description | |
| | *Validate/Country* | These parameters indicate whether to perform country code validation for the MX message. | |
| | | Name | Description |
| | | *Validate* | Indicates whether to perform country code validation. Valid values are `true` and `false`. |
| | | *ErrorCode* | Error code generated if validation fails. Default value is `Sw.Stds.D00004`. |
| | | *TypeName* | Tag identifier for the country code field in the MX message. |

<table>
<tr><th>Parameter Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>*Validate/CurrencyCode*</td><td>These parameters indicate whether to perform currency code validation for the MX message.
<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>*Validate*</td><td>Indicates whether to perform currency code validation. Valid values are `true` and `false`.</td></tr>
<tr><td>*ErrorCode*</td><td>Error code generated if validation fails. Default value is `Sw.Stds.D00005`.</td></tr>
<tr><td>*TypeName*</td><td>Tag identifier for the currency code field in the MX message.</td></tr>
<tr><td>*AttrName*</td><td>Attribute name that contains the value for the currency code.</td></tr>
</table>
</td></tr>
</table>
</td></tr>
<tr><td></td><td>
<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>*Validate/CurrencyAmount*</td><td>These parameters indicate whether to perform currency amount validation for the MX message.
<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>*Validate*</td><td>Indicates whether to perform currency amount validation. Valid values: `true` and `false`.</td></tr>
<tr><td>*ErrorCode*</td><td>Error code generated if validation fails. Default value is `Sw.Stds.D00007`.</td></tr>
<tr><td>*TypeName*</td><td>Tag ID for the currency amount field.</td></tr>
</table>
</td></tr>
</table>
</td></tr>
</table>

## wm.xmlv2.dev Folder

The service in this folder creates Trading Networks items for a particular message type.

# wm.xmlv2.dev:createSWIFTItems

This service creates a Data PDU document for a particular MT or MX message type.

## Input Parameters

*msgTypeName*
**String** The type of message, for example `fin.101` (MT type) or `camt.029.001.01` (MX type). The message type identifies the particular message type from the Data PDU.

*format*
**String** Valid values are `MT` or `MX`.

*finFormat*
**String** Optional. Defines the format of the IS document generated for the MT message type. The following values specify how the element is formatted within the IS document:

- `TAG_BIZNAME` (default)-The SWIFT message tag, followed by the business name, for example, `23G`_*Function of the Message*.
- `TAGONLY`. The SWIFT Message tag only, for example 23G.
- `BIZNAMEONLY`. The business name of the field only, for example, `A-Account Servicing Institution`.
- `XMLTAG`. XML-compatible tag name. This format cannot contain colons or tags that begin with a number, for example, `F52A`.

*version*
**String** Optional (required only for MT messages). Specifies the version of the SWIFT specification (for example, nov10).

*subfieldFlag*
**String** Optional. Specifies whether to parse the fields in the IS document type generated for this MT message to the subfield level. Valid values:

- `true` (default). Parse to the subfield level.
- `false`. Parse to the field level.

*createProcessingRule*
**String** Optional. Creates a default processing rule for the specified message type. Valid values are `true` and `false`. The default is `false`.

**Note:** The SWIFT Module samples contain a sample that provides more details for this parameter. For information about the samples, see *webMethods SWIFT Module Samples Guide*.

*createTPA*
**String** Optional. Creates a Trading Networks TPA for this message that specifies variables used in WmFIN for processing and validation. Valid values: `true` or `false`. The default is `true`.

*createDocType* String Optional. Indicates whether to create and insert a TN document type for this message (used to recognize an incoming message). Valid values: `true` or `false`. The default is `true`.

### Output Parameters

None.

## wm.xmlv2.doc Folder

The service in this folder allows you to use a TPA for a message type and configure all necessary information to enable document processing.

## wm.xmlv2.doc:XMLV2Params

The document type used as the TPA document. After creating a TPA for the corresponding message type, you can configure all necessary information for processing a document.

## wm.xmlv2.notifications Folder

The service in this folder handles incoming delivery notifications.

## wm.xmlv2.notifications:handleDeliveryNotifications

This service processes incoming MT and MX message documents as follows:

- MT Message Delivery Notification Processing: Extracts the MIR from the Delivery Notification and searches the documents in the Trading Networks database for the Sender Reference that matches the MIR extracted from the notification.

  When the search is successful, the service relates the Delivery Notification to the search results message as a “Delivery Notification.”

- MX Message Delivery Notification Processing: Extracts the *ReconciliationInfo* value from the Delivery Notification and searches the Trading Networks database for the corresponding Transmission Report with the matching *ReconciliationInfo* value.

  When the search is successful, the service relates the Delivery Notification to the Transmission Report of the MX message. The Transmission Report is, in turn, already related to the MX message that was sent.

When the relevant record is found in the Trading Networks database, this service changes the status of the Delivery Notification to “reconciled.”

### Input Parameters

| | |
|---|---|
| bizdoc | **Object** The TN document type of the bizdoc. |

### Output Parameters

None.

## wm.xmlv2.process Folder

This folder contains services that apply processing rules to documents exchanged over SAA.

## wm.xmlv2.process:createSAADoc

This utility service converts a DataPDU XML element into IS DataPDU.

### Input Parameters

| | |
|---|---|
| *xmldata* | **String** DataPDU in XML format. |

### Output Parameters

| | |
|---|---|
| *DataPDU* | **Document Reference** An instance of the wm.swift.doc:saa_2 IS document type. |

## wm.xmlv2.process:getInboundMessageType

This service identifies the category of the Data PDU input, provided in XML format, and processes it to determine the type of the incoming document from SAA.

### Input Parameters

| | |
|---|---|
| *xmldata* | **String** The Data PDU of the incoming document in XML format. |

### Output Parameters

| | |
|---|---|
| *Type* | **String** The type of the incoming document from SAA. |
| *TransmissionReport* | **String** Conditional. The incoming document is a Transmission Report. |

*DeliveryNotification* **String** Conditional. The incoming document is a Delivery Notification.

*DeliveryReport* **String** Conditional. The incoming document is a Delivery Report.

*HistoryReport* **String** Conditional. The incoming document is a History Report.

*MessageStatus* **String** Conditional. The incoming document is a Message Status.

*SessionStatus* **String** Conditional. The incoming document is a Session Status.

## wm.xmlv2.process:outbound

This service processes an outbound Trading Networks bizdoc object. This service sends the Data PDU in XML format to SAA, using the transport type specified in the corresponding TPA. (SWIFT Module supports two types of transport: MQ or FTA.)

### Input Parameters

*bizdoc* **Object** The TN document type of the bizdoc.

### Output Parameters

None.

## wm.xmlv2.process:processInbound

This service processes all inbound documents from SAA to SWIFT Module, performing the preliminary processing of the inbound Data PDU and submitting it to Trading Networks for further processing.

### Input Parameters

*xmldata* **String** The Data PDU in XML format.

### Output Parameters

None.

## wm.xmlv2.process:reconcileInboundDocuments

This service reconciles all incoming notifications from SAA. It identifies the category of the notification and relates the notification to the original document based on the Sender Reference of the document attribute. It also updates the status of the original document depending on the incoming notification.

### Input Parameters

| | |
|---|---|
| *xmldata* | **String** Content of the inbound MX message received from SAA in *bizdoc* format. |

### Output Parameters

None.

## wm.xmlv2.transport Folder

The service in this folder handles the routing of an incoming message in XML format.

## wm.xmlv2.transport:submitDataPDU

This service persists the DataPDU in Trading Networks and routes the resultant bizdoc for further processing.

### Input Parameters

| | |
|---|---|
| *xmldata* | **String** Populates the SenderReference and UserReference elements in the message header. |

### Output Parameters

None.

## wm.xmlv2.utils Folder

This folder contains utility services for message encoding.

## wm.xmlv2.utils:encodeBlock4

This service encodes block 4 of an MT message to a base64 string.

### Input Parameters

| | |
|---|---|
| *block4* | **String** The block4 contents of the MT message. |

### Output Parameters

| | |
|---|---|
| *encodedBlock* | **String** The base-64- encoded block 4 of the MT message included in the body section of the Data PDU for transport. |

## wm.xmlv2.utils:encodeFinMessage

This service encodes block 4 of a FIN message to a base64 string. It extracts the block 4 contents from the input FIN message and encodes it to a base64 string.

### Input Parameters

| | |
|---|---|
| *finMsg* | **String** The block 4 contents of the FIN message. |

### Output Parameters

| | |
|---|---|
| *encodedMsg* | **String** The base-64-encoded block 4 of the FIN message. |

## wm.xmlv2.utils:formatXMLV2

This service formats the XML contents of the Data PDU to a proper XMLv2 format, as follows:

- Prefix (1 byte): the hexadecimal character 0x1f.
- Length (6 bytes): the length (in bytes) of the Signature and Data PDU fields. The length is base-10 encoded as 6 ASCII characters, left padded with 0s, if needed.
- Signature (24 bytes): the Signature computed on the Data PDU using the HMAC-SHA256 algorithm, base-64 encoded. This signature authenticates the originator of the Data PDU (the application or SAA), and guarantees the integrity of the Data PDU. If this authentication is not required on the SAA side, the field must be filled with NULL characters.

  **Note:** SWIFT Module does not support signature generation and the signature field is always filled with NULL characters.

- Data PDU: the XML structure that contains the information relevant for processing the document encoded in UTF--8 format. The first byte of this field must be the character '<' (`0x3C`): byte-order marker is not supported.

### Input Parameters

| | |
|---|---|
| *message* | String The Data PDU in XML format. |
| *filename* | String The fully qualified file name where the generated XMLv2 message is added after generation. |

### Output Parameters

| | |
|---|---|
| *outMessage* | String The generated formatted XMLv2 message. |

## wm.xmlv2.utils:getDataPDUsFromFile

This service takes a batch file containing the Data PDUs as input and extracts all the Data PDUs from the file. It also generates a string array of the Data PDUs.

### Input Parameters

| | |
|---|---|
| *filePath* | String The fully qualified name of the batch file containing multiple Data PDUs. |

### Output Parameters

| | |
|---|---|
| *DataPDUs* | String List The Data PDUs extracted from the batch file. |

## wm.xmlv2.utils:putInBatchFile

This service creates a batch file of the Data PDUs that is submitted to SAA for processing.

### Input Parameters

| | |
|---|---|
| *message* | String The Data PDU in XML format that will be formatted and added to the batch file. |
| *filename* | String The fully qualified file name where the formatted Data PDU will be added. |

### Output Parameters

| | |
|---|---|
| *outMessage* | String List Formatted Data PDU in XMLv2 format. |

# WmSWIFTCommon Package

This package contains common services that are used by different packages. This package contains the following folders:

| Folder | Contains services you use to... |
|---|---|
| com.wm.common.CacheHandler Folder | Establish the security context for the message partner from the shared cache. |
| com.wm.common.docs Folder | Contains IS documents to be used by other SWIFT Module packages. |
| com.wm.common.Init Folder | Create user interface components for SWIFT Module. |
| com.wm.common.services Folder | Process inbound and outbound messages transported using MQHA. |
| com.wm.common.Util Folder | This folder contains various common utility services. |
| wm.swift.doc Folder | Imported document types and schema generated from saa_2.xsd (provided by SWIFT). |

## com.wm.common.CacheHandler Folder

This folder contains services that establish the security context for the message partner from the shared cache.

## com.wm.common.CacheHandler.getContextForMessagePartner

This service retrieves the security context for the specified message partner from the shared cache.

### Input Parameters

| | |
|---|---|
| *messagePartner* | **String** The message partner specified for the message exchange. |

### Output Parameters

| | |
|---|---|
| *securityContext* | **String** The security context retrieved from the shared cache for the message partner specified in the input. |

## com.wm.common.CacheHandler.saveContextForMessagePartner

This service saves the security context for the message partner in the cache after the initialization process is complete.

### Input Parameters

| | |
|---|---|
| *messagePartner* | String The message partner specified for the message exchange. |
| *securityContext* | String The security context retrieved from the shared cache for the message partner you specified in the input. |

### Output Parameters

| | |
|---|---|
| *result* | String Saved security context in the shared cache. |

## com.wm.common.docs Folder

This folder contains IS documents to be used by other SWIFT Module packages.

## com.wm.common.Init Folder

The services in this folder create user interface components for SWIFT Module.

## com.wm.common.services Folder

The services in this folder perform various tasks such as:

- Create Trading Networks BizDocEnvelope for the MQ response.
- Retrieve the SAG response envelope and the response message from the MQ response.
- Submit the MQ response, as well as any outgoing request, to SWIFT via Trading Networks.

## com.wm.common.services.createTNDocForMQResponse

This service creates a TN document type for the response received from MQHA. The TN document type, created to represent the MQ response, contains two content parts: msgcontent as `xmldata` and SAG envelope as `sagenv`.

### Input Parameters

*contextResponse* **String** The MQ response.

### Output Parameters

*bizdoc* **Document Reference** Trading Networks BizDocEnvelope containing the SWIFT message.

## com.wm.common.services.getEnvAndXMLReqFromMQResponse

This service breaks down the MQ response into a SAG envelope and a response message.

### Input Parameters

*mqResponse* **String** The MQ response.

### Output Parameters

*sagenv* **String** The SAG response envelope.

*xmldata* **String** The response message as XML data.

## com.wm.common.services.getSagEnv

This service retrieves the *sagenv* from the *bizdoc*. The input *bizdoc* provided to this service must have *sagenv* as one of the content parts.

### Input Parameters

*bizdoc* **Document** Trading Networks BizDocEnvelope containing the SWIFT message.

### Output Parameters

*sagenv* **String** The SAG response envelope.

## com.wm.common.services.getSagReqEnvAsString

This service creates a string representation of the SAG response envelope. This service internally uses wmFlatFile services to create a string *sagenv*.

### Input Parameters

| | |
|---|---|
| *sagReqDoc* | Document Reference The SAG response envelope from the MQ response. |

### Output Parameters

| | |
|---|---|
| *sagenv* | String The SAG response envelope. |

## com.wm.common.services.getXMLData

This service retrieves the SWIFT request or response XML data from the TN BizDocEnvelope. The input *bizdoc* provided to this service must have *xmldata* as one of the content parts.

### Input Parameters

| | |
|---|---|
| *bizdoc* | Document Trading Networks BizDocEnvelope containing the SWIFT message. |

### Output Parameters

| | |
|---|---|
| *xmldata* | String The SWIFT response or request XML data. |

## com.wm.common.services.handleContextResponse

This service saves the security context for the message partner obtained in the context response from SWIFT, provided such a context has been successfully created.

### Input Parameters

| | |
|---|---|
| *bizdoc* | Document Trading Networks BizDocEnvelope containing the SWIFT message. |

### Output Parameters

None.

## com.wm.common.services.submitContextResponse

This service routes the MQ response to Trading Networks.

### Input Parameters

*contextResponse* **String** The MQ response.

### Output Parameters

*bizdoc* **Document** Trading Networks BizDocEnvelope containing the SWIFT message.

## com.wm.common.services.submitMQResponseToTN

This service submits the MQ response to Trading Networks. This service adds *sagenv*, *correlationId* and *msgId* as content parts of the bizdoc created for the XML data.

### Input Parameters

*xmldata* **String** The message response as XML data.

*sagenv* **String** The SAG response envelope.

*correlationId* **Object** The IBM MQ correlation ID included in the JMS headers of the response sent back to SAG.

*msgId* **Object** The IBM MQ message ID included in the JMS headers of the request sent to the server application from SAG.

### Output Parameters

*bizdoc* **Object** Trading Networks BizDoc object.

## com.wm.common.services.submitRequestToTN

This service submits any outgoing request to SWIFT through Trading Networks.

### Input Parameters

*xmldata* **String** The message response as XML data.

*sagReqDoc* **Document Reference** The SAG response envelope from the MQ response.

### Output Parameters

*bizdoc* **Object** Trading Networks BizDoc object.

## com.wm.common.Util Folder

This folder contains various common utility services.

# com.wm.common.Util.createSagReqEnv

This service creates a SAG request envelope for the message partner and the security context.

### Input Parameters

*messagePartner* String The message partner specified for the message exchange.

*securityContext* String The security context retrieved from the shared cache for the message partner you specified in the input.

### Output Parameters

*sagReqEnv* Document Reference The SAG request envelope.

# com.wm.common.Util.invokeMQService

This service invokes the MQ service identified by the service name.

### Input Parameters

*msgBodyByteArray* bytes Message payload for the MQ service in byte format.

*serviceName* String The name of the MQ service.

### Output Parameters

*responseByteArray* bytes The response after invoking the MQ service.

# com.wm.common.Util:migrateServices

This service iterates through all flow services in a particular package and replaces all occurrences of old service names with the corresponding new service names from the service map maintained by SWIFT Module. The service also creates a backup of the previous version of the modified flow service, and stores it as flow.xml.previous.

**Important!** You must reload the package specified in the input parameters manually after running the migration service for changes to take effect.

### Input Parameters

*packageName* **String** The name of the package where the migration utility will look for flow services.

*oldServiceNames* **String List** Old service names from the WmIPCore package.

### Output Parameters

*results* **Document List** Conditional. The results from the flow service mapping.

*oldServiceName* **String** The name of the old service for which the result document is created.

*newServiceName* **String** Conditional. The name of the new service that replaces the old one.

*message* **String** Conditional. A message that this service returns when no occurrences of the old service were found in the package.

*filesChanged* **String List** Conditional. List of files in which the name of the old service was replaced by the new service.

## com.wm.common.Util.resolveNameSpaceAndEntity

This service resolves any namespace prefixes for the XML request. This service adds namespace declarations for all SAG and SNL primitive prefixes encountered in the XML message.

### Input Parameters

*xmldata* **String** Input XML for which namespace resolution is required.

### Output Parameters

*xmldata* **String** Formatted XML with declarations for namespace prefixes.

## wm.swift.doc Folder

This folder contains imported document types and schema generated from saa_2.xsd (provided by SWIFT). This schema is used to format message envelopes in XML v2 format as specified by SWIFT.

## WmEstdCommonLib Package

This package contains generic services that enable you to use various eStandards Modules with webMethods Integration Server. SWIFT Module uses the wm.estd.common.rec folder and the following services from this package:

- wm.estd.common.bizdoc:addErrorContentPart
- wm.estd.common.profile:getTPA
- wm.estd.common.ui:addSubmenu
- wm.estd.common.ui:removeSubmenu
- wm.estd.common.util:invokeService
- wm.estd.common.util:writeToFile

For detailed information about the folder and the services that SWIFT Module uses from this package, see *webMethods eStandards Modules Common Built-In Services Reference*.

## WmSWIFTNetClient Package

This package contains the elements (flow services, Java services, record descriptions, and wrapper services) that support webMethods SWIFTNet client-side functionality. This package contains the following folders:

| Folder | Contains services to... |
|---|---|
| wm.swiftnet.client.doc Folder | The NS records that represent the SNL primitives exchanged for FileAct and InterAct operations. |
| wm.swiftnet.client.init Folder | Start and terminate the client process. |
| wm.swiftnet.client.mq Folder | Send requests from your client application to SWIFT over the MQ transport. |
| wm.swiftnet.client.property Folder | Load properties specified in the *Integration Server_directory*\ packages\ WmSWIFTNetClient\config\snl.cnf file. |
| wm.swiftnet.client.services Folder | Exchange SNL primitives with SAG over RA. |
| wm.swiftnet.client.transport Folder | Transfer files from your client application to SAG using the FTA interface. |

| Folder | Contains services to... |
|---|---|
| wm.swiftnet.client.util Folder | Various utility services. |

# wm.swiftnet.client.doc Folder

This folder contains the NS records that represent the SNL primitives exchanged for FileAct and InterAct operations.

# wm.swiftnet.client.init Folder

This folder contains the services that start and terminate the client process, and create the user interface links for the SWIFTNet client configuration.

## wm.swiftnet.client.init:printRemoteErrors

This service logs the standard output and standard error from the client process that is connected to SAG. The errors are logged to the Integration Server console. This service must be used only to trace an error and not used otherwise.

### Input Parameters

None.

### Output Parameters

None.

## wm.swiftnet.client.init:shutdown

This service is registered as a shutdown service for the WmSWIFTNetClient package. It terminates the client process that is connected to SAG.

### Input Parameters

None.

### Output Parameters

None.

## wm.swiftnet.client.init:startup

This service starts a client process that connects to SAG. This client process connects to SAG whenever a request needs to be sent over SWIFTNet.

### Input Parameters

None.

### Output Parameters

None.

## wm.swiftnet.client.mq Folder

This folder contains services that send requests from your client application to SWIFT over the MQ transport.

## wm.swiftnet.client.mq:processRequest

This service sends the XML request to SWIFT over the MQ transport. This service creates the SAG envelope to be submitted to SWIFT.

### Input Parameters

| | |
|---|---|
| *messagePartner* | String The message partner specified for the message exchange. |
| *xmldata* | String The XML request message. |

### Output Parameters

| | |
|---|---|
| *responseXml* | String The response XML to be submitted to SWIFT. |

## wm.swiftnet.client.mq:sendToMQ

This service gets the *sagenv* and *xmldata* from the Trading Networks BizDocEnvelope and creates an MQ request to be sent to SWIFT over the MQ transport.

### Input Parameters

| | |
|---|---|
| *bizdoc* | Document The Trading Networks BizDocEnvelope containing the SWIFT message. |

### Output Parameters

| | |
|---|---|
| *string* | String The MQ request to be sent to SWIFT over MQ. |

# wm.swiftnet.client.property Folder

This folder contains a service that loads properties specified in the *Integration Server_directory*\packages\WmSWIFTNetClient\config\snl.cnf file.

## wm.swiftnet.client.property:getProperty

This service retrieves the value of the specified property from the *Integration Server_directory*\packages\WmSWIFTNetClient\config\snl.cnf file.

### Input Parameters

| | |
|---|---|
| *propertyName* | String Property value to be retrieved. |

### Output Parameters

| | |
|---|---|
| *value* | String Value of the property. |

# wm.swiftnet.client.services Folder

This folder contains services that exchange SNL primitives with SAG over RA. The wm.swiftnet.client.services:swArguments service must be invoked prior to invoking any other services in this folder.

The services in this folder can be invoked in a predefined sequence to perform FileAct and InterAct real-time and SnF operations. In essence the services in this folder are the building blocks to perform higher level FileAct and InterAct operations.

## wm.swiftnet.client.services:createContextRequest

This service requests SAG to create a security context. It sends the SwSec:CreateContextRequest to SAG over RA and returns the SwSec:CreateContextResponse received from SAG.

### Input Parameters

| | |
|---|---|
| *SwSecCreateContext Request* | Document Reference Request to create a security context. |

### Output Parameters

| | |
|---|---|
| *SwSecCreateContext Response* | Document Reference Response indicating success or failure of security context creation in SAG. |
| *error* | String Whether an error occurred. Valid values: `true` and `false`. |

*errorXMLString* String Conditional. Error details received from SAG.

## wm.swiftnet.client.services:destroyContextRequest

This service requests SAG to destroy a security context. The service sends the SwSec:DestroyContextRequest to SAG over RA and returns the SwSec:DestroyContextResponse received from SAG.

### Input Parameters

*SwSecDestroyContextRequest* Document Reference Request to destroy a security context.

### Output Parameters

*SwSecDestroyContextResponse* Document Reference Response indicating success or failure of security context destruction in SAG.

*error* String Whether an error occurred. Valid values: `true` and `false`.

*errorXMLString* String Conditional. Error details received from SAG.

## wm.swiftnet.client.services:exchangeFileRequest

This service requests SAG to perform a FileAct operation (real-time get file or put file, and SnF put file). The information whether to put a file or get a file is specified in the Sw:ExchangeFileRequest primitive. The service sends the Sw:ExchangeFileRequest to SAG over RA and returns the Sw:ExchangeFileResponse received from SAG.

### Input Parameters

*SwExchangeFileRequest* Document Reference Request to perform a FileAct operation.

### Output Parameters

*SwExchangeFileResponse* Document Reference Response indicating success or failure of the FileAct operation.

*error* String Whether an error occurred. Valid values: `true` and `false`.

*errorXMLString* String Conditional. Error details received from SAG.

# wm.swiftnet.client.services:exchangeRequest

This service requests SAG to send an InterAct message. The service sends the SwInt:ExchangeRequest to SAG over RA and returns the SwInt:ExchangeResponse received from SAG.

## Input Parameters

| | |
|---|---|
| *SwIntExchangeRequest* | Document Reference Request to exchange a synchronous request. |

## Output Parameters

| | |
|---|---|
| *SwIntExchangeResponse* | Document Reference Synchronous response received. |
| *error* | String Whether an error occurred. Valid values: `true` and `false`. |
| *errorXMLString* | String Conditional. Error details received from SAG. |

# wm.swiftnet.client.services:exchangeSnFRequest

This service requests SAG to send a SnF message. The service sends the Sw:ExchangeSnFRequest to SAG over RA and returns the Sw:ExchangeSnFResponse received from SAG.

## Input Parameters

| | |
|---|---|
| *SwExchangeSnFRequest* | Document Reference Request related to SnF protocol, for example, acquire a queue. |

## Output Parameters

| | |
|---|---|
| *SwExchangeSnFResponse* | Document Reference Response returned by SWIFTNet. |
| *error* | String Whether an error occurred. Valid values: `true` and `false`. |
| *errorXMLString* | String Conditional. Error details received from SAG. |

# wm.swiftnet.client.services:fetchFileRequest

This service sends the Sw:FetchFileRequest to SAG over RA and returns the Sw:FetchFileResponse received from SAG.

## Input Parameters

| | |
|---|---|
| *SwFetchFileRequest* | Document Reference Request to fetch a file from an SnF queue. |

### Output Parameters

| | |
|---|---|
| *SwFetchFileResponse* | Document Reference Response returned by SWIFTNet for a fetch file request. |
| *error* | String Whether an error occurred. Valid values: `true` and `false`. |
| *errorXMLString* | String Conditional. Error details received from SAG. |

## wm.swiftnet.client.services:getFileStatusRequest

This service sends the Sw:GetFileStatusRequest to SAG over RA and returns the Sw:GetFileStatusResponse received from SAG.

### Input Parameters

| | |
|---|---|
| *SwGetFileStatusRequest* | Document Reference Request to fetch a file from an SnF queue. |

### Output Parameters

| | |
|---|---|
| *SwGetFileStatusResponse* | Document Reference File transfer status response. |
| *error* | String Whether an error occurred. Valid values: `true` and `false`. |
| *errorXMLString* | String Conditional. Error details received from SAG. |

## wm.swiftnet.client.services:initRequest

This service sends the Sw:InitRequest to SAG over RA and returns the Sw:InitResponse received from SAG. This is the initialization primitive exchanged before any other primitives are exchanged.

### Input Parameters

| | |
|---|---|
| *SwInitRequest* | Document Reference Initialization request primitive. |

### Output Parameters

| | |
|---|---|
| *SwInitResponse* | Document Reference Initialization primitive response. |
| *error* | String Whether an error occurred. Valid values: `true` and `false`. |
| *errorXMLString* | String Conditional. Error details received from SAG. |

## wm.swiftnet.client.services:pullSnFRequest

This service sends the Sw:PullSnFRequest to SAG over RA and returns the Sw:PullSnFResponse received from SAG.

### Input Parameters

*SwPullSnFRequest* Document Reference Request to pull a message from the SnF queue.

### Output Parameters

*SwPullSnFResponse* Document Reference Response returned by SWIFTNet for a pull request.

*error* String Whether an error occurred. Valid values: `true` and `false`.

*errorXMLString* String Conditional. Error details received from SAG.

## wm.swiftnet.client.services:sendRequest

This service sends the SwInt:SendRequest to SAG over RA and returns the SwInt:SendResponse received from SAG. This is the asynchronous version of SwInt:ExchangeRequest primitive.

### Input Parameters

*SwIntSendRequest* Document Reference Asynchronous request primitive.

### Output Parameters

*SwIntSendResponse* Document Reference Immediate response received from SAG, without waiting for the actual response from SWIFTNet.

*error* String Whether an error occurred. Valid values: `true` and `false`.

*errorXMLString* String Conditional. Error details received from SAG.

## wm.swiftnet.client.services:sendSynchronousRequest

This service formats the input request primitive into an XML string and then invokes the wm.swiftnet.client.services:swCall service to send the request primitive to SAG over RA. The response XML string received is then formatted into the appropriate response primitive.

### Input Parameters

*requestDocument* Document Request primitive to be sent.

*requestDocNSName* **String** NS record name of request primitive.

*responseDocNSName* **String** NS record name of response primitive.

### Output Parameters

*responseDocument* **Document** Response primitive received.

*error* **String** Whether an error occurred. Valid values: `true` and `false`.

*errorXMLString* **String** Conditional. Error details received from SAG.

## wm.swiftnet.client.services:signEncryptRequest

This service sends the SwSec:SignEncryptRequest to SAG over RA and returns the SwSec:SignEncryptResponse received from SAG.

### Input Parameters

*SwSecSignEncryptRequest* **Document Reference** Request sign and/or encrypt payload.

### Output Parameters

*SwSecSignEncryptResponse* **Document Reference** Conditional. Response received from SAG.

*error* **String** Whether an error occurred. Valid values: `true` and `false`.

*errorXMLString* **String** Conditional. Error details received from SAG.

## wm.swiftnet.client.services:swArguments

This service initializes the client application by invoking the SwArguments() function defined in the SNL libraries. The service takes a String[ ] of arguments as input. The only mandatory parameter to be passed is the *SAGMessagePartner* defined in SAG.

For example:

```
args[0] = "WmSWIFTNetClient"args[1] = "-SagMessagePartner"args[2] = "<message
partner name defined in SAG>"
```

### Input Parameters

*args* **String Array** Initialization arguments to be passed to the SNL libraries.

### Output Parameters

None.

## wm.swiftnet.client.services:swCall

This service invokes the SwCall() function in the SNL libraries to send a request primitive to SAG and returns the response primitive received from SAG.

### Input Parameters

| | |
|---|---|
| *xmlRequest* | **String** Request primitive to be sent to SAG. |

### Output Parameters

| | |
|---|---|
| *xmlResponse* | **String** Response received from SAG. |
| *error* | **String** Whether an error occurred. Valid values: `true` and `false`. |
| *errorXMLString* | **String** Conditional. Error details received from SAG. |

## wm.swiftnet.client.services:termRequest

This service sends the Sw:TermRequest to SAG over RA and returns the Sw:TermResponse received from SAG.

### Input Parameters

| | |
|---|---|
| *SwTermRequest* | **Document Reference** Session termination request to SAG. |

### Output Parameters

| | |
|---|---|
| *SwTermResponse* | **Document Reference** Session termination request to SAG. |
| *error* | **String** Whether an error occurred. Valid values: `true` and `false`. |
| *errorXMLString* | **String** Conditional. Error details received from SAG. |

## wm.swiftnet.client.services:verifyDecryptRequest

This service sends the SwSec:VerifyDecryptRequest to SAG over RA and returns the SwSec:VerifyDecryptResponse received from SAG.

### Input Parameters

| | |
|---|---|
| *SwSecVerifyDecrypt Request* | **Document Reference** Request to verify a signed/encrypted message. |

### Output Parameters

| | |
|---|---|
| *SwSecVerifyDecrypt Response* | **Document Reference** Message decryption response from SAG. |
| *error* | **String** Whether an error occurred. Valid values: `true` and `false`. |
| *errorXMLString* | **String** Conditional. Error details received from SAG. |

## wm.swiftnet.client.services:waitRequest

This service sends the SwInt:WaitRequest to SAG over RA and returns the SwInt:WaitResponse received from SAG. This is the primitive exchanged to retrieve a response asynchronously.

### Input Parameters

| | |
|---|---|
| *SwIntWaitRequest* | **Document Reference** Request to retrieve response asynchronously. |

### Output Parameters

| | |
|---|---|
| *SwIntWaitResponse* | **Document Reference** Asynchronous response received. |
| *error* | **String** Whether an error occurred. Valid values: `true` and `false`. |
| *errorXMLString* | **String** Conditional. Error details received from SAG. |

## wm.swiftnet.client.transport Folder

The services in this folder transfer files from your client application to SAG using the FTA interface.

## wm.swiftnet.client.transport.FTA:generateCompanionFile

This service generates a companion .par file with the specified inputs.

### Input Parameters

| | |
|---|---|
| *companionFileDocument* | **Document** Companion file content as required by the SWIFT documentation. |
| *outputDirectory* | **String** The directory in which the companion .par file is created. |
| *fileName* | **String** The name of the companion .par file to be created with the file. |

### Output Parameters

None.

## wm.swiftnet.client.transport.FTA:scanForReports

This service scans the input directory for the report files and submits them to Trading Networks. This directory must point to the location where SAG will drop all the report files generated for the file transfer.

### Input Parameters

| | |
|---|---|
| *dir* | String The input directory for the report files generated for the files transfer. |

### Output Parameters

None.

## wm.swiftnet.client.transport.FTA:submitToTN

This service submits the file to Trading Networks. This service should be used only for submitting report files and companion files that are in XML format. Before running this service, make sure to import in Trading Networks the document types for report.xml (FTADocTypes.dat), using the import document types feature of Trading Networks.

### Input Parameters

| | |
|---|---|
| *filename* | String The name of the file to submit to Trading Networks. |

### Output Parameters

None.

## wm.swiftnet.client.util Folder

This folder contains various utility services.

## wm.swiftnet.client.util:formatXML

This service formats an XML string by appending the following namespace declarations: “Sw,” “SwInt,” “SwGbl,” and “SwSec.” If these namespaces are not appended to the root tag, the incoming XML response primitives cannot be converted into IData objects in Integration Server.

### Input Parameters

| | |
|---|---|
| *xmlRequest* | String XML string to be formatted with namespaces. |

### Output Parameters

| | |
|---|---|
| *formattedXML* | String XML string with namespaces appended after the root tag. |

## WmSWIFTNetServer Package

This package contains the elements (flow services, Java services, record descriptions, and wrapper services) that support webMethods SWIFTNet server-side functionality. This package contains the following folders:

| Folder | Contains services you use to... |
|---|---|
| wm.swiftnet.server.doc Folder | The NS records that represent the SNL primitives exchanged for FileAct and InterAct operations. |
| wm.swiftnet.server.init Folder | Start and terminate the server process. |
| wm.swiftnet.server.mq Folder | Receive requests from SAG over the MQ transport. |
| wm.swiftnet.server.property Folder | This folder contains services that load properties specified in the *IntegrationServer_directory*\packages\WmSWIFTNetServer\config\snl.cnf file. |
| wm.swiftnet.server.services Folder | Handle incoming requests. |
| wm.swiftnet.server.util Folder | Various utility services. |

## wm.swiftnet.server.doc Folder

This folder contains the NS records that represent the SNL primitives exchanged for FileAct and InterAct operations.

## wm.swiftnet.server.init Folder

This folder contains services that start and terminate the server process.

## wm.swiftnet.server.init:printRemoteErrors

This service logs the standard output and standard error from the server process that is connected to SAG. The errors are logged to the Integration Server console. This service must be used only to trace an error and not used otherwise.

### Input Parameters

None.

### Output Parameters

None.

## wm.swiftnet.server.init:shutdown

This service is registered as a shutdown service for the WmSWIFTNetClient package. It terminates the server process that is connected to SAG.

### Input Parameters

None.

### Output Parameters

None.

## wm.swiftnet.server.init:startup

This service starts a server process that connects to SAG. The server process is registered as the server application for the message partner specified in the *Integration Server_directory*\packages\WmSWIFTNetServer\config\snl.cnf file. The following primitives are exchanged with SAG on startup in this order:

1. Sw:HandleInitRequest
2. SwSec:CreateContextRequest
3. SwSec:CreateContextResponse
4. Sw:SubscribeFileEventRequest
5. Sw:SubscribeFileEventResponse
6. Sw:HandleInitResponse

### Input Parameters

None.

### Output Parameters

None.

## wm.swiftnet.server.mq Folder

The services in this folder receive requests from SAG over the MQ transport.

# wm.swiftnet.server.mq.inbound.getInfoFromNotificationDoc

This service fetches the MQ message body and message ID for the document.

## Input Parameters

| | |
|---|---|
| *docName* | String The name of the document. |

## Output Parameters

| | |
|---|---|
| *msgBodyByteArray* | Object The MQ message body. |
| *msgIdByteArray* | Object The message Id. |

# wm.swiftnet.server.mq.inbound.handleSWIFTRequest

You must configure this service for the notification document triggered for an inbound request. This service retrieves the message body and message ID from the incoming document. It then adds the message body as XML data content part, and the message ID as msgId content part to the TN BizDocEnvelope. Finally, this service submits the document to Trading Networks for further processing.

## Input Parameters

None.

## Output Parameters

None.

# wm.swiftnet.server.mq.trp.respond

Sends the server responses back to SWIFT through the MQ transport.

## Input Parameters

| | |
|---|---|
| *bizdoc* | Document The Trading Networks BizDocEnvelope that contains the SWIFT message. |

## Output Parameters

None.

## wm.swiftnet.server.mq.util.sendToMQ

This service sends the server requests to SAG through the MQ transport.

### Input Parameters

| | |
|---|---|
| *bizdoc* | Document The Trading Networks BizDocEnvelope that contains the SWIFT message. |

### Output Parameters

| | |
|---|---|
| *string* | Byte Array The response content. |

## wm.swiftnet.server.property Folder

This folder contains services that load properties specified in the *Integration Server_directory*\packages\WmSWIFTNetServer\config\snl.cnf file.

## wm.swiftnet.server.property:getCommonProperties

This service retrieves the most commonly used properties from the *Integration Server_directory*\packages\WmSWIFTNetServer\config\snl.cnf file.

### Input Parameters

None.

### Output Parameters

| | |
|---|---|
| *SAGMessage Partner* | String Must correspond to a "Server" type message partner defined in SAG. |
| *server_pki_profile* | String User name of the profile defined in SAG. |
| *server_pki_password* | String Password associated with the user name used to unlock the security information in SAG. |
| *Sign,Decrypt and Authorization* | String Values used for populating SwSec:CreateContextRequest primitive exchanged during server initialization. Valid values: `True` and `False`. |
| *encryptDN* | String Distinguished Name to be used for encryption operations. |
| *cryptoMode* | String Specifies whether encryption operations are performed automatically by SAG/SNL. Valid values: `Automatic` and `Manual`. |

## wm.swiftnet.server.property:getProperty

This service retrieves the value of the specified property from the *Integration Server_directory*\ packages\WmSWIFTNetServer\config\snl.cnf file.

### Input Parameters

*propertyName* String Property value to be retrieved.

### Output Parameters

*value* String Value of the property.

## wm.swiftnet.server.property:listProperties

This service retrieves all the properties specified in the *Integration Server_directory*\packages\ WmSWIFTNetServer\config\snl.cnf file.

### Input Parameters

None.

### Output Parameters

*properties* Document All properties in the snl.cnf file.

## wm.swiftnet.server.property:reloadProperties

This service reloads all the properties specified in the configuration file: *Integration Server_directory*\ packages\WmSWIFTNetServer\config\snl.cnf. This could be useful if more properties are added or existing properties have been changed and the changes need to be reflected in Integration Server immediately.

### Input Parameters

None.

### Output Parameters

*properties* Document All properties reloaded from the snl.cnf file.

# wm.swiftnet.server.property:setProperty

This service sets the property specified in the input. You can set the properties in the snl.cnf file using this service.

### Input Parameters

| | |
|---|---|
| *propertyName* | String Property value to be set. |
| *value* | String Value of the property. |

### Output Parameters

None.

# wm.swiftnet.server.services Folder

The services in this folder handle incoming requests.

# wm.swiftnet.server.services:handleRequest

The SwCallBack function in WmSWIFTNetServer.dll invokes this service when a request is received from SAG. This service recognizes the incoming request primitive as a TN document type and invokes the processing rule specified by the user. The output of the service specified by the user for the processing rule must contain the string variable *xmlResponse* that is send back to SAG as the response to the incoming request.

### Input Parameters

| | |
|---|---|
| *xmlRequest* | String Incoming request primitive. |
| *SwSecUserDN* | String User DN returned by security context created in SAG at startup. |

### Output Parameters

| | |
|---|---|
| *xmlResponse* | String Outgoing response primitive. |

# wm.swiftnet.server.services:swCall

This service invokes the SwCall() function in the SNL libraries to send a request primitive to SAG. The response primitive received from SAG is then output to the pipeline.

### Input Parameters

| | |
|---|---|
| *xmlRequest* | String Request primitive to be sent to SAG. |

*SwSecUserDN* String User DN returned by security context created in SAG at startup.

### Output Parameters

*xmlResponses* String Array Outgoing response primitive.

*error* String Whether an error occurred. Valid values: `true` and `false`.

*errorXMLStrings* String Array Conditional. Error details received from SAG.

## wm.swiftnet.server.util Folder

This folder contains utility services.

## wm.swiftnet.server.util:formatXML

This service formats an XML string by appending the following namespace declarations: “Sw,” “SwInt,” “SwGbl,” and “SwSec. If these namespaces are not appended to the root tag, the incoming XML response primitives cannot be converted into IData objects in Integration Server.

### Input Parameters

*xmlRequest* String Request primitive to be sent to SAG.

### Output Parameters

*formattedXML* String XML string with namespaces appended after the root tag.

## SWIFTNet Server and Client Errors

The following status and error messages may occur during SWIFTNet client or server processing.

| Error Code | Type | Description |
|---|---|---|
| 0 | SWLIB_E_SUCCESS | The function is successfully executed. |
| 1 | SWLIB_E_INTERNAL | An error occurred that does not fit into any of the other codes categories. Normally, this is an internal error. |
| 2 | SWLIB_E_IS_CLIENT | Returned by SwServer if SwCall was called before, indicating the application is a client. |

| Error Code | Type | Description |
|---|---|---|
| 3 | SWLIB_E_CALLED_TWICE | Indicates that a function is called twice. Returned by SwServer or SwArguments. |
| 4 | SWLIB_E_NO_CALLBACKS | Indicates that mandatory callback functions are not registered when SwServer is called. |
| 5 | SWLIB_E_BAD_DLL | Indicates that some mandatory functions are not implemented in the libraries. If the libraries provide only a client implementation, the server functions also return this error code. |
| 6 | SWLIB_E_ACCESS_DLL | A library cannot be loaded, either because the access rights are not correctly set, or because the library does not exist, or because the loaded file is not a library. |
| 7 | SWLIB_E_ACCESS_CFG | The given category refers to one or more configuration files. This error code is returned when at least one configuration file cannot be read, either because the access rights are not correctly set, or because one or more files do not exist. |
| 8 | SWLIB_E_NO_VERSION | The given category has no corresponding libraries. |
| 9 | SWLIB_E_FORMAT_CFG | One or more configuration files do not have the expected format. |
| 10 | SWLIB_E_FORMAT_ARG | Returned only by SwArguments: some arguments do not have the correct format. |
| 11 | SWLIB_E_PREVIOUS_ERROR | The call of the function is rejected because a previous operation failed and returned an error. |
| 12 | SWLIB_E_RMI_ERROR | SWIFT Module internal error. |

## Services and the SNL Request and Response Primitives

The SWIFT Module services make calls to the following SNL request and response primitives that are involved in communication between the client module, the server module, and SWIFTNet:

| | |
|---|---|
| Sw:ExchangeFileRequest | Sw:ExchangeFileResponse |
| Sw:ExchangeSnFRequest ‘ | Sw:ExchangeSnFResponse |
| Sw:FetchFileRequest | Sw:FetchFileResponse |
| Sw:HandleFileEventRequest | Sw:HandleFileEventResponse |

| | |
|---|---|
| Sw:HandleFileRequest | Sw:HandleFileResponse |
| Sw:HandleInitRequest | Sw:HandleInitResponse |
| Sw:HandleSnFRequest | Sw:HandleSnFResponse |
| Sw:InitRequest | Sw:InitResponse |
| Sw:PullSnFRequest | Sw:PullSnFResponse |
| SwSec:CreateContextRequest | SwSec:CreateContextResponse |
| SwSec:DestroyContextRequest | SwSec:DestroyContextResponse |
| Sw:SubscribeFileEventRequest | Sw:SubscribeFileEventResponse |
| Sw:TermRequest | Sw:TermResponse |
| SwInt:ExchangeRequest | SwInt:ExchangeResponse |
| SwInt:HandleRequest | SwInt:HandleResponse |
| SwInt:SendRequest | SwInt:SendResponse |
| SwInt:WaitRequest | SwInt:WaitResponse |

# B XML Parsing Templates for SWIFT FIN Messages

# Overview

SWIFT Module provides XML parsing template files to define the structure of SWIFT FIN messages. Each parsing template describes the message using an XML syntax, and each parsing template defines a unique SWIFT message. SWIFT Module uses the parsing template when it receives a message of that type.

**Important!** XML parsing templates are used only when receiving messages in Integration Server.

To fully define the entire set of SWIFT FIN messages, a parsing template is required for each type of SWIFT message. The parsing templates are installed in the appropriate category and version folder. SWIFT Module reads the parsing template as needed at run time.

The name of each parsing template is based on the definition of the message type it contains. Each message type has a unique ID, which is usually a three-digit number. Typically, the name of a parsing template follows a convention that indicates the MT defined in the parsing template. The following table shows the format used for the names of parsing templates:

| Format of Parsing Template Name | Used for... |
|---|---|
| swiftmtF21.xml | Incoming ACK/NACK messages returned by the SWIFT system. Any service message will follow this file name format. |
| `swiftmt`*nnn*.xml<br>where *nnn* is the unique id for the message type. | All other incoming messages types and all outgoing message. SWIFT Module looks for a specific parsing template file for the specific type of message, for example, swiftmt101.xml. |

The parsing templates that are provided with SWIFT Module are a mixture of messages that conform to the older SWIFT message standard (ISO 1775) and the new standard (ISO 15022). SWIFT Module can support both standards because of the flexible parsing template syntax. As a result, when SWIFT issues an update of their message standards, you can define new parsing templates for new SWIFT message formats or update existing parsing templates for updated SWIFT message formats.

The wm.fin.trp:receiveMessage converts messages that are received from SWIFT into Integration Server document structures. Likewise, the wm.fin.trp:sendMessage service converts Integration Server documents into SWIFT FIN messages, so that the messages can be sent to SWIFT.

## SWIFT Message Data

The following message is a sample of *MT 101*. Users unfamiliar with the SWIFT format should take time to study this data. Alternate blocks and repeating sequences within block 4 have been highlighted for clarity.

```
{1:F01PASOBEB0AXXX0071007172}{2:O1011509010306LRLRXXXX4A00000009622301030616
09N}
{3:{108:MT101 005 OF 007}}{4:
:20:00054
:50H:/12345-67891
WALT DISNEY COMPANY
:30:000228
:
:21:DP951101TRSGB
:32B:USD132546,93
:50L:WALT DISNEY PRODUCTION HOLLYWOOD CA
:57A:TESTGBVT
:59:/0010499
TRISTAN RECORDING STUDIOS
35 SURREY ROAD
BROMLEY, KENT
:71A:OUR
:21:WDC951101RPCUS
:32B:USD377250,
:50L:WALT DISNEY COMPANY LOS ANGELES, CA
:57A:TESTUSVT
:59:/26351-38947
RIVERS PAPER COMPANY
37498 STONE ROAD
SAM RAMON, CA
:71A:OUR
-}{5:{MAC:711DDD87}{CHK:A66AB15C6E3F}{TNG:}}{S:{SAC:}{COP:P}}
```

### Sample SWIFT Message Definition

The following tables provide the definition of block 4 in SWIFT message 101. Blocks 1, 2, 3 and 5 are not shown because they have a fixed definition for all message types.

| Field Name | Mandatory |
|---|---|
| 21R | No |
| 50L | No |
| 50H | No |
| 52A or<br>52C | No |
| 51A | No |
| 30 | Yes |

| Field Name | Mandatory |
|---|---|
| 25 | No |

| Field Name | Mandatory | Notes |
|---|---|---|
| 21 | Yes | |
| 21F | No | |
| 23E | Yes | Field can repeat multiple times. |
| 32B | Yes | |
| 50L | No | |
| 50H | No | |
| 52A<br>or<br>52C | No | |
| 56A<br>or<br>56C<br>or<br>56D | Yes | |
| 57A<br>or<br>57C<br>or<br>57D | No | |
| 59 | Yes | |
| 70 | No | |
| 77B | No | |
| 33B | No | |
| 71A | Yes | |
| 25A | No | |
| 36 | No | |

# Parsing Template Structure

All SWIFT FIN messages are essentially a sequence of fields that are contained within blocks. The message syntax allows for the fact that blocks and fields can be optional, and that blocks can be nested to any level, can repeat, or embed sub-messages. Every SWIFT message consists of one to five blocks as shown in the following table.

| Block ID | Block Name | Mandatory | Description |
|---|---|---|---|
| 1 | Basic Header | Yes | Contains fixed length, untagged fields. |
| 2 | Application Header | No | Contains fixed length, untagged fields. |
| 3 | User Header | No | Contains tagged, delimited fields that are mapped to individual fields. |
| 4 | Text | No | Contains tagged, delimited fields that are mapped to individual fields within a sub structure. This block can contain nested blocks of fields that are mapped into further sub-structures. It also can contain repeating sequences of fields, which are mapped to a sequence of structures. |
| 5 | Trailers | No | Contains delimited, tagged fields that are mapped to individual fields within a sub structure. |

Each of these five basic blocks is enclosed in braces {….} and is identified by a single digit. Although defined as an optional block, in practice block 4 is always present because it contains the actual message text.

Despite these complexities, the parsing templates use just two main elements: *block* and *lineAttribute*.

## Sample Parsing Template

A sample parsing template is illustrated below:

```
<?xml version="1.0"?>
<block id="101" isMandator y = "true" isList ="false">
   <lineAttribute id="1:"   isMandatory="true" extract Hint="BR,{,},T,S"
idHint="FL,0,2" B2BMap="" EAImap="B1" />
   <lineAttribute id="2:"   isMandatory="false" extractHint="BR,{,},T,S"
idHint="FL,0,2" B2BMap="" EAImap="B2" />
<block id="3:" isMandatory="true" isList="false" termString=""
extractHint="BR,{,},T,S" idHint="FL,0,2" EAImap="B3">
     <lineAttribute id="103:"   isMandatory="false" extractHint="BR,{,},T,S"
idHint="FL,0,4" B2BMap="" EAImap="O103" />
     <lineAttribute id="113:"   isMandatory="false" extractHint="BR,{,},T,S"
idHint="FL,0,4" B2BMap="" EAImap="O113" />
     <lineAttribute id="108:"   isMandatory="false" extractHint="BR,{,},T,S"
```

```
idHint="FL,0,4" B2BMap="" EAImap="O108" />
     <lineAttribute id="119:"   isMandatory="false" extractHint="BR,{,},T,S"
idHint="FL,0,4" B2BMap="" EAImap="O119" />
     <lineAttribute id="115:"   isMandatory="false" extractHint="BR,{,},T,S"
idHint="FL,0,4" B2BMap="" EAImap="O115" />
</block>
<block id="4:\r\n"   isMandatory="false" isList="false" termString="\r\n"
extractHint="BR,{,-},T,S" idHint="FL,0,4" EAImap="B4">
     <lineAttribute id=":20:" isMandatory="true" extractHint="DL,:,T,S"
idHint="FL,0,4" B2BMap="" EAImap="M20" />
     <lineAttribute id=":21R:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O21R" />
     <lineAttribute id=":50L:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O50L" />
     <lineAttribute id=":50H:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O50H" />
     <lineAttribute id=":52A:,:52C:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O52A,O52C" />
     <lineAttribute id=":51A:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O51A" />
     <lineAttribute id=":30:" isMandatory="true" extractHint="DL,:,T,S"
idHint="FL,0,4" B2BMap="" EAImap="M30" />
     <lineAttribute id=":25:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,4" B2BMap="" EAImap="O25" />
     <block id="B4B" isMandatory="true" isList="true" termString="\r\n"
EAImap="B4B">
          <lineAttribute id=":21:" isMandatory="true" extractHint="DL,:,T,S"
idHint="FL,0,4" B2BMap="" EAImap="M21" />
          <lineAttribute id=":21F:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O21F" />
          <block id="B423E" isMandatory="false" isList="true"termString="\r\n"
EAImap="B423E">
             <lineAttribute id=":23E:" isMandatory="true"
extractHint="DL,:,T,S" idHint="FL,0,5" B2BMap="" EAImap="O23E"/>
          </block>
          <lineAttribute id=":32B:" isMandatory="true" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="M32B" />
          <lineAttribute id=":50L:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O50L" />
          <lineAttribute id=":50H:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O50H" />
          <lineAttribute id=":52A:,:52C:" isMandatory="false"
extractHint="DL,:,T,S" idHint="FL,0,5" B2BMap="" EAImap="O52A,O52C" />
          <lineAttribute id=":56A:,:56C:,:56D:" isMandatory="false"
extractHint="DL,:,T,S" idHint="FL,0,5" B2BMap="" EAImap="O56A,O56C,O56D" />
          <lineAttribute id=":57A:,:57C:,:57D:" isMandatory="false"
extractHint="DL,:,T,S" idHint="FL,0,5" B2BMap="" EAImap="O57A,O57C,O57D" />
          <lineAttribute id=":59:" isMandatory="true" extractHint="DL,:,T,S"
idHint="FL,0,4" B2BMap="" EAImap="M59" />
          <lineAttribute id=":70:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,4" B2BMap="" EAImap="O70" />
          <lineAttribute id=":77B:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O77B" />
<lineAttribute id=":33B:" isMandatory="false" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="O33B" />
          <lineAttribute id=":71A:" isMandatory="true" extractHint="DL,:,T,S"
idHint="FL,0,5" B2BMap="" EAImap="M71A" />
```

```
<lineAttribute id=":25A:" isMandatory="false" extractHint="DL,:,T,S" 
idHint="FL,0,5" B2BMap="" EAImap="O25A" />
          <lineAttribute id=":36:" isMandatory="false" extractHint="DL,:,T,S" 
idHint="FL,0,4" B2BMap="" EAImap="O36" />
        </block>
     </block>
     <block id="5:"   isMandatory="false" isList="false" termString="\r\n" 
extractHint="BR,{,},T,S" idHint="FL,0,2" EAImap="B5">
          <lineAttribute id="MAC"   isMandatory="true" extractHint="BR,{,},T,S" 
idHint="FL,0,3" B2BMap="" EAImap="MMAC" />
          <lineAttribute id="CHK"   isMandatory="true" extractHint="BR,{,},T,S" 
idHint="FL,0,3" B2BMap="" EAImap="MCHK" />
          <lineAttribute id="TNG"   isMandatory="false" extractHint="BR,{,},T,S" 
idHint="FL,0,3" B2BMap="" EAImap="OTNG" />
          <lineAttribute id="PDE"   isMandatory="false" extractHint="BR,{,},T,S" 
idHint="FL,0,3" B2BMap="" EAImap="OPDE" />
          <lineAttribute id="SYS"   isMandatory="false" extractHint="BR,{,},T,S" 
idHint="FL,0,3" B2BMap="" EAImap="OSYS" />
          <lineAttribute id="PDM"   isMandatory="false" extractHint="BR,{,},T,S" 
idHint="FL,0,3" B2BMap="" EAImap="OPDM" />
          <lineAttribute id="DLM"   isMandatory="false" extractHint="BR,{,},T,S" 
idHint="FL,0,3" B2BMap="" EAImap="ODLM" />
          <lineAttribute id="PAC"   isMandatory="false" extractHint="BR,{,},T,S" 
idHint="FL,0,3" B2BMap="" EAImap="OPAC" />
          <lineAttribute id="MRF"   isMandatory="false" extractHint="BR,{,},T,S" 
idHint="FL,0,3" B2BMap="" EAImap="OMRF" />
     </block>
</block>
```

## Block Syntax of a Parsing Template

The `block` elements in the SWIFT parsing template define the blocks in the SWIFT message. The syntax of a `block` directive is shown below. (Note that optional parameters are shown in square braces [ ].)

```
<block id="id" isMandatory="true|false" isList="true|false" 
[termString="string"] [ EAImap="string"] [[extractHint="hint" idHint="hint"] 
[ loadBlockHint="hint" blockPointer="pointer"]]
```

For the first (outer most) block directive, only the mandatory parameters are supplied. The first block directive identifies the entire message, which translates to the top level Enterprise document type. The syntax of the first block directive always takes the following syntax:

```
<block id="nnn" isMandatory="true" isList="false">
```

where *nnn* is the SWIFT message type number. This number must match the number in the parsing template file name, for example, for *MT 101*, the value of *nnn*must be 101 in the file `swiftmt101.xml`.

For all subsequent blocks, the `block` directive requires optional parameters. The full set of parameters are described below.

| Elements | Values | Description |
|---|---|---|
| id | block id | The block number in the SWIFT message, including delimiting characters. For the first block directive this contains the message type ID. |
| isMandatory | true \| false | false—The block is optional and is not required for the message to be valid.<br>true—The block is mandatory in an incoming message or the message will fail validation. |
| isList | true \| false | false—Only one instance of the block can occur.<br>true—The block can repeat one or more times. |
| termString | \r (carriage return)<br>\n (line feed) | Characters that occur at the end of the block. Block 4 must be terminated with a carriage return, plus line feed. Blocks 1, 2, and 3 are not terminated with a carriage return plus line feed and the value of termString should be an empty string (termString=””). |
| EAImap | document structure name | Represents the name of the structure in the document that the SWIFT message block is mapped to or from. By convention, the EAImap value for blocks is prefixed with “B.” |
| extractHint | Parameter list | The first member must be BR, block is enclosed in braces. The syntactical clue used to identify the beginning and end of blocks.<br>■ Incoming message—extractHint strips block markers from the raw data.<br>■ Outgoing message—extractHint specifies the padding characters to apply to form a syntactically correct message block.<br>**Note:** The “braces” used to enclose blocks can comprise any characters, such as {……*blockdata*…...-} or:16R:TRADE……*blockdata*……:16S:TRADE<br>For a full explanation of this element, see “Hint Processing” on page 305. |

| Elements | Values | Description |
|---|---|---|
| IdHint | Parameter list | The first member must be FL. Tag is fixed length. Or EH tag is derived from the extractHint.<br>Syntactical clue used to extract the tag that identifies the block extracted using the extractHint.<br>■ FL is usually used and strips the first few characters from the remaining raw data.<br>■ EH extracts the first few characters from the block marker stripped by the extract hint. This is used where the block delimiters are themselves a string, such as, 16R:TRADE.<br>For SWIFT Module to identify the block correctly, the text returned by idHint must match the value in the block ID element.<br>For a full explanation of this element, see "Hint Processing" on page 305. |
| loadBlockHint | Parameter list | The first member must be FL, tag is fixed length. Used for embedded messages, such as in n92, n95 and n96. These messages embed block 4 of a previously processed message. The embedded message can be any of the SWIFT message types.<br>loadBlockHint indicates the sub-template that must be embedded in the current parsing template whenever an embedded message is received. It extracts the message type number (100, 101, 521 etc.) from the data returned by a preceding lineAttribute.<br>For a full explanation of this element, see "Hint Processing" on page 305.<br>**Note:** loadBlockHint must be used with blockPointer. If these elements are specified, extractHint and idHint are omitted. |
| blockPointer | 4:\r\n | Used along with loadBlockHint. Specifies that block 4 is to be included from the embedded parsing template. |

## Line Attribute Syntax of a Parsing Template

The lineAttribute elements typically define a single field in the message, although they are also used in the generic parsing template to include an entire block of fields. The syntax of the lineAttribute directive is shown below.

```
<block id="id" isMandatory="true|false"
```

```
EAImap="string" extractHint="hint" idHint="hint"
[ blockLoadField="true"]>
```

**Note:** Note that optional parameters are shown in square braces [ ].

| Elements | Values | Description |
|---|---|---|
| `id` | field id<br>or<br>field id list | The field tag in the SWIFT message, including delimiting characters. Mutually exclusive fields are depicted as a comma separated list (see fields 52, 56 and 57 in the example parsing template above). |
| `isMandatory` | true / false | Specifies if the field represented by the `lineAttribute` must be present in an incoming message.<br>■ `false`—The *lineAttribute* is optional and is not needed for the message to be considered valid.<br>■ `true`—The *lineAttribute* must be present or the message will fail validation.<br>In an outgoing message, the corresponding field in the Enterprise document (specified by the EAImap element) must be populated, or the message will fail validation. |
| `EAImap` | document field name<br>or<br>document field name list | The name of the field in the Enterprise document that the `lineAttribute` is mapped to or from. By convention, the EAImap value for `lineAttributes` are prefixed with "O" for optional fields or "M" for mandatory fields.<br>Mutually exclusive fields are depicted as a comma separated list that must match the field ID list. |
| `extractHint` | Parameter list | The first member must be BR; field is enclosed by braces. DL—Field is delimited. CK—Field contains the entire block data in a single chunk.<br>**Note:** CK is used for the generic parsing template only.<br>Syntactical clue used to identify the beginning and end of fields. For an incoming message, `extractHint` is used to identify the end of the field data and to strip any braces from the raw data. For a full explanation of this element, see "Hint Processing" on page 305. |

| Elements | Values | Description |
|---|---|---|
| idHint | Parameter list | The first member of which must be FL, ID is fixed length, or CK, ID will not be extracted as the field contains the entire block in a single chunk.<br>Syntactical clue used to identify and strip the ID of the field extracted using the extractHint. The idHint is applied to the raw data line and typically uses a fixed length parameter to return the first few characters of the data.<br>For SWIFT Module to identify the field correctly, the text returned by idHint using FL must match the value in the lineAttribute ID element. For a full explanation of this element, see "Hint Processing" on page 305. |
| blockLoadField | true (if specified) | Optional element that causes the lineAttribute to be referenced by a subsequent loadBlockHint directive. blockLoadField must precede loadBlockHint in the parsing template. |

## Hint Processing

ExtractHint, idHint, and loadBlockHint are used in block and lineAttribute directives. The full syntax is provided in the following sections.

- Braced fields—Used where the data is enclosed in one or more bracing characters BR, <open brace characters>, <close brace characters>, <tag flag>, <tag position>.
- Delimited fields—Used where fields are delimited by a single character DL, <delimiting character>, <tag flag>, <tag position>
- Chunk data—Used where data is treated as a single chunk, without further parsing CK, <tag flag>, <tag position> or extractHint CK[, <from position>, <to position>] for idHint.
- Fixed length—Used where data occupies a fixed number of characters FL, <from position>, <to position>.
- Extract hint—Used to extract a fixed number of characters from the extract hint EH, <from position>, <to position>.

The remaining parameters are provided in the following table:

| Parameter | Values | Description |
|---|---|---|
| open brace character | any sequence of characters | The character(s) used to identify the beginning of the block or field |
| close brace character | any sequence of characters | The character(s) used to identify the end of the block or field |

| Parameter | Values | Description |
|---|---|---|
| `delimiting character` | any character | A single character that separates fields |
| `from position` | numeric character | Starting character position used to extract fixed length data from the raw data after the block characters have been stripped off. First character is position 0. |
| `to position` | numeric character | Last character position plus one used to extract fixed length data. |
| `tag flag` | T or N | `T` (tagged)—Field tag is included with the data.<br>`N` (not tagged)—Data does not include the tag field |
| `tag position` | S or E | `S` (start)—Data tag precedes the data.<br>`E` (end)—Data tag follows the data. |

## Miscellaneous Notes

The following notes should be read and understood before attempting to maintain webMethods SWIFT Module parsing templates.

- While SWIFT defines block 1 as the only mandatory block, in general the parsing templates define blocks 1 through 4 as mandatory.
- Individual optional repeating fields must be defined as a mandatory field within an optional block.
- When `loadBlockHint` / `blockPointer` is used, `isMandatory` must be true and `isList` must be false.
- If `extractHint` uses chunk parameter (CK), `idHint` must also be CK.
- `loadBlockField` and associated `loadBlockHint` / `blockPointer` can only occur once in the parsing template.
- Block 3 must include field 108. This is required to correctly process ACK/NACK messages received from the SWIFT network.

# C Administering webMethods SWIFT Module in a Cluster

# What Is webMethods Integration Server Clustering?

Clustering is an advanced feature of the webMethods product suite that substantially extends the reliability, availability, and scalability of webMethods Integration Server.

Clustering accomplishes this by providing the infrastructure and tools to deploy multiple Integration Servers as if they were a single virtual server and to deliver applications that leverage that architecture.

With clustering, you get the following benefits:

- Scalability: Without clustering, only vertical scalability is possible. That is, increased capacity requirements can only be met by deploying on larger, more powerful machines, typically housing multiple CPUs. Clustering provides horizontal scalability, which allows virtually limitless expansion of capacity by simply adding more machines of the same or similar capacity.
- Availability: Without clustering - even with expensive Fault-Tolerant systems - a failure of the system (hardware, java runtime, or software) may result in unacceptable downtime. Clustering provides virtually uninterrupted availability by deploying applications on multiple Integration Servers; in the worst case, a server failure produces degraded but not disrupted service.
- Reliability: Unlike a server farm (an independent set of servers), clustering provides the reliability required for mission-critical applications. Distributed applications must address network, hardware, and software errors that might produce duplicate (or failed) transactions. Clustering makes it possible to deliver “exactly once” execution as well as checkpoint/restart functionality for critical operations.

For details on Integration Server clustering, see the Integration Server clustering guide for your release. See “About this Guide” for specific document titles.

# SWIFT Module in a Clustered Environment

## Clustering Requirements for Each Integration Server in a Cluster

The requirements of each Integration Server in a given cluster are given below:

- All Integration Servers in a cluster must be of the same version.
- All SWIFT Module instances in a cluster must be of the same version.
- All the SWIFT Module packages on one Integration Server must be replicated to all other Integration Servers in the cluster.
- Each SWIFT Module service must appear on all servers in the cluster so that any Integration Server in the cluster can handle the request identically.

If you allow different Integration Servers to contain different services, you will not derive the full benefits of clustering. For example, if a client requests a service that resides on only one server, and that server is unavailable, the request cannot be successfully redirected to another server.

## Clustering Requirements When Installing SWIFT Module Packages

For each Integration Server in the cluster, use the standard SWIFT Module installation procedure for each machine, as described in Chapter 2, "Installing webMethods SWIFT Module".

You must installSWIFT Module on each host in the cluster. Each installation must be identical.

## Configuring SWIFT Module in a Clustered Environment

When you configure SWIFT Module - that is, when you use it to create packages of generated services - you must ensure that each Integration Server in the cluster contains an identical set of packages. You can create custom packages of generated services of SWIFT Module on one host and use package replication to publish custom packages to each of the other hosts. For information on package replication, see the Integration Server administration guide for your release. See "About this Guide" for specific document titles.

Note: The following sections assume that you have already configured the Integration Server cluster.

### Replicating Packages and Configuration Information to Integration Servers

Each Integration Server in the cluster should contain an identical set of packages that you define using SWIFT Module. To ensure consistency, make sure that you create all packages on one server and replicate these packages to the other servers. If you allow different servers to contain different packages, you will not derive the full benefits of clustering. For example, if a Trading Networks processing rule requests a service that resides in only one server, and that server is unavailable, the request cannot be redirected to another server.

### SWIFT Module Configuration Information

SWIFT Module stores configuration information updated in the SWIFTNet Client Configuration and SWIFTNet Server Configuration screens as configuration files within packages, and the imported BIC, BICPlusIBAN, IS and SR lists are stored in the database. The Trading Networks-related configuration information is stored in the database (JDBC pools).

The SWIFTNet Client Configuration and SWIFTNet Server Configuration information includes the following items:

- SWIFTNet Client Environment Information
- SWIFTNet Client SAG Connection Information
- SWIFTNet Remote Process Connection Configuration
- SWIFTNet Server Environment Information
- SWIFTNet Server SAG Connection Properties

The configuration information is visible only to the server on which SWIFT Module resides; it does not share a common storage facility. Therefore, when using SWIFT Module in a clustered environment, you need to replicate the SWIFTNet Client SAG Connection Information and SWIFTNet Server SAG Connection Properties information in the SWIFTNet Client Configuration and SWIFTNet Server Configuration screens across all the nodes in the cluster.

## Trading Networks Configuration Information

The configuration information created to use SWIFT Module with Trading Networks includes the following items:

- Document attributes and type definitions
- Processing rules
- Trading Partner Agreements

The following information is imported into the database from the Import BIC List, Import IS List, Import SR List and Import BICPlusIBAN List screens:

- BIC List
- BICPlusIBAN List
- IBAN Structure (IS) List
- SEPA Routing (SR) List

When using SWIFT Module in a clustered environment, the Trading Networks-related configuration information and the imported lists are in the database, which is common for all the clustered nodes and should not be replicated.

Note: Ensure that the document attributes, document type definitions, processing rules, Trading Partner Agreements, and imported list files are available in the Integration Server where the configuration information is replicated.

Note: XMLv2 notification reconciliation: The notifications sent by SWIFT are automatically reconciled to the original document, even if the document is sent from a node other than the node receiving the notification. Because the data for reconciling the document exists in the Trading Networks information in the database (JDBC pools) that is shared in the cluster, notification data does not need to be replicated.

To replicate the configuration

1. Ensure that all clustered Integration Server nodes point to the same JDBC pools. For information on how to define JDBC connection pools, see the webMethods installation guide for your release. See "About this Guide" for specific document titles.
2. Replicate all custom flow services and packages by using the copy and send mechanism from the Integration Server Administrator or Designer. For information about replicating packages, see the chapter on managing packages in the Integration Server administration guide for your release. See "About this Guide" for specific document titles.
3. Replicate the SWIFTNet Client SAG Connection Information and SWIFTNet Server SAG Connection Properties configurations as specified on the SWIFTNet Client Configuration screen and the SWIFTNet Server Configuration screen. For more information about these screens, see Chapter 16, "Configuration Steps for InterAct and FileAct Messaging Services over SAG MQHA".

# Clustering Implementation Considerations

There is no specific SWIFT Module-related implementation when using SWIFT Module in a clustered environment, other than the transport considerations described in this section.

**Important!** The SWIFTNet component of SWIFT Module 7.1 SP1 does not support clustering when you use RAHA as the transport for the message exchange.

## AFT Transport

When you use the AFT transport to send and receive SWIFT messages, you must set up the AFT environment as described in "Using AFT to Communicate with SWIFT" on page 116 across the nodes. You must specify AFT as the transport used for the particular message exchange in the TPA. Because the Trading Networks information is available in the database (pointing to the same JDBC pool across the nodes in the cluster), the information does not need to be replicated.

## CASmf Transport

When you use the CASmf transport to send and receive SWIFT messages, you must set up the CASmf environment as described in "Using the CASmf Services to Communicate with SWIFT" on page 112 across the nodes. You must specify CASmf as the transport used for the particular message exchange in the TPA. Because the Trading Networks information is available in the database (pointing to the same JDBC pool across the nodes in the cluster), the information does not need to be replicated.

## MQHA Transport

When you use the MQHA transport to send and receive SWIFT messages, you must ensure that you:

1. Install the webMethods WebSphere MQ Adapter with the latest fix on all the nodes in the cluster.
2. Replicate the packages containing the WebSphere MQ Adapter connections, listeners, and listener notifications across all nodes in the cluster.
3. Enable the replicated WebSphere MQ Adapter connections, listeners, and listener notifications across all nodes in the cluster.

You must set MQHA as the transport used for the particular message exchange in the TPA. Because the Trading Networks information is available in the database (pointing to the same JDBC pool across the nodes in the cluster), the information does not need to be replicated.

**Note:** The Server and Client application contexts created as part of the SWIFTNet component message exchange over the MQHA transport are stored in shared cache and do not need to be replicated. For more information on how to initialize the client and server application contexts, see "Step 3: Initialization and Request-Time Operations for Your Client or Server Application" on page 172.

For more information about the MQHA setup, see "Using WebSphere MQ Adapter to Communicate with SWIFT" on page 110.

# D Examples of Data PDU Content of Documents

# Data PDU Content of Different Types of Notifications

The following are examples of the Data PDU content of different types of notifications that SWIFT Alliance Access sends to webMethods SWIFT Module.

## Data PDU Content of a Delivery Notification Example

The following is an example of the Data PDU content of a Delivery Notification.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Saa:DataPDU xmlns:Saa="urn:swift:saa:xs d:saa.2.0"
xmlns:Sw="urn:swift:snl:ns.Sw"
xmlns:SwInt="urn:swift:snl:ns.SwInt"xmlns:SwGbl="urn:swift:snl:ns.SwGbl"
xmlns:SwSec="urn:swift:snl:ns.SwSec">
<Saa:Revision>
2.0.1</Saa:Revision>
<Saa:Header>
<Saa:DeliveryNotification>
<Saa:ReconciliationInfo>
090624PTSAUSA0AXXX0077000422</Saa:ReconciliationInfo>

<Saa:ReceiverDeliveryStatus>
RcvDelivered</Saa:ReceiverDeliveryStatus>
<Saa:MessageIdentifier>
fin.011</Saa:MessageIdentifier>
<Saa:Receiver>
<Saa:BIC12>
PTSAUSA0AXXX</Saa:BIC12>
<Saa:FullName>
<Saa:X1>

PTSAUSA0XXX</Saa:X1>
</Saa:FullName>
</Saa:Receiver>
<Saa:InterfaceInfo>
<Saa:MessageCreator>
FINInterface</Saa:MessageCreator>
<Saa:MessageContext>
Original</Saa:MessageContext>
<Saa:MessageNature>

Network</Saa:MessageNature>
</Saa:InterfaceInfo>
<Saa:NetworkInfo>

<Saa:Priority>
System</Saa:Priority>
<Saa:IsPossibleDuplicate>
true</Saa:IsPossibleDuplicate>
<Saa:DuplicateHistory>
<Saa:PDM>

{PDM:1102090624PTSAUSA0AXXX0078000885}</Saa:PDM>
</Saa:DuplicateHistory>
<Saa:Service>
```

```
swift.fin</Saa:Service>
<Saa:Network>
FIN</Saa:Network>
<Saa:SessionNr>
0079</Saa:SessionNr>
<Saa:SeqNr>

000888</Saa:SeqNr>
<Saa:FINNetworkInfo>
<Saa:MessageSyntaxVersion>
0805</Saa:MessageSyntaxVersion>
<Saa:CorrespondentInputReference>
090624DYDYXXXXHXXX0000836861</Saa:CorrespondentInputReference>

<Saa:CorrespondentInputTime>
20090624100200</Saa:CorrespondentInputTime>
<Saa:LocalOutputTime>
20090624111400</Saa:LocalOutputTime>
<Saa:SystemOriginated>
{SYS:}</Saa:SystemOriginated>
</Saa:FINNetworkInfo>
</Saa:NetworkInfo>
<Saa:SecurityInfo>
<Saa:FINSecurityInfo>
<Saa:ChecksumResult>

Success</Saa:ChecksumResult>
<Saa:ChecksumValue>
A5D4C6F14E1E</Saa:ChecksumValue>
</Saa:FINSecurityInfo>
</Saa:SecurityInfo>

</Saa:DeliveryNotification>
</Saa:Header>
<Saa:Body>

ezE3NToxMDQ5fXsxMDY6MDkwNjI0UFRTQVVTQTBBWFhYMDA3NzAwMDQyMn17MTA4Ok1UOTEwNTg2MzI4
fXsxNzU6MTA1OX17MTA3OjA5MDYyNFBUU0FVU0EwQVhYWDAwNzgwMDA4ODJ9</Saa:Body>
</Saa:DataPDU>
```

## Data PDU Content of a Delivery Report Example

The following is an example of the Data PDU content of a Delivery Report.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Saa:DataPDU xmlns:Saa="urn:swift:saa:xs d:saa.2.0"
xmlns:Sw="urn:swift:snl:ns.Sw"
xmlns:SwInt="urn:swift:snl:ns.SwInt"xmlns:SwGbl="urn:swift:snl:ns.SwGbl"
xmlns:SwSec="urn:swift:snl:ns.SwSec">
<Saa:Header>
<Saa:DeliveryReport>
<Saa:SenderReference>
MT210244895</Saa:SenderReference>
<Saa:ReceiverDeliveryStatus>
RcvDelivered</Saa:ReceiverDeliveryStatus>
```

```
<Saa:OriginalInstanceAddressee>
<Saa:X1>
PTSAUSA0XXX</Saa:X1>
</Saa:OriginalInstanceAddressee>
<Saa:ReportingApplication>
TrafficReconciliation</Saa:ReportingApplication>
<Saa:NetworkInfo>
<Saa:Priority>

Normal</Saa:Priority>
<Saa:IsPossibleDuplicate>
false</Saa:IsPossibleDuplicate>
<Saa:IsNotificationRequested>
true</Saa:IsNotificationRequested>
<Saa:Service>
swift.fin</Saa:Service>
<Saa:Network>

FIN</Saa:Network>
<Saa:SessionNr>
0077</Saa:SessionNr>
<Saa:SeqNr>
000419</Saa:SeqNr>

<Saa:FINNetworkInfo>
<Saa:MessageSyntaxVersion>
0805</Saa:MessageSyntaxVersion>
</Saa:FINNetworkInfo>
</Saa:NetworkInfo>

<Saa:Interventions>
<Saa:Intervention>
<Saa:IntvCategory>
DeliveryReport</Saa:IntvCategory>
<Saa:CreationTime>
20090624055941</Saa:CreationTime>
<Saa:OperatorOrigin>
SYSTEM</Saa:OperatorOrigin>
<Saa:Contents>
{175:1049}{106:090624PTSAUSA0AXXX0077000419}{108:MT210244895}{175:1049}{107:0906
24PTSAUSA0AXXX0077000872}</Saa:Contents>
</Saa:Intervention>
</Saa:Interventions>
<Saa:IsRelatedInstanceOriginal>

true</Saa:IsRelatedInstanceOriginal>
<Saa:MessageCreator>
ApplicationInterface</Saa:MessageCreator>
<Saa:IsMessageModified>
false</Saa:IsMessageModified>
<Saa:MessageFields>

HeaderAndBody</Saa:MessageFields>
<Saa:Message>
<Saa:SenderReference>
MT210244895</Saa:SenderReference>
<Saa:MessageIdentifier>
```

```
fin.210</Saa:MessageIdentifier>
<Saa:Format>

MT</Saa:Format>
<Saa:SubFormat>
Input</Saa:SubFormat>
<Saa:Sender>
<Saa:BIC12>
PTSAUSA0AXXX</Saa:BIC12>
<Saa:FullName>
<Saa:X1>
PTSAUSA0XXX</Saa:X1>
</Saa:FullName>
</Saa:Sender>
<Saa:Receiver>
<Saa:BIC12>
PTSAUSA0XXXX</Saa:BIC12>

<Saa:FullName>
<Saa:X1>
PTSAUSA0XXX</Saa:X1>
</Saa:FullName>
</Saa:Receiver>
<Saa:InterfaceInfo>
<Saa:UserReference>
MT210244895</Saa:UserReference>
<Saa:MessageCreator>

ApplicationInterface</Saa:MessageCreator>
<Saa:MessageContext>
Report</Saa:MessageContext>
<Saa:MessageNature>
Financial</Saa:MessageNature>
</Saa:InterfaceInfo>
<Saa:NetworkInfo>
<Saa:Priority>

Normal</Saa:Priority>

<Saa:IsPossibleDuplicate>
false</Saa:IsPossibleDuplicate>
<Saa:IsNotificationRequested>
true</Saa:IsNotificationRequested>
<Saa:Service>
swift.fin</Saa:Service>
<Saa:Network>

FIN</Saa:Network>
<Saa:SessionNr>
0077</Saa:SessionNr>
<Saa:SeqNr>
000419</Saa:SeqNr>
<Saa:FINNetworkInfo>
<Saa:MessageSyntaxVersion>
0805</Saa:MessageSyntaxVersion>
</Saa:FINNetworkInfo>
</Saa:NetworkInfo>
```

```
<Saa:SecurityInfo>
<Saa:FINSecurityInfo>
<Saa:ChecksumResult>
Success</Saa:ChecksumResult>
<Saa:ChecksumValue>
6F4ACBD3AA04</Saa:ChecksumValue>
<Saa:MACResult>
Success</Saa:MACResult>
<Saa:MACValue>

00000000</Saa:MACValue>
<Saa:MACSignatureValue>
<SwSec:Signature>
<SwSec:SignedInfo>
<Sw:Reference>
<Sw:DigestValue>
3KgCRof2mgp47iXXIrxAuzFE/thjoETNXUUFtv7PG4o=</Sw:DigestValue>


</Sw:Reference>

</SwSec:SignedInfo>
<SwSec:SignatureValue>
PEMF@Proc-Type: 4,MIC-ONLY
Content-Domain: RFC822
EntrustFile-Version: 2.0
Originator-DN: cn=finuser,o=ptsausaa,o=swift
Orig-SN: 1238170352
MIC-Info: SHA256, RSA,
  nxNjzFQJeeMuk4vcXq4qi9/ZGcH01yZ94N4jzCKqTlWZYF5sWqf5b8w88KSKw5Vrt52ABEvR8/79LC
  ASarCcFZQcv4GOrf9BRu6AjdnUgVnxdbPhJtR+Pfj+TP5Twa8eS82vwbNFK9T7787mrnalQNUih2rA
  Lz3GmA7bcd5N7I2hs2eA35ol0KQaRg/8a+9hI9vd7meeLQVTSQBrLC41HMp+4Gb8kiyaafONxMNB2O
  kGY7bZda1PlmObPYyvrvKRM1xXus6wn2d++hWP3d4CJ3/26FRkWRqK6qKissgyA96AoXSNmqdnzJWy
  5jwnx/ry2kcYHiLBCyJ7gWa2HPZhyg==
</SwSec:SignatureValue>
<SwSec:KeyInfo>
<SwSec:SignDN>
cn=finuser,o=ptsausaa,o=swift</SwSec:SignDN>
<SwSec:CertPolicyId>
</SwSec:CertPolicyId>
</SwSec:KeyInfo>
<SwSec:Manifest>
<Sw:Reference>
<Sw:DigestRef>

M</Sw:DigestRef>
<Sw:DigestValue>
M7WI3Vo173HohEQ5SVRd3RS1V+OQapQFfK+DPLkg3mO=</Sw:DigestValue>
</Sw:Reference>
</SwSec:Manifest>
</SwSec:Signature>
</Saa:MACSignatureValue>
</Saa:FINSecurityInfo>
```

```
</Saa:SecurityInfo>
</Saa:Message>
</Saa:DeliveryReport>
</Saa:Header>
<Saa:Body>
DQo6MjA6MDA0MzkNCjozMDowMDAxMDMNCjoyMToxMjM0NTYvREVWDQo6MzJCOlVTRDEwNSwNCjo1MEM6
VENTRkZSUFA=</Saa:Body>
</Saa:DataPDU>
```

## Data PDU Content of a History Report Example

The following is an example of the Data PDU content of a History Report.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Saa:DataPDU xmlns:Saa="urn:swift:saa:xsd:saa.2.0"xmlns:Sw="urn:swift:snl:ns.Sw"
xmlns:S wInt="urn:swift:snl:ns.SwInt" xmlns:SwGbl="urn:swift:snl:ns.SwGbl"
xmlns:SwSec="urn:swift:snl:ns.SwSec">
<Saa:Revision>
2.0.1</Saa:Revision>
<Saa:Header>
<Saa:HistoryReport>
<Saa:SenderReference>
IPTSAUSA0XXX399TRNMSG1000$0906241473</Saa:SenderReference>


<Saa:OriginalInstanceAddressee>
<Saa:X1>
PTSAUSA0XXX</Saa:X1>
</Saa:OriginalInstanceAddressee>
<Saa:ReportingApplication>
ApplicationInterface</Saa:ReportingApplication>
<Saa:Interventions>

<Saa:Intervention>
<Saa:IntvCategory>
Routing</Saa:IntvCategory>
<Saa:CreationTime>
20090624065518</Saa:CreationTime>
<Saa:OperatorOrigin>
SYSTEM</Saa:OperatorOrigin>
<Saa:Text>
Routed from rp
[_AI_from_APPLI] to rp [_SI_to_SWIFT]; 1 instance(s) created at [FromSAAToMQ]
respectively;On Processing by Function AI_from_APPLI with result
Success;(Rule:USER,200)</Saa:Text>
</Saa:Intervention>

</Saa:Interventions>
<Saa:IsRelatedInstanceOriginal>
true</Saa:IsRelatedInstanceOriginal>
<Saa:MessageCreator>
ApplicationInterface</Saa:MessageCreator>
<Saa:IsMessageModified>
false</Saa:IsMessageModified\
>
<Saa:MessageFields>
```

```
HeaderAndBody</Saa:MessageFields>
<Saa:Message>
<Saa:SenderReference>
IPTSAUSA0XXX399TRNMSG1000$0906241473</Saa:SenderReference>
<Saa:MessageIdentifier>


fin.399</Saa:MessageIdentifier\
>
<Saa:Format>
MT</Saa:Format>
<Saa:SubFormat>
Input</Saa:SubFormat>
<Saa:Sender>
<Saa:BIC12>
PTSAUSA0AXXX</Saa:BIC12>
<Saa:FullName>
<Saa:X1>
PTSAUSA0XXX</Saa:X1>
</Saa:FullName>
</Saa:Sender>
<Saa:Receiver>

<Saa:BIC12>
PTSAUSA0XXXX</Saa:BIC12>
<Saa:FullName>
<Saa:X1>
PTSAUSA0XXX</Saa:X1>
</Saa:FullName>
</Saa:Receiver>
<Saa:InterfaceInfo>
<Saa:UserReference>
MT1990035656006</Saa:UserReference>
<Saa:MessageCreator\
>
ApplicationInterface</Saa:MessageCreator>
<Saa:MessageContext>
Report</Saa:MessageContext>
<Saa:MessageNature>
Financial</Saa:MessageNature>
</Saa:InterfaceInfo>
<Saa:NetworkInfo>
<Saa:Priority>


Urgent</Saa:Priority>
<Saa:IsPossibleDuplicate>
false</Saa:IsPossibleDuplicate>
<Saa:IsNotificationRequested>
true</Saa:IsNotificationRequested>
<Saa:Service>
swift.fin</Saa:Service>
<Saa:FINNetworkInfo>

<Saa:MessageSyntaxVersion>
0805</Saa:MessageSyntaxVersion>
</Saa:FINNetworkInfo>
```

```
</Saa:NetworkInfo>
</Saa:Message>
</Saa:HistoryReport>
</Saa:Header>
<Saa:Body>

DQo6MjA6VFJOIE1TRzEwMDANCjo3OTpCTEFESU5HUyBUTyBCRSBJU1NVRUQgTk9UIExBVEVSIFRIQU4=
</Saa:Body>
</Saa:DataPDU>
```

## Data PDU Content of a Transmission Report Example

The following is an example of the Data PDU content of a Transmission Report.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Saa:DataPDU xmlns:Saa="urn:swift:saa:xs d:saa.2.0"
xmlns:Sw="urn:swift:snl:ns.Sw"
xmlns:SwInt="urn:swift:snl:ns.SwInt" xmlns:SwGbl="urn:swift:snl:ns.SwGbl"
xmlns:SwSec="urn:swift:snl:ns.SwSec">
<Saa:Header>
<Saa:TransmissionReport>
<Saa:SenderReference>
IXXXXXXXXXXX05$0906251484</Saa:SenderReference>
<Saa:NetworkDeliveryStatus>

NetworkAcked</Saa:NetworkDeliveryStatus>
<Saa:OriginalInstanceAddressee>
<Saa:X1>
XXXXXXXXXXX</Saa:X1>
</Saa:OriginalInstanceAddressee>
<Saa:ReportingApplication>
FINInterface</Saa:ReportingApplication>

<Saa:NetworkInfo>
<Saa:Priority>
System</Saa:Priority>
<Saa:IsPossibleDuplicate>
false</Saa:IsPossibleDuplicate>
<Saa:IsNotificationRequested>
false</Saa:IsNotificationRequested>
<Saa:Service>

swift.fin</Saa:Service>
<Saa:Network>
FIN</Saa:Network>
<Saa:SessionNr>
0084</Saa:SessionNr>
<Saa:SeqNr>
000429</Saa:SeqNr>

<Saa:FINNetworkInfo>
<Saa:MessageSyntaxVersion>
0805</Saa:MessageSyntaxVersion>

</Saa:FINNetworkInfo>
</Saa:NetworkInfo>
```

```
<Saa:Interventions>
<Saa:Intervention>
<Saa:IntvCategory>
TransmissionReport</Saa:IntvCategory>
<Saa:CreationTime>
20090625040648</Saa:CreationTime>

<Saa:OperatorOrigin>
SYSTEM</Saa:OperatorOrigin>
<Saa:Contents>
{1:F25PTSAUSA0AXXX0084000429}{4:{331:00840906250908090625091000000000100000000004
290004290008980008 97}}</Saa:Contents>
</Saa:Intervention>
</Saa:Interventions>
<Saa:IsRelatedInstanceOriginal>
true</Saa:IsRelatedInstanceOriginal>
<Saa:MessageCreator>
FINInterface</Saa:MessageCreator>
<Saa:IsMessageModified>
false</Saa:IsMessageModified>

<Saa:MessageFields>
HeaderAndBody</Saa:MessageFields>
<Saa:Message>
<Saa:SenderReference>
IXXXXXXXXXXX05$0906251484</Saa:SenderReference>
<Saa:MessageIdentifier>
fin.05</Saa:MessageIdentifier>
<Saa:Format>

MT</Saa:Format>
<Saa:SubFormat>
Input</Saa:SubFormat>
<Saa:Sender>
<Saa:BIC12>
PTSAUSA0AXXX</Saa:BIC12>
<Saa:FullName>
<Saa:X1>
PTSAUSA0XXX</Saa:X1>
</Saa:FullName>
</Saa:Sender>
<Saa:Receiver>
<Saa:BIC12>

XXXXXXXXXXX</Saa:X1>
<Saa:FullName>
<Saa:X1>
</Saa:FullName>
</Saa:Receiver>
<Saa:InterfaceInfo>
<Saa:MessageCreator>
FINInterface</Saa:MessageCreator>
<Saa:MessageContext>

Report</Saa:MessageContext>
<Saa:MessageNature>
Service</Saa:MessageNature>
```

```
</Saa:InterfaceInfo>
<Saa:NetworkInfo>
<Saa:Priority>
System</Saa:Priority>
<Saa:IsPossibleDuplicate>

false</Saa:IsPossibleDuplicate>


<Saa:IsNotificationRequested>
false</Saa:IsNotificationRequested>
<Saa:Service>
swift.fin</Saa:Service>
<Saa:Network>
FIN</Saa:Network>
<Saa:SessionNr>
0084</Saa:SessionNr>

<Saa:SeqNr>
000429</Saa:SeqNr>
<Saa:FINNetworkInfo>
<Saa:MessageSyntaxVersion>
0805</Saa:MessageSyntaxVersion>
</Saa:FINNetworkInfo>
</Saa:NetworkInfo>
</Saa:Message>
</Saa:TransmissionReport>
</Saa:Header>

</Saa:DataPDU>
```

# MT/MX Message Data PDU Content

The following are examples of the Data PDU content of the MT and MX messages that webMethods SWIFT Module exchanges with the SWIFT Network over SWIFT Alliance Access.

## MT Message Data PDU Content Example

The following is an example of the Data PDU content of a fin.535 (MT message type) message.

```
<?xml version="1.0"?>
<ns:DataPDU
       xmlns:ns="urn:swift:saa:xsd:saa.2.0">
   <ns:Header>
     <ns:Message>
       <ns:SenderReference>MT535946242</ns:SenderReference>
       <ns:MessageIdentifier>fin.535</ns:MessageIdentifier>
       <ns:Format>MT</ns:Format>
       <ns:Sender>
         <ns:BIC12>PTSAUSA0AXXX</ns:BIC12>
         <ns:FullName>
           <ns:X1>PTSAUSA0XXX</ns:X1>
         </ns:FullName>
       </ns:Sender>
       <ns:Receiver>
          <ns:BIC12>PTSAUSA0XXXX</ns:BIC12>
          <ns:FullName>
            <ns:X1>PTSAUSA0XXX</ns:X1>
          </ns:FullName>
            </ns:Receiver>
            <ns:InterfaceInfo>
              <ns:UserReference>MT535946242</ns:UserReference>
            </ns:InterfaceInfo>
            <ns:NetworkInfo>
              <ns:IsNotificationRequested>true</ns:IsNotificationRequested>
            </ns:NetworkInfo>
       </ns:Message>
   </ns:Header>


 <ns:Body>
DQo6MTZSOkdFTkwNCjoyOEU6MTIzNDUvTO5MWQ0KOjEzQTo6U1RBVC8vQTJDDQo6MjBDOjpTRU1FLy8w
MTM4OA0KOjIzRzpORVdNLONPRFUNCjo5OEE6OlBSRVAvLzE5OTkxMjMxDQo6OThBOjpTVEFULy8xOTk5
MTIzMQ0KOjIyRjo6U0ZSRS9BMkM0RTZHOC9BREh
PDQo6MjJGOjpDTORFLOEyQzRFNkc4LONPTVANCjoyMkY6OlNUVFkvQTJDNEU2RzgvQUNDVA0KOjIyRjo
6U1RCQS9BMkM0RTZHOC9CTO9LDQo6MTZSOkxJTksNCjoxM0E6OkxJTksvLzUwMw0KOjIwQzo6UkVMQS8
veA0KOjE2UzpMSU5LDQo6MTZSOkxJTksNCjoxM0
E6OkxJTksvLzUwMw0KOjIwQzo6UFJFVi8veA0KOjE2UzpMSU5LDQo6OTdBOjpTQUZFLy94DQo6MTdCOj
pBQ1RJLy9ZDQo6MTdCOjpBVURULy9ZDQo6MTdCOjpDTO5TLy9ZDQo6MTZTOkdFTkwNCjoxNlI6U1VCU0
FGRQ0KOjk1Ujo6QUNPVy9BLzEyMzQNCjo5NOE6O
lNBRkUvLzM1eA0KOjE3Qjo6QUNUSS8vWQ0KOjE2UjpGSU4NCjozNUI6SVNJTiBBMkM0RTZHOEkwSzINC
joyMkg6OkNBT1AvLONBUOgNCjo5MEE6Ok1SS1QvLORJUOMvMSwzNDU2Nzg5MDEyMzQ1DQo6OTRCOjpQU
klDLy9MTUFSLOFCQ0QNCjo5OEE6OlBSSUMvLzE5
OTkxMjMxDQo6OTNCOjpBR0dSLOEyQzRFNkc4LOEyQzQvTjEsMzQ1Njc4OTAxMjM0NQ0KOjE2UjpTVUJC
QUwNCjo5M0M6OlBFTkQvLOFNT1IvQVZBSS9OMSwzNDU2Nzg5MDEyMzQ1DQo6OTRCOjpTQUZFLOEyQzRF
Nkc4LOEyQzQveA0KOjcwQzo6U1VCQi8veA0KOjE
2UzpTVUJCQUwNCjo5OUE6OkRBQUMvLO4xMjMNCjoxOUE6OkhPTEQvLO5VU0QxLDM0DQo6MTlBOjpCTO9
LLy9OVVNEMSwzNA0KOjE5QTo6QUNSVS8vTlVTRDEsMzQNCjo5MkI6OkVYQ0gvL1VTRC9FVVIvMSwzNDU
2Nzg5MDEyMzQ1DQo6NzBFOjpIT0xELy94DQo6MT
ZTOkZJTg0KOjE2UzpTVUJTQUZFDQo6MTZSOkFERElORk8NCjo5NVI6Ok1FT1IvQTJDNEU2RzgveA0KOj
k1Ujo6TUVSRS9BMkM0RTZHOC94DQo6MTlBOjpITOxQLy9OVVNEMSwzNA0KOjE5QTo6SE9MUy8vTlVTRD
EsMzQNCjoxNlM6QURESU5GTw==</ns:Body>
</ns:DataPDU>
```

## MX Message Data PDU Content Example

The following is an example of the different content parts of the Data PDU of a camt.029.001.01 (MX message type) message.

### xmldata (Data PDU)

```
<?xml version="1.0"?>
<ns:DataPDU
       xmlns:ns="urn:swift:saa:xsd:saa.2.0">
     <ns:Header>
```

```
      <ns:Message>
         <ns:SenderReference>MXWebM522237</ns:SenderReference>
         <ns:MessageIdentifier>camt.029.001.01</ns:MessageIdentifier>
         <ns:Format>AnyXML</ns:Format>
         <ns:SubFormat>Input</ns:SubFormat>
         <ns:Sender>
           <ns:DN>o=ptsausaa,o=swift</ns:DN>
           <ns:FullName>
              <ns:X1>PTSAUSAAXXX</ns:X1>
           </ns:FullName>
      </ns:Sender>
      <ns:Receiver>
        <ns:DN>o=ptsausaa,o=swift</ns:DN>
        <ns:FullName>
              <ns:X1>PTSAUSAAXXX</ns:X1>
        </ns:FullName>
      </ns:Receiver>
      <ns:InterfaceInfo>
        <ns:UserReference>MXWebM522237</ns:UserReference>
      </ns:InterfaceInfo>
      <ns:NetworkInfo>
        <ns:IsNotificationRequested>true</ns:IsNotificationRequested>
        <ns:Service>swift.generic.ia!x</ns:Service>
      </ns:NetworkInfo>
    </ns:Message>
  </ns:Header>
   <ns:Body>
<ns:AppHdr
        xmlns:ns="urn:swift:xsd:$ahV10">
   <ns:MsgRef>REF10610311505</ns:MsgRef>
   <ns:CrDate>2006-10-31T03:05:41.502</ns:CrDate>
</ns:AppHdr>
<ns:Document
        xmlns:ns="urn:swift:xsd:swift.eni$camt.029.001.01">
   <ns:camt.029.001.01>
     <ns:Assgnmt>
        <ns:Id>RCUSTA20050001</ns:Id>
        <ns:Assgnr>AAAAGB2L</ns:Assgnr>
        <ns:Assgne>CUSAGB2L</ns:Assgne>
        <ns:CreDtTm>2005-01-27T11:04:27</ns:CreDtTm>
      </ns:Assgnmt>
      <ns:RslvdCase>
        <ns:Id>CCCC-MOD-20050127-0003</ns:Id>
        <ns:Cretr>CUSAGB2L</ns:Cretr>
      </ns:RslvdCase>
      <ns:Sts>
         <ns:Conf>MODI</ns:Conf>
      </ns:Sts>
     </ns:camt.029.001.01>
   </ns:Document></ns:Body>
   </ns:DataPDU>
```

## MX Header

```
<ns:AppHdr
      xmlns:ns="urn:swift:xsd:$ahV10">
   <ns:MsgRef>REF10610311505</ns:MsgRef>
   <ns:CrDate>2006-10-31T03:05:41.502</ns:CrDate>
</ns:AppHdr>
```

## MX Document

```
<ns:Document
      xmlns:ns="urn:swift:xsd:swift.eni$camt.029.001.01">
   <ns:camt.029.001.01>
     <ns:Assgnmt>
       <ns:Id>RCUSTA20050001</ns:Id>
       <ns:Assgnr>AAAAGB2L</ns:Assgnr>
       <ns:Assgne>CUSAGB2L</ns:Assgne>
       <ns:CreDtTm>2005-01-27T11:04:27</ns:CreDtTm>
     </ns:Assgnmt>
     <ns:RslvdCase>
       <ns:Id>CCCC-MOD-20050127-0003</ns:Id>
       <ns:Cretr>CUSAGB2L</ns:Cretr>
     </ns:RslvdCase>
     <ns:Sts>
       <ns:Conf>MODI</ns:Conf>
     </ns:Sts>
   </ns:camt.029.001.01>
</ns:Document>
```