# **Integration**

## **Introduction:**

### **What and Why?**

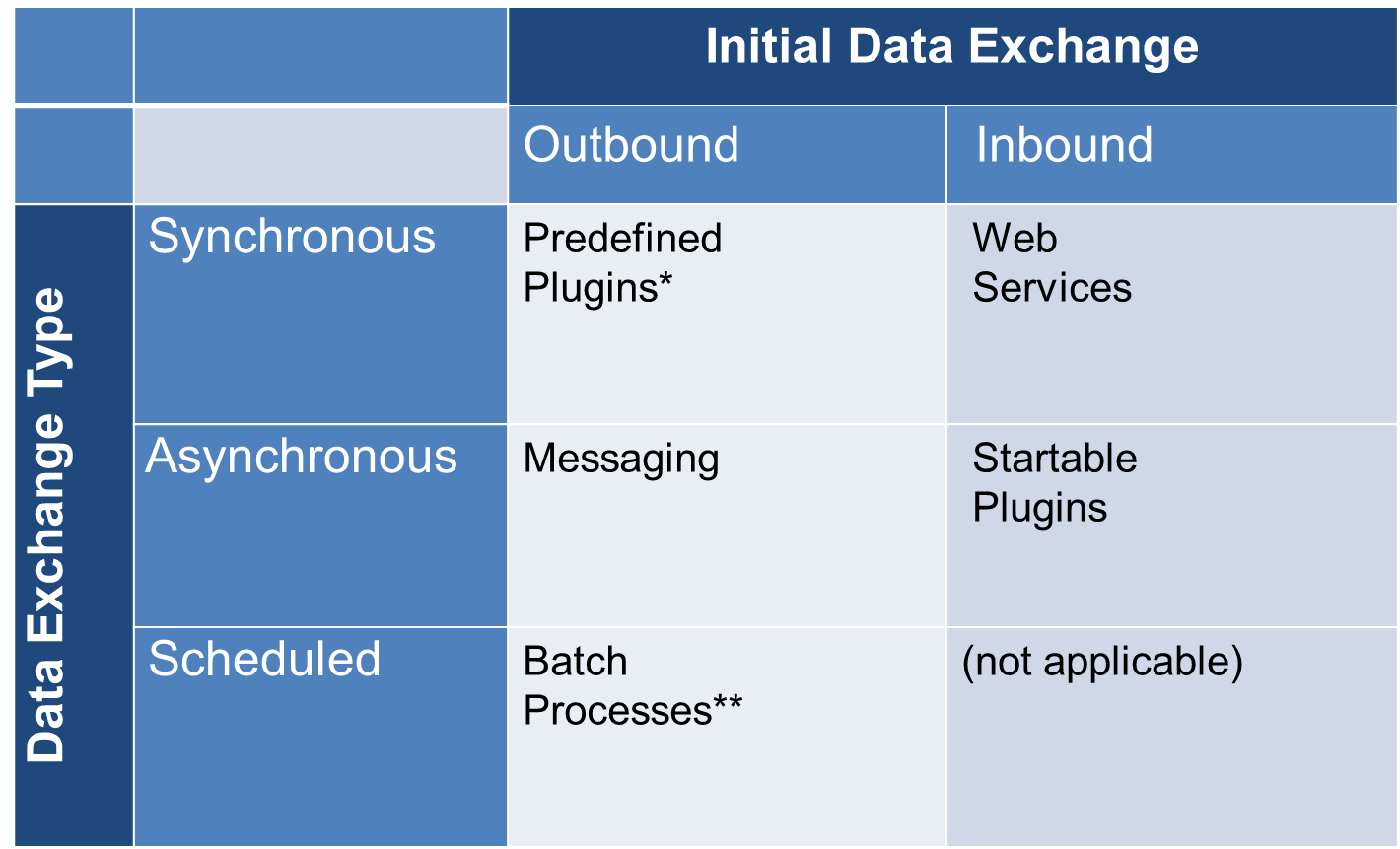
* **Integration** is the communication between Guidewire applications and external systems

### **Integration technologies:**

* + Files
  + Database tables
  + Remote procedure calls (RPC)
  + Message queues

### **Integration mechanisms:**

* Guidewire integration mechanisms are features used to build integration points that typically make use of integration technologies.



"Outbound", which means the data exchange is initiated by Guidewire, and "Inbound", which means the data exchange is initiated by the external system

## **XML:**

### **What is an XML?**

Extensible Markup Language (**XML**) is a hierarchy data structure designed to transport and store data

### **What is an XSD?**

An XSD (XML Schema Document) describes the structure for elements in one or more XML documents

### **Untyped and strongly typed XML**

* If no XSD exists, then the parsed XML is untyped XML.
* If an existing XSD validates and informs the XML, then the parsed XML is strongly typed.

### **The XmlElement class**

* Represent elements from XML
* A **QName** is a **qualified name** for an XML element, consisting of a namespace URI and a local part

### **XML model:**

* An **XML model** is a file that generates dynamic XML based on an input object
* XML Model editor creates GX Models

#### **Steps to create XML model:**

* + Create GX model
  + Add fields to the model
  + Use the model to generate the message payload

## **Plugins:**

* In Guidewire, a plugin is a Gosu, OSGi Java, or Java class that contains a set of methods called by internal code to execute application functionality.
* A plugin is a .gwp file

### **Types of plugins:**

1. Predefined plugin: Used to implement customizable behavior for predefined points in the application
2. Startable plugin: Used to listen for incoming messages and processes them asynchronously
3. Messaging plugin: Used to send messages to external systems and process the resulting replies.

### **Steps to create a plugin:**

1. Determine plugin requirements
2. Write plugin class in Guidewire Studio:
   1. Create plugin class
   2. Implement an interface
   3. Override the methods of interface to perform the required task.
3. Create or modify and then configure the plugin registry file:
4. Create plugin registry file (Registry-> New -> Plugin)
5. Configure Plugin(Class, Parameters)
6. Deploy the changes: Restart the server

## **Web service – An integration Mechanism:**

### **What is a web service?**

* A **web service** is a collection of application program interfaces (APIs) that lets one application send and receive information from another, usually synchronously

### **What is a WSDL?**

* A **WSDL** (Web Services Description Language) is an XML file that defines the operations available for a web service and the required input and output values
  + Generated by the "publisher"
  + Used by the "consumer" to create requests

#### **Structure of WSDL:**

### **Types of web services used in Guidewire:**

1. RPC Encoded web service
2. Document literal (WS-I) web service

### **Operations:**

1. Publishing a web service: GW creates a gosu class(web service) and generates a WSDL for the web service, External system consumes that web service using the WSDL
2. Consuming a web service: External system creates a web service and gives a WSDL to GW which is used to consume that web service.

### **Consuming web service:**

#### **Steps to consume RPC Encoded web service:**

1. Create a new **web service endpoint**:
2. /configuration/config/RPCE-Encoded Web Services -> New -> Web Service(ac.rws)
3. Load the resources:

-Specify the WSDL URL

* 1. To fetch WSDL, the external system must be running

1. Deploy your changes: Restart the server
2. Reference the web service in Gosu as needed

#### **Steps to consume WS-I web service:**

1. Create a package for the web service collection: Normal package
2. Create a new **web service collection**:

* Select package
* New 🡪 Webservice Collection

1. Fetch the resources : Specify the WSDL URL
2. Deploy your changes : Restart the server
3. Reference the web service in Gosu as needed

### **Publishing a web service:**

* A **Guidewire web service** is a web service hosted by the Guidewire application that external systems use.
* Guidewire supports WS-I web services

#### **Steps to create a Guidewire WS-I web service:**

1. Create the package: Normal package
2. Create the Gosu class: Normal Gosu class
3. Add the web service annotations:

Required

* + @WsiWebService annotation is required for WS-I web service class

Optional

* + @WsiPermissions specifies user permissions; default to soap admin
  + @WsiAvailability first usable run level; default is maintenance
  + @WsiExportable – For a GOSU class whose instances are used as return types for webservice and class need to be marked as final

1. Add methods
   * Method that modify Guidewire data must create and commit a bundle
2. Deploy your changes:

* Deploy happens when consumer requests WSDL
* Client requests WSDL
* http://<host>:<port>/<appcode>/ws/<package>/<class>?WSDL

1. Gosu classes for return type of Webservice should have @WSIExportable annotation
2. **Regen -wsi-local** to create a local WSDl for created webservice to test it.

#### **Steps to consume a Guidewire web service by External system:**

* 1. Guidewire creates the WSDL
  2. A third-party tool, such as Apache CXF or Axis, creates client code as needed from the WSDL
  3. The client code must then establish a connection and call the API

## **Batch process:**

### **What and Why?**

A **batch process** is a background process that performs tasks independent of a user.

The mechanism that manages the distribution of work is called a work queue, a way to achieve batch process job

### **Types of Batch process:**

1. **Predefined batch processes:** a batch process that comes with the base application.
2. **Custom batch process:** a batch process created by an integration developer.

### **BatchProcessType typelist:**

### **ProcessHistory:**

* An Event Aware entity
* instance manages a batch process
* Events can trigger messaging relevant to the running of the batch process

### **Running batch process:**

BatchProcessTypeUsage typelist.: Defines the circumstances under which the batch process can be executed.

Typecodes:

* + Schedulable
  + UI Runnable
  + API Runnable
  + Maintenance only

-**UI Runnable**: Through internal tools, To access Internal Tools, the EnableInternalTools parameter in config.xml must be set to true

**-Schedulable:** Defines batch process schedules

* + …\config\scheduler\scheduler-config.xml

<CronSchedule…\> attributes define when process runs

-**API Runnable:**

* Command line syntax
  + **maintenance\_tools -startprocess processName -user user -password password   
     –server server**
* Web service syntax
  + **processID =MaintenanceToolsAPI.startBatchProcess(processName**)

### **Steps to implement custom Batch Process:**

1. Add a batch process typecode to the BatchProcessType typelist
   * Every batch process must be listed in this typelist.
2. Define typecode categories to specify the batch process execution options
   * one or more typecode categories are taken from the BatchProcessTypeUsage typelist
3. Create a new custom batch process class
   * Extends BatchProcessBase, implements a specific constructor syntax, and overrides doWork() function.
   * Optional methods: checkInitialConditions() ,requestTermination()
   * Optional properties: Exclusive
4. Implement/modify the ProcessesPlugin class to instantiate and return the custom batch process object. ProcessPlugin overrides a function createBatchProcess() which contains a switch statement to list all the custom batch process code implementation classes

PolicyCenter 8.0.x and ClaimCenter 8.0.x both implement IProcessesPlugin.gwp in the base application. BillingCenter 8.0.x and ContactManager 8.0.x do not implement IProcessesPlugin.gwp in the base application.

* + In ClaimCenter and PolicyCenter, there is an existing ProcessesPlugin class. To implement a custom batch process, you must add a case statement to the switch statement as necessary.

1. Deploy your changes

## **Messaging:**

### **What?**

* **Messaging** is an asynchronous integration mechanism
  + Triggered by new or changed business data, messages are sent from a Guidewire application to an external system
  + An External system replies to the message back to the Guidewire application

### **Physical message table (xx\_Message):**

* Stores messages
* An exception in the initial sending of the message throws an error. An error can also occur in the external system that the message is being sent to. In either case, Guidewire reports an error with the message. Guidewire will always retries to resend the message with an error. The message status is Retryable Error.

### **Stages of messaging:**

1. Triggering - message creation is triggered by adding an event
2. Payload generation
3. Sending Messages
4. Acknowledging Messages

### **Primary components:**

1. Event
2. Event Fired Rule Set
3. Message Payload
4. Message
5. Destination
6. Message Table
7. Message Plugins

### **Primary Entities needed:**

#### **Event Aware entity:**

* The EventAware entity is an internal, non-persistent entity that implements the messaging interface. For all entities that implement the EventAware delegate, the Guidewire application automatically creates specific events: entityNameChanged, entityNameRemoved, and entityNameAdded.

#### **Message Context Entity:**

* MessageContext is an internal, non-persistent entity
* used in Event Fired rules to create messages
* Contains information such as the event name, messaging destination ID, and the Root object.

#### **Message Entity:**

* Contains information such as event name, message root and payload

#### **Delegate:**

Delegate, Guidewire feature is an entity associated with an implemented interface that multiple parent entities can reuse.

#### **Internal, non-persistent:**

Virtual entities that are created a run-time and do not store data in the database.

### **Stages of Messaging – Triggering:**

* Triggering a message involves message creation.
* Message creation is done by using Event Fired Rules which involves adding an event.
* Implementing Event Aware delegate creates specific events automatically to that entity
* Can also create custom events manually through code:

**entity.addEvent("customEventName")**

* Message events cannot be triggered for non-EventAware entities.

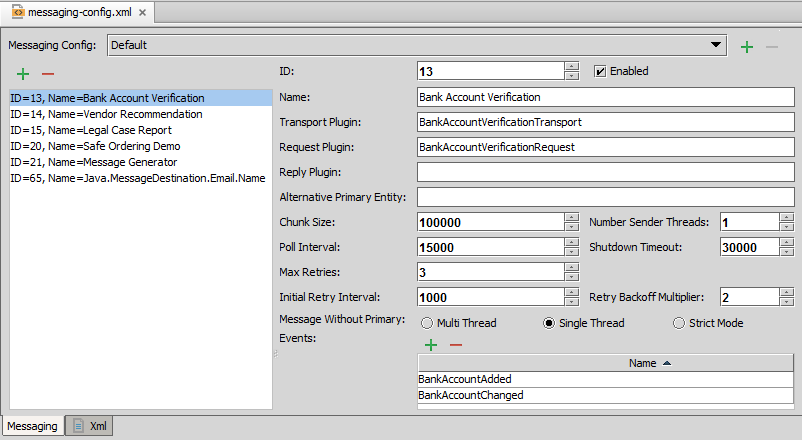
### **Primary Component – Destination:**

* A message **destination** is an abstraction of an external system that receives and replies to messages
* Each Guidewire message is sent to a single destination only. If there is a situation where multiple external systems need to receive the same information in response to a single event, then the Event Fired rules should create multiple messages, one for each destination of interest. (The term "multicasting" means that the same message is sent simultaneously to multiple receivers. Guidewire message does not make use of multicasting.)
* The mechanism for the external system response depends on if the message is synchronous or asynchronous. For synchronous replies, often the message transport plugin is the reply mechanism. For asynchronous replies, often a Guidewire Application published web service or a message reply plugin is the reply mechanism.

#### **Defining a message destination:**

* **ID**: The message ID must be an integer from 0 to 63, inclusive. Values greater than 63 are reserved by Guidewire.
* **Transport Plugin:** The message transport plugin is responsible for sending your message to the external system. The nature of this plugin will vary depending on the external system and/or the technology used to communicate to the external system (such as a JMS queue, a web service request to an external system, or an FTP call). Every destination must provide a message transport plugin implementation, but multiple destinations can use the same implementation.
* **Request Plugin:** The message request plugin transforms a message's payload immediately before sending it. For example, two messages might be sent from a single event, but data received in a reply to the first message (such as an external identification code) might need to be included in the second. Because this identifier is not known at the time the message payloads are initially created, the request plugin is needed to transform the payload of the second message. If a given destination requires no payload transformation, then this plugin is not implemented.
* **Reply plugin:** The message reply plugin initializes a listener queue and associates it with the destination. This plugin is used for destinations that receive asynchronous replies via a listener queue. If a given destination does not reply asynchronously (or it does reply asynchronously, but the replies are received through some other mechanism, such as a web service), then this plugin is not implemented.

**Example:**



### **Stages of Messaging – payload generation:**

#### **Event Fired Rule Set:**

* Responsible for

Creating messages

Generating message payloads

#### **Message Payload:**

This is the data generated in the event fired rule that is sent to external system.

The payload typically:

* + Contains dynamically generated data
  + Is formatted based on the needs of the external system

##### **Ways of generating payloads:**

1. XML model
2. GOSU Templates

###### **XML models:**

* An **XML model** is a file that generates dynamic XML based on an input object
* A key property - Defines a unique identifier for an entity represented in the XML
* A normal property - By default exported when containing instance is exported – this behavior can be changed
* XML Model editor creates GX Models - File extension is .gx

**Steps to create:**

* 1. Right-click the parent package
  2. Select New 🡪 GX Model
  3. Specify the Entity
  4. Specify a model name
  5. Add fields to the model

###### **GOSU Templates:**

* A **Gosu template** is a file that generates dynamic, formatted text based on input parameters
* **Steps to create:**
  1. Right-click the parent package
  2. Select New 🡪 Gosu Template
  3. Provide a name for the template
* All Gosu template files have a .gst extension
* Input parameter declaration syntax:
  1. **<%@ params ( param1 : type, param2 : type, …) %>**
* Template expression syntax
  1. **${param1}** or **<%= param1 %>**
* Evaluate scriptlet syntax
  1. **<% if (something=true) { %> … <% } %>**

#### **Creating an Event Fired Rule:**

1. Need to verify the Destination ID(Highest level of Hierachy)
2. Need to verify the messageContext.Root type object with the entity we are going to use (second highest level of hierarchy)
3. Need to verify the Event Name using messageContext.EventName
4. Optional logic
5. **messageContext.createMessage(PayloadString) –** Used for creating payload

### **Stages of Messaging – Sending Messages:**

1. **Transport Plugin (Required Plugin)** is responsible for sending messages and If acknowledgements are expected synchronously, the transport plugin also manages acknowledgements.
2. **Request Plugin (Optional Plugin)** is responsible for late binding. Late binding is the process of changing a payload immediately before the message is sent to include values that cannot be determined when the message is generated.

**Sender RefID**: When message acknowledgement is asynchronous, the external system needs a way to identify which message it is acknowledging. To do this, a SenderRefID is assigned to the message when it is sent. When the external system sends an acknowledgement, it uses the SenderRefID to identify the message. This is an optional identifier. When GW determines Sender RefID then it is set by request plugin during late binding. When external system determines Sender RefID then it is received by reply plugin(asynchronous)/ transport plugin (synchronous) and set to the message.

#### **Steps to Implement MessageRequest Plugin:**

1. Create the request plugin class
   1. Implement the **MessageRequest** interface
   2. Implement **beforeSend()** as needed
2. Create the plugin registry
3. Configure the plugin
4. Specify the destination plugin
5. Deploy your changes

#### **Steps to Implement MessageTransport Plugin:**

1. Create the message transport plugin class
   1. Implement the MessageTransport interface
   2. Implement send() as needed
2. Create the plugin registry
3. Configure the plugin
4. Specify the destination plugin
5. Deploy your changes

#### **Handling errors in sending a message:**

Sometimes a message transport plugin has a send error and expects it to be temporary. To support this common use case, if the message transport plugin throws an exception from its send method, the Guidewire application will retry to send the message after a specified amount of delay time, and will continue to retry sending the message multiple times.

When there is an exception is thrown in a transport plugin class in a send method, the Guidewire application response is to retry sending the message until reaching the maximum number of retries. The message's destination configuration defines the value for the maximum number of retries. When reaching the maximum number of retries, Guidewire suspends the destination.

Destinations have specific parameters that impact how messages are retried.

* Initial Retry Interval - The amount of time (in milliseconds) to wait before attempting to resend a message after an exception is thrown.
* Max Retries - The number of retries to attempt
* Retry Backoff Multiplier - The amount to increase the time between resends, specified as a multiplier of the time previously attempted.
* Shutdown Timeout - The length of time to wait before attempting to shut down the messaging subsystem.

#### **Safe Ordering:**

* Guidewire has a built-in mechanism called safe ordering
  + Its primary purpose is to ensure that the external system gets related messages one at a time
* When safe ordering is enforced:
  + For a given destination/entity pair, only one message is sent
  + The external system must acknowledge the first message before the next message is sent

By default, Safe ordering in

CC: applies only to messages related directly or indirectly to instances of Claim

PC: applies only to messages related directly or indirectly to instances of Account

BC: not used for any messages

CM: applies to ABContact

You can re configure them for specific destination by specifying “Alternative Primary Entity” in messaging-config.xml

### **Stages of Messaging – Acknowledging Messages:**

1. **Transport Plugin:** For synchronous responses, the transport plugin used to manage acknowledgements.
2. **Reply Plugin:** For asynchronous responses, a message reply plugin is used to manage acknowledgements.
3. **GW Published web service:** For asynchronous responses, we can also use GW webservice to manage acknowledgements.

**Message acknowledgement** is the process in which Guidewire interprets a messaging response from an external system

Guidewire can interpret a response to be one of three types: positive acknowledgement, error, and duplicate response. Guidewire applications must also decide how to react when the external system never sends a response to a message.

**Message and Message History Table:**

* A message can move from the message table to the message history table if the message is positively acknowledged or skipped. Otherwise, the message remains in the message table.

#### **Steps to Implement MessageReply Plugin:**

1. Create the message reply plugin class
   1. Implement the MessageReply interface
   2. Implement initTools() as needed
2. Create the plugin registry
3. Configure the plugin
4. Specify the destination plugin
5. Deploy your changes

**Message.reportAck():** Used for reporting positive acknowledgement.

**Message.reportError(retryTime):** Used when the message needs to be retried and the message has not reached its maximum number of retries.

**Message.reportError(errorCategory):** Used When the message needs to be retried, but the message has reached its maximum number of retries.