

Guidewire BillingCenter®

BillingCenter Integration Guide

RELEASE 8.0.4

Copyright © 2001-2015 Guidewire Software, Inc. All rights reserved.

Guidewire, Guidewire Software, Guidewire ClaimCenter, Guidewire PolicyCenter, Guidewire BillingCenter, Guidewire Reinsurance Management, Guidewire ContactManager, Guidewire Vendor Data Management, Guidewire Client Data Management, Guidewire Rating Management, Guidewire InsuranceSuite, Guidewire ContactCenter, Guidewire Studio, Guidewire Product Designer, Guidewire Live, Guidewire DataHub, Guidewire InfoCenter, Guidewire Standard Reporting, Guidewire ExampleCenter, Guidewire Account Manager Portal, Guidewire Claim Portal, Guidewire Policyholder Portal, Gosu, Deliver Insurance Your Way, and the Guidewire logo are trademarks, service marks, or registered trademarks of Guidewire Software, Inc. in the United States and/or other countries.

All other trademarks are the property of their respective owners.

This material is confidential and proprietary to Guidewire and subject to the confidentiality terms in the applicable license agreement and/or separate nondisclosure agreement.

Guidewire products are protected by one or more United States patents.

Product Name: Guidewire BillingCenter

Product Release: 8.0.4

Document Name: BillingCenter Integration Guide

Document Revision: 02-July-2015

Contents

| | |
|--|----------|
| About BillingCenter Documentation | 9 |
|--|----------|

Part I Planning Integration Projects

| | |
|--|-----------|
| 1 Integration Overview | 13 |
| Overview of Integration Methods | 13 |
| Important Information about BillingCenter Web Services | 16 |
| BillingCenter Integration With Policy Administration Systems | 16 |
| Preparing for Integration Development | 17 |
| Integration Documentation Overview | 18 |
| Regenerating Integration Libraries and WSDL | 20 |
| What are Required Files for Integration Programmers? | 21 |
| Public IDs and Integration Code | 22 |

Part II Web Services

| | |
|---|-----------|
| 2 Web Services Introduction | 27 |
| What are Web Services? | 27 |
| What Happens During a Web Service Call? | 28 |
| Reference of All Built-in Web Services | 29 |
| 3 Publishing Web Services | 31 |
| Web Service Publishing Overview | 32 |
| Publishing and Configuring a Web Service | 36 |
| Testing Web Services with Local WSDL | 40 |
| Generating WSDL | 42 |
| Adding Advanced Security Layers to a Web Service | 45 |
| Web Services Authentication Plugin | 49 |
| Checking for Duplicate External Transaction IDs | 51 |
| Request or Response XML Structural Transformations | 51 |
| Reference Additional Schemas in Your Published WSDL | 52 |
| Validate Requests Using Additional Schemas as Parse Options | 52 |
| Invocation Handlers for Implementing Preexisting WSDL | 53 |
| Locale Support | 56 |
| Setting Response Serialization Options, Including Encodings | 57 |
| Exposing Typelists and Enumerations as String Values | 57 |
| Transforming a Generated Schema | 58 |
| Login Authentication Confirmation | 58 |
| Stateful Session Affinity Using Cookies | 59 |
| Calling a BillingCenter Web Service from Java | 59 |
| 4 Calling Web Services from Gosu | 65 |
| Adding Configuration Options | 72 |
| One-Way Methods | 79 |
| Asynchronous Methods | 80 |
| MTOM Attachments with Gosu as Web Service Client | 81 |

| | |
|---|------------|
| 5 Billing Web Services..... | 83 |
| Billing Web Services Overview | 83 |
| Policy Administration System Core Web Service APIs (BillingAPI) | 85 |
| Invoice Details Web Service APIs (InvoiceDetailsAPI) | 106 |
| Payments Web Service APIs (PaymentAPI) | 107 |
| Payment Instrument Web Service APIs (PaymentInstrumentAPI) | 112 |
| Billing Summary Web Service APIs (BillingSummaryAPI) | 113 |
| Trouble Ticket Web Service APIs (TroubleTicketAPI)..... | 114 |
| Other Billing Web Service APIs (BCAPI)..... | 115 |
| 6 General Web Services | 125 |
| Mapping Typecodes to External System Codes..... | 125 |
| Importing Administrative Data..... | 127 |
| Maintenance Tools Web Service | 128 |
| System Tools Web Services | 129 |
| Workflow Web Services..... | 130 |
| Profiling Web Services | 131 |

Part III Plugins

| | |
|---|------------|
| 7 Plugin Overview | 135 |
| Overview of BillingCenter Plugins..... | 136 |
| Error Handling in Plugins..... | 141 |
| Temporarily Disabling a Plugin | 141 |
| Example Gosu Plugin | 141 |
| Special Notes For Java Plugins..... | 142 |
| Getting Plugin Parameters from the Plugins Registry Editor..... | 143 |
| Writing Plugin Templates For Plugins That Take Template Data..... | 144 |
| Plugin Registry APIs | 145 |
| Plugin Thread Safety | 147 |
| Reading System Properties in Plugins | 151 |
| Do Not Call Local Web Services From Plugins..... | 152 |
| Creating Unique Numbers in a Sequence..... | 152 |
| Restarting and Testing Tips for Plugin Developers | 153 |
| Summary of All BillingCenter Plugins..... | 153 |
| 8 Billing Plugins | 159 |
| Account Evaluation Calculation Plugin | 160 |
| Incentive Calculation Plugin..... | 162 |
| Commission Plugins | 162 |
| Delinquency Processing Customization Plugin | 165 |
| Numbers and Sequences Plugin | 166 |
| BillingCenter Parameter Calculation Plugin..... | 167 |
| Billing Instruction Execution Customization Plugin | 167 |
| Policy Period Information Customization Plugin | 168 |
| BillingCenter Application Event Customization Plugin..... | 170 |
| ChargeInitializer Plugin | 172 |
| Payment Plan Plugin..... | 174 |
| Invoice Plugin..... | 176 |
| Invoice Assembler Plugin..... | 177 |
| Invoice Stream Plugin..... | 180 |
| Invoice Item Exception Information Plugin | 182 |
| Date Sequence Plugin | 183 |
| Agency Cycle Distribution Pre-fill Customization Plugin | 184 |

| | |
|---|------------|
| Premium Report Customization Plugin | 185 |
| Account Information Customization Plugin..... | 187 |
| Producer Information Customization Plugin | 187 |
| Account Distribution Limit Plugin | 187 |
| Collateral Cash Plugin | 188 |
| Agency Distribution Disposition Plugin..... | 189 |
| Policy System Plugin | 189 |
| Direct Bill Payment Plugin..... | 192 |
| Suspense Payment Plugin..... | 197 |
| Funds Tracking Plugin | 198 |
| 9 Authentication Integration | 207 |
| Overview of User Authentication Interfaces | 207 |
| User Authentication Source Creator Plugin | 209 |
| User Authentication Service Plugin | 211 |
| Deploying User Authentication Plugins | 214 |
| Database Authentication Plugins | 215 |
| ContactManager Authentication | 216 |
| 10 Document Management | 217 |
| Document Management Overview | 217 |
| Choices for Storing Document Content and Metadata..... | 219 |
| Document Storage Plugin Architecture | 221 |
| Implementing a Document Content Source for External DMS | 222 |
| Storing Document Metadata In an External DMS | 225 |
| The Built-in Document Storage Plugins..... | 226 |
| Asynchronous Document Storage | 227 |
| APIs to Attach Documents to Business Objects..... | 228 |
| Retrieval and Rendering of PDF or Other Input Stream Data | 229 |
| 11 Document Production | 231 |
| Document Production Overview | 231 |
| Document Template Descriptors | 239 |
| Generating Documents from Gosu | 250 |
| 12 Encryption Integration | 253 |
| Encryption Integration Overview | 253 |
| Changing Your Encryption Algorithm Later | 258 |
| Encryption Features for Staging Tables | 259 |
| 13 Management Integration | 261 |
| Management Integration Overview | 261 |
| The Abstract Management Plugin Interface | 262 |
| Integrating With the Included JMX Management Plugin..... | 263 |
| 14 Other Plugin Interfaces | 265 |
| Automatic Address Completion and Fill-in Plugin | 265 |
| Phone Number Normalizer Plugin | 266 |
| Testing Clock Plugin (Only For Non-Production Servers) | 266 |
| Work Item Priority Plugin | 268 |
| Official IDs Mapped to Tax IDs Plugin | 268 |
| Preupdate Handler Plugin..... | 268 |
| Defining Base URLs for Fully-Qualified Domain Names | 270 |
| Exception and Escalation Plugins..... | 271 |
| 15 Startable Plugins | 273 |
| Startable Plugins Overview | 273 |

| | |
|---|------------|
| Writing a Startable Plugin..... | 274 |
| Configuring Startable Plugins to Run on All Servers..... | 277 |
| Java and Startable Plugins | 280 |
| Persistence and Startable Plugins | 280 |
| 16 Multi-threaded Inbound Integration..... | 281 |
| Multi-threaded Inbound Integration Overview..... | 281 |
| Inbound Integration Configuration XML File | 283 |
| Inbound File Integration | 285 |
| Inbound JMS Integration | 290 |
| Custom Inbound Integrations | 292 |
| Understanding the Polling Interval and Throttle Interval..... | 299 |

Part IV Messaging

| | |
|---|------------|
| 17 Messaging and Events | 303 |
| Messaging Overview | 304 |
| Message Destination Overview | 315 |
| List of Messaging Events in BillingCenter..... | 323 |
| Generating New Messages in Event Fired Rules | 331 |
| Message Ordering and Multi-Threaded Sending | 337 |
| Late Binding Data in Your Payload | 342 |
| Reporting Acknowledgements and Errors | 343 |
| Tracking a Specific Entity With a Message | 347 |
| Implementing Messaging Plugins..... | 347 |
| Resynchronizing Messages for a Primary Object..... | 353 |
| Message Payload Mapping Utility for Java Plugins..... | 356 |
| Message Status Code Reference..... | 357 |
| Monitoring Messages and Handling Errors | 358 |
| Messaging Tools Web Service | 360 |
| Batch Mode Integration | 362 |
| Included Messaging Transports | 363 |

Part V Importing Billing and Account Data

| | |
|---|------------|
| 18 Importing from Database Staging Tables..... | 367 |
| Introduction to Database Staging Table Import | 367 |
| Overview of a Typical Database Staging Table Import | 371 |
| Database Import Performance and Statistics | 377 |
| Table Import Tools | 377 |
| Populating the Staging Tables | 379 |
| Loading BillingCenter-specific Entities | 382 |
| Data Integrity Checks | 389 |
| Table Import Tips and Troubleshooting..... | 390 |
| Staging Table Import of Encrypted Properties | 391 |

Part VI Other Integration Topics

| | |
|--|------------|
| 19 Contact Integration..... | 395 |
| Integrating with a Contact Management System | 395 |
| Contact Web Service APIs | 401 |

| | |
|--|------------|
| 20 Multicurrency Integration between BillingCenter and PolicyCenter | 405 |
| Set up Currencies for Multicurrency Integration | 405 |
| Configure Account Numbers for Multicurrency Accounts in BillingCenter | 406 |
| 21 Custom Batch Processing | 407 |
| Overview of Custom Batch Processing | 407 |
| Developing Custom Work Queues | 410 |
| Example Work Queues | 417 |
| Developing Custom Batch Processes | 420 |
| Example Batch Processes | 424 |
| Enabling Custom Batch Processing to Run | 427 |
| Monitoring Batch Processing | 429 |
| Periodic Purging of Batch Processing Entities | 431 |
| 22 Servlets | 433 |
| 23 Data Extraction Integration | 441 |
| Why Gosu Templates are Useful for Data Extraction | 441 |
| Data Extraction Using Web Services | 442 |
| 24 Logging | 445 |
| Logging Overview For Integration Developers | 445 |
| Logging Properties File | 446 |
| Logging APIs for Java Integration Developers | 447 |
| 25 Java and OSGi Support | 451 |
| Overview of Java and OSGi Support | 451 |
| Accessing Entity and Typecode Data in Java | 455 |
| Accessing Gosu Classes from Java Using Reflection | 466 |
| Gosu Enhancement Properties and Methods in Java | 467 |
| Class Loading and Delegation for non-OSGi Java | 467 |
| Deploying Non-OSGi Java Classes and JARs | 468 |
| OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor | 469 |
| Advanced OSGi Dependency and Settings Configuration | 476 |
| Updating Your OSGi Plugin Project After Product Location Changes | 477 |

About BillingCenter Documentation

The following table lists the documents in BillingCenter documentation.

| Document | Purpose |
|------------------------------------|--|
| <i>InsuranceSuite Guide</i> | If you are new to Guidewire InsuranceSuite applications, read the <i>InsuranceSuite Guide</i> for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications. |
| <i>Application Guide</i> | If you are new to BillingCenter or want to understand a feature, read the <i>Application Guide</i> . This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with BillingCenter. |
| <i>Upgrade Guide</i> | Describes how to upgrade BillingCenter from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing BillingCenter application extensions and integrations. |
| <i>New and Changed Guide</i> | Describes new features and changes from prior BillingCenter versions. Intended readers are business users and system administrators who want an overview of new features and changes to features. Consult the "Release Notes Archive" part of this document for changes in prior maintenance releases. |
| <i>Installation Guide</i> | Describes how to install BillingCenter. The intended readers are everyone who installs the application for development or for production. |
| <i>System Administration Guide</i> | Describes how to manage a BillingCenter system. The intended readers are system administrators responsible for managing security, backups, logging, importing user data, or application monitoring. |
| <i>Configuration Guide</i> | The primary reference for configuring initial implementation, data model extensions, and user interface (PCF) files. The intended readers are all IT staff and configuration engineers. |
| <i>Globalization Guide</i> | Describes how to configure BillingCenter for a global environment. Covers globalization topics such as global regions, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who localize BillingCenter. |
| <i>Rules Guide</i> | Describes business rule methodology and the rule sets in BillingCenter Studio. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu. |
| <i>Contact Management Guide</i> | Describes how to configure Guidewire InsuranceSuite applications to integrate with ContactManager and how to manage client and vendor contacts in a single system of record. The intended readers are BillingCenter implementation engineers and ContactManager administrators. |
| <i>Best Practices Guide</i> | A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers. |
| <i>Integration Guide</i> | Describes the integration architecture, concepts, and procedures for integrating BillingCenter with external systems and extending application behavior with custom programming code. The intended readers are system architects and the integration programmers who write web services code or plugin code in Gosu or Java. |
| <i>Gosu Reference Guide</i> | Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration. |
| <i>Glossary</i> | Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications. |

Conventions in This Document

| Text style | Meaning | Examples |
|--------------------------------|---|--|
| <i>italic</i> | Emphasis, special terminology, or a book title. | A <i>destination</i> sends messages to an external system. |
| bold | Strong emphasis within standard text or table text. | You must define this property. |
| narrow bold | The name of a user interface element, such as a button name, a menu item name, or a tab name. | Next, click Submit . |
| <code>monospaced</code> | Literal text that you can type into code, computer output, class names, URLs, code examples, parameter names, string literals, and other objects that might appear in programming code. In code blocks, bold formatting highlights relevant sections to notice or to configure. | Get the field from the <code>Address</code> object. |
| <code>monospaced italic</code> | Parameter names or other variable placeholder text within URLs or other code snippets. | Use <code>getName(first, last)</code> . <code>http://SERVERNAME/a.html</code> . |

Support

For assistance, visit the Guidewire Resource Portal – <http://guidewire.custhelp.com>

part I

Planning Integration Projects

Integration Overview

You can integrate a variety of external systems with BillingCenter by using services and APIs that link BillingCenter with custom code and external systems. This overview provides information to help you plan integration projects for your BillingCenter deployment and provides technical details critical to successful integration efforts.

This topic includes:

- “Overview of Integration Methods” on page 13
- “Important Information about BillingCenter Web Services” on page 16
- “BillingCenter Integration With Policy Administration Systems” on page 16
- “Preparing for Integration Development” on page 17
- “Integration Documentation Overview” on page 18
- “Regenerating Integration Libraries and WSDL” on page 20
- “What are Required Files for Integration Programmers?” on page 21
- “Public IDs and Integration Code” on page 22

See also

- For complete information about the integration between BillingCenter and PolicyCenter, see the documentation set for PolicyCenter, not this documentation set for BillingCenter. In the PolicyCenter documentation set, see “Billing Integration” in the *PolicyCenter Integration Guide*.

Overview of Integration Methods

BillingCenter addresses the following integration architecture requirements:

- **A service-oriented architecture** – Encapsulate your integration code so upgrading the core application requires few other changes. Also, a service-oriented architecture allows APIs to use different languages or platform.
- **Customize behavior with the help of external code or external systems** – For example, implement special account validation logic, use a legacy system that generates account numbers, or query a legacy system.

- **Send messages to external systems in a transaction-safe way** – Trigger actions after important events happen inside BillingCenter, and notify external systems if and only if the change is successful and no exceptions occurred. For example, alert a policy database if anyone changes account information.
- **Flexible export** – Providing different types of export minimizes data conversion logic. Simplifying the conversion logic improves performance and code maintainability for integrating with diverse and complex legacy systems.
- **Predictable error handling** – Find and handle errors cleanly and consistently for a stable integration with custom code and external systems.
- **Link business rules to custom task-oriented Gosu or Java code** – Let Gosu-based business rules in Guidewire Studio or Gosu templates call Java classes directly from Gosu.
- **Import or export data to/from external systems** – There are many methods of importing and exporting data to BillingCenter, and you can choose which methods make the most sense for your integration project.
- **Use clearly-defined industry standard protocols for integration points** – BillingCenter includes built-in APIs to retrieve accounts, create users, manage documents, trigger events, validate records, and trigger bulk import/export. However, most legacy system integrations require additional integration points customized for each system.

To achieve these goals, the BillingCenter integration framework includes multiple methods of integrating external code with the BillingCenter product. The primary integration methods:

- **Web service APIs** – Web service APIs are a general-purpose set of application programming interfaces that you can use to query, add, or update Guidewire data, or trigger actions and events programmatically. Because these APIs are web services, you can call them from any language and from any operating system. A typical use of the web service APIs would be to programmatically add accounts into BillingCenter. You can use the built-in SOAP APIs, but you can also design your own SOAP APIs in Gosu and expose them for use by remote systems. Additionally, your Gosu code can easily call web services hosted on other computers to trigger actions or retrieve data.
- **Plugins** – BillingCenter plugins are mini-programs that BillingCenter invokes to perform an action or calculate a result. Guidewire recommends writing plugins in Gosu. You can also write plugins in Java. Note that Gosu code can call third-party Java classes and Java libraries.

General categories of plugins include:

- **Messaging plugins** – Send messages to remote systems, and receive acknowledgements of each message. BillingCenter has a sophisticated transactional messaging system to send information to external systems in a reliable way. Any server in the cluster can create a message for any data change. The batch server, in a separate thread and database transaction, sends messages and tracks (waits for) message acknowledgments. For details, see “Messaging and Events” on page 303.
- **Authentication plugins** – Integrate custom authentication systems. For instance, define a user authentication plugin to support a corporate directory that uses the LDAP protocol. Or define a database authentication plugin to support custom authentication between BillingCenter and its database server. For details, see “Authentication Integration” on page 207.
- **Document and form plugins** – Transfer documents to and from a document management system, and help prepare new documents from templates. Additionally, use Gosu APIs to create and attach documents. For details, see “Document Management” on page 217.
- **Inbound integration plugins** – Multi-threaded inbound integrations for high-performance data import. BillingCenter includes built-in implementations for reading files and receiving JMS messages, but you can also write your implementations that leverage the multi-threaded framework. See “Multi-threaded Inbound Integration” on page 281.
- **Other plugins** – For example, compute an evaluation rating for an account based on the number of delinquencies. For more information, see “Summary of All BillingCenter Plugins” on page 153.

See also

- “Plugin Overview” on page 135

- **Templates** – Generate text-based formats that contain combinations of BillingCenter data and fixed data. Templates are ideal for name-value pair export, HTML export, text-based form letters, or simple text-based protocols. For details, see “Data Extraction Integration” on page 441.
- **Database import from staging tables** – Bulk data import into production databases using temporary loading into “staging” database tables. BillingCenter performs data consistency checks on data before importing new data (although it does not run validation rules). Database import is faster than adding individual records one by one. To initially load data from an external system or performing large daily imports from another system, use database staging table import. For details, see “Importing from Database Staging Tables” on page 367.

The following table compares the main integration methods.

| Integration type | Description | What you write |
|------------------|---|---|
| Web services | A full API to manipulate BillingCenter data and trigger actions externally using any programming language or platform using BillingCenter Web services APIs, accessed using the platform-independent SOAP protocol. | <ul style="list-style-type: none"> • To publish web services, write Gosu classes that implement each operation as a class method. • To consume web services that a Guidewire application publishes: write Java or Gosu code on an external system. The external system calls web services using the WSDL that BillingCenter generates. • To consume external web services from Gosu, write Gosu code that uses WSDL from the external system. Gosu creates native types from the WSDL that you download to the Studio environment. • In all cases, define objects that encapsulate only the data you need to transfer to and from BillingCenter. The general name for such objects are Data Transfer Objects (DTOs). Define DTOs as Gosu classes or using XSDs. |
| Plugins | Small programs that BillingCenter calls to perform tasks or calculate a result. Plugins run in the same Java virtual machine (JVM) as BillingCenter. | <ul style="list-style-type: none"> • Gosu classes that implement a Guidewire-defined interface. • Java classes that implement a Guidewire-defined interface. Optionally use OSGi, a Java component system. See “Overview of Java and OSGi Support” on page 451 |
| Messaging code | Changes to application data trigger events that generate messages to external systems. You can send messages to external systems and track responses called acknowledgements. | <ul style="list-style-type: none"> • Gosu code in the Event Fired rule set. • Messaging plugins, most importantly the MessageTransport plugin. There is also the MessageRequest plugin (for pre-processing your payload) and the MessageReply plugin (for asynchronous replies). • Configure one or more messaging destinations in Studio to use your messaging plugins. After you register your plugins in the Plugins editor, you must also use the Messaging editor for each destination. • In all cases, define objects that encapsulate only the data you need to transfer to and from BillingCenter. The general name for such objects are Data Transfer Objects (DTOs). Define DTOs as Gosu classes or using XSDs. |

| Integration type | Description | What you write |
|---------------------------|--|--|
| Guidewire XML (GX) models | The GX model editor in Studio lets you create custom schema definitions (XSDs) for your data to assist in integrations. You customize which properties in an entity (or other type) to export in XML. Then, you can use this model to export XML or import XML in your integrations. For example, your messaging plugins could send XML to external systems, or your web services could take this XML format as its payload from an external system. For more information, see “The Guidewire XML (GX) Modeler” on page 310 in the <i>Gosu Reference Guide</i> . | <ul style="list-style-type: none"> Using the GX model editor in Studio, you customize which properties in an entity (or other type) to export in XML. Various integration code that uses the XSD it creates. |
| Templates | Several data extraction mechanisms that perform database queries and format the data as necessary. For example, send notifications as form letters and use plain text with embedded Gosu code to simplify deployment and ongoing updates. Or design template that exports HTML for easy development of web-based reports of BillingCenter data. | <ul style="list-style-type: none"> Text files that contain small amounts of Gosu code |
| Database import | You can import large amounts of data into BillingCenter by first populating a separate set of staging database tables. Staging tables are temporary versions of the data that exist separate from the production database tables. Next, use BillingCenter web service APIs or command line tools to validate and import that data. | <ul style="list-style-type: none"> Custom tools that take legacy data and convert them into database tables of data. |

Important Information about BillingCenter Web Services

BillingCenter supports WS-I web services. WS-I web services use the SOAP protocol and are compatible with the WS-I standard. For an overview of web services, including a list of all built-in web services, see “Web Services Introduction” on page 27.

| Topic | See |
|--|---|
| Overview of web services | “Web Services Introduction” on page 27 |
| Reference of all built-in web services | “Reference of All Built-in Web Services” on page 29 |
| Publishing a web service on the BillingCenter server | “Publishing Web Services” on page 31 |
| Consuming a web service | “Calling Web Services from Gosu” on page 65 |

BillingCenter Integration With Policy Administration Systems

BillingCenter supports integration with policy administration systems. For example, BillingCenter can receive policy and charge information from your policy administration system. In addition, BillingCenter can call your policy administration system. For example, BillingCenter can notify your policy administration system that an insured failure to make a payment, which could trigger a cancellation.

PolicyCenter Integration with BillingCenter

BillingCenter includes a robust built-in integration with PolicyCenter. For complete information about the integration between BillingCenter and PolicyCenter, see the documentation set for PolicyCenter, not this documentation set for BillingCenter. In the PolicyCenter documentation set, see “Billing Integration” in the *PolicyCenter Integration Guide*.

Policy Administration System Integration with Billing Center

Information in the documentation for BillingCenter covers integration points with policy administration systems in general. BillingCenter has web services and plugin interfaces for integration PolicyCenter. You can use the same web services and plugin interfaces in BillingCenter for integration with other third-party policy administration systems.

The most important topics in the *BillingCenter Integration Guide* for billing-specific integration with policy administration systems are as follows:

- **Web services** – For billing-related web services that your external policy administration system can call, see “Billing Web Services” on page 83.
- **Plugin interfaces** – For billing-related plugins, some of which can call external policy administration systems, see “Billing Plugins” on page 159.

Preparing for Integration Development

During integration development, edit configuration files within the hierarchy of files in the product installation directory. In most cases you modify data only through the Studio interface, which handles any SCM (Source Configuration Management) requests. The Studio interface also copies read-only files to the **configuration** module in the file hierarchy for your files and makes files writable as appropriate.

However, in some cases you need to add files directly to certain directories in the **configuration** module hierarchy, such as Java class files for Java plugin support. The configuration module hierarchy for your files is in the hierarchy:

`BillingCenter/modules/configuration`

This directory contains subdirectories such as:

| Directory under configuration module | Description |
|--------------------------------------|--|
| <code>config/web</code> | Your web application PCF files. |
| <code>config/logging</code> | Your logging configuration files. See “Logging” on page 445. |
| <code>config/templates</code> | This directory contains two types of templates relevant for integration: <ul style="list-style-type: none">• Plugin templates – Use Gosu plugin templates for the small number of plugin interfaces that require them. These plugin templates extract important properties from entities, and they generate text that describes the results. Whenever BillingCenter needs to call the plugin, BillingCenter passes the results to the plugin as text-based parameters.• Messaging templates – Use optional Gosu messaging templates for your messaging code. Use messaging templates to define notification letters or other similar messages contain large amounts of text but a small amount of Gosu code. There are built-in versions some of these templates. To modify the built-in versions in Studio, navigate to Resources → Configuration → Other Resources → Templates. |

| Directory under configuration module | Description |
|--------------------------------------|--|
| config/docmgmt | Document management files. See "Document Management" on page 217. |
| plugins | <p>Your Java plugin files. To add Java files for plugin support, see "Special Notes For Java Plugins" on page 142 and "Overview of Java and OSGi Support" on page 451.</p> <p>Register your plugin implementations in the Plugin registry in Studio. When you register the plugin in the Plugin registry, you can define a <i>plugin directory</i>, which is the name of a subdirectory of the plugins directory. If you do not define a subdirectory, BillingCenter defaults the plugin directory to the shared subdirectory. Be aware that if you are using the Java API introduced in version 8.0, your classes and libraries must be in a subdirectory of the plugin directory called basic. See "Deploying Non-OSGi Java Classes and JARs" on page 468.</p> <p>For more information about registering a plugin, see "Plugin Overview" on page 135 and "Using the Plugins Registry Editor" on page 109 in the <i>Configuration Guide</i>.</p> <p>For a messaging plugin, you must register this information two different registries.</p> <ul style="list-style-type: none"> • the plugin registry in the plugin editor in Studio • the messaging registry in the Messaging editor in Studio. See "Messaging and Events" on page 303 and "Messaging Editor" on page 131 in the <i>Configuration Guide</i>. |

There are some important directories other than the subdirectories for files in the configuration module.

| Other directory | Description |
|---------------------------------|---|
| BillingCenter/bin | <p>Command line tools such as gwbc.bat. Use this for the following integration tasks:</p> <ul style="list-style-type: none"> • Regenerating the Java API libraries and web service (SOAP) WSDL. See "Regenerating Integration Libraries and WSDL" on page 20. • Regenerating the <i>Data Dictionary</i>. See "Data Dictionary Documentation" on page 20 |
| BillingCenter/soap-api | <p>Files related to web services. BillingCenter scripts generate this directory. To regenerate these, see "Regenerating Integration Libraries and WSDL" on page 20.</p> |
| BillingCenter/soap-api/wsi/wsdl | <p>For web services, this directory contains WSDL files generated locally. To regenerate these, see "Regenerating Integration Libraries and WSDL" on page 20.</p> |
| BillingCenter/admin | <p>Command line tools that control a running BillingCenter server. Almost all of these are small Gosu scripts that call public web service APIs.</p> |

Integration Documentation Overview

Use the *Integration Guide* for important integration information. Also, consult other reference documentation during development:

- *API Reference Javadoc Documentation*
- *Data Dictionary Documentation*

API Reference Javadoc Documentation

The easiest way to learn what interfaces are available is by reading *BillingCenter API Reference Javadoc* documentation, which is web browser-based documentation.

There are multiple types of reference documentation.

Java API Reference Javadoc

The Java API Javadoc includes:

- The specification of the plugin definitions for Java plugin interfaces. These also are the specifications for plugins implemented in Gosu.
- The details of full entities, including both the entity data (get and set methods) plus additional methods called domain methods. For a high-level overview of entity differences and different entity access types, see the diagram in “What are Required Files for Integration Programmers?” on page 21. For writing Java plugins, also see “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.
- General Java utility classes.
- After regenerating the generated files, find it in `BillingCenter/java-api/doc/api/index.html`

To regenerate these files, see “Regenerating Integration Libraries and WSDL” on page 20.

Web Service API Reference Documentation

Web Service Description Language (WSDL)

On a running BillingCenter server, you can get up-to-date WSDL from published services. See “Getting WSDL from a Running Server” on page 42

After regenerating SOAP API reference files, there are additional local WSDL files on the server. To regenerate these files, see “Regenerating Integration Libraries and WSDL” on page 20. BillingCenter generates the WSDL at the path

```
BillingCenter/soap-api/wsi/wsdl
```

For more information about web services, see “Web Services Introduction” on page 27.

Web Service Javadoc

For WS-I web services, there is no built-in Javadoc generation because there are no built-in generated libraries for a specific SOAP system. The exact method signatures and syntax vary based on the language which the SOAP implementation that generates libraries from the WSDL.

For more information about web services, see “Web Services Introduction” on page 27.

Gosu Generated Documentation

Integration programmers might want to use the Gosu documentation that you can generate from the command line using the gwbc tool:

```
gwbc regen-gosudoc
```

You will then find the documentation at `BillingCenter/build/gosudoc/index.html`.

This documentation is particularly valuable for integration programmers implementing plugins in Gosu. The information in the Javadoc-formatted files are more relevant for Gosu programmers than Java programmers due to package differences and visibility from Java.

For more information, see “Gosu Generated Documentation (Gosudoc)” on page 38 in the *Gosu Reference Guide*.

Using Javadoc-formatted Documentation

Several types of generated documentation are in web-based Javadoc format. For Javadoc-formatted documentation, to look within a particular package namespace, click in the top-left pane of the documentation on the package name. The bottom left pane displays interfaces and classes listed for that package. If you click on a class, the right pane shows the methods for that class.

Data Dictionary Documentation

Another set of documentation called the *BillingCenter Data Dictionary* provides documentation on classes that correspond to BillingCenter data. You must regenerate the *Data Dictionary* to use it.

To view the documentation, refer to:

```
BillingCenter/build/dictionary/data
```

To regenerate the dictionary, open a command window and change directory to `BillingCenter/bin` and run the command:

```
gwbc regen-dictionary
```

The *BillingCenter Data Dictionary* typically has more information about data-related objects than the API Reference documentation has for that class/entity. The *Data Dictionary* documents only classes corresponding to data defined in the data model. It does not document all API functions or utility classes.

The *Data Dictionary* lists some objects or properties that are part of the data model and the relevant database tables as internal properties. You must not modify internal properties within the database or use any other mechanism. Getting or setting internal data properties might provide misleading data or corrupt the integrity of application data.

Regenerating Integration Libraries and WSDL

You must regenerate the Java API libraries and SOAP WSDL after you make certain changes to the product configuration. Regenerate these files in the following situations:

- After you install a new BillingCenter release
- After you make changes to the BillingCenter data model, such as data model extensions, typelists, field validators, and abstract data types

As you work on both configuration and integration tasks, you may need to regenerate the Java API libraries and SOAP WSDL frequently. However, if you make significant configuration changes and then work on integration at a later stage, wait until you need the APIs updated before regenerating BillingCenter files. Next, you can recompile integration code against the generated libraries, if necessary.

- The location for the Java generated libraries is:
`BillingCenter/java-api/lib`
- The location of the generated Java library documentation:
`BillingCenter/java-api/doc`
- The location for the WS-I web service WSDL is:
`BillingCenter/soap-api/wsi/wsdl`

Call the main `gwbc.bat` file to regenerate these

- For Java development, generate libraries and documentation with:
`gwbc.bat regen-java-api`
- For web service (SOAP) development, generate WSDL using the command:
`gwbc.bat regen-soap-api`

For example, to generate Java API libraries and documentation:

1. In Windows, open a command window.
2. Change your working directory with the following command:
`cd BillingCenter/bin`
3. Type the command:
`gwbc regen-java-api`

As part of its normal behavior while regenerating the documentation for the Java or the SOAP files, the script displays some warnings and errors.

See also

- “What are Required Files for Integration Programmers?” on page 21

What are Required Files for Integration Programmers?

Depending on what kind of integrations you require, there are special files you must use. For example, libraries to compile your Java code against, to import in a project, or to use in other ways.

There are several types of integration files:

- **Web services** – Some files represent files you can use with SOAP API client code.
- **Plugin interfaces** – Some files represent plugin interfaces for your code to respond to requests from BillingCenter to calculate a value or perform a task.
- **Java API libraries** – Some files represent *entities*, which are BillingCenter objects defined in the data model with built-in properties and can also have data model extension properties. For example, Account and Address are examples of Guidewire entities. To access entity data and methods from Java, you need to use Java API libraries. See “Java and OSGi Support” on page 451.

In all cases, BillingCenter entities such as Account contain data properties that can be manipulated either directly or from some contexts using getters and setters (`get...` and `set...` methods).

Depending on the type of integration point, there may be additional methods available on the objects. These additional *domain methods* often contain valuable functionality for you. If an integration point can access both entity data and domain methods, it is said to have access to the *full entities*.

The following table summarizes the different entity integration implications for each integration type. For this table, *full entity instances* means access to all properties and domain methods.

| Entity access | Description | Entities | Necessary libraries |
|----------------------------|---|-----------------------|---------------------|
| Gosu plugin implementation | Plugin interface defined in Gosu. | Full entity instances | None |
| Java plugin implementation | Java code that accesses an entity associated with a plugin interface parameter or return value. | Full entity instances | Java API libraries |

| Entity access | Description | Entities | Necessary libraries |
|-----------------------------|---|--|---|
| Java class called from Gosu | Java code called from Gosu that accesses an entity passed as a parameter from Gosu, or a return result to be passed back to Gosu. | Full entity instances | Java API libraries |
| Web services | WS-I web services are the newer standard for web services. | <p>There is no support for entity types as arguments or return values. Instead, create your own data transfer objects (DTO) as either:</p> <ul style="list-style-type: none"> • Gosu class instances that contain only the properties required for each integration point. • XML objects with structure defined in XSD files. • XML objects with structure defined with the GX modeler. The GX modeler is a tool to generate an XML model with only the desired subset of properties for each entity for each integration point. See "The Guidewire XML (GX) Modeler" on page 310 in the <i>Gosu Reference Guide</i>. | There is no support for entity types as arguments or return values. |

See also

- “Web Services Introduction” on page 27
- “Plugin Overview” on page 135

Public IDs and Integration Code

BillingCenter creates its own unique ID for every entity in the system after it fully loads in the BillingCenter system. However, this internal ID cannot be known to an external system while the external system prepares its data. Consequently, if you get or set BillingCenter information, use unique public ID values to identify an entity from external systems connecting to a Guidewire application.

Your external systems can create this public ID based on its own internal unique identifier, based on an incrementing counter, or based on any system that can guarantee unique IDs. Each entity type must have unique public IDs within its class. For instance, two different Address objects cannot have the same public ID.

However, a policy and an account may share the same public ID because they are different entities.

If loading two related objects, the incoming request must tell BillingCenter that they are related. However, the web service client does not know the internal BillingCenter IDs as it prepares its request. Creating your own public IDs guarantees the web service client can explain all relationships between objects. This is true particularly if entities have complex relationships or if some of the objects already exist in the database.

Additionally, an external system can tell BillingCenter about changes to an object even though the external system might not know the internal ID that BillingCenter assigned to it. For example, if the external system wants to change a contact’s phone number, the external system only needs to specify the public ID of the contact record.

BillingCenter allows most objects associated with data to be tagged with a public ID. Specifically, all objects in the *Data Dictionary* that show the `keyable` attribute contain a public ID property. If your API client code does not need particular public IDs, let BillingCenter generate public IDs by leaving the property blank. However, other non-API import mechanisms require you to define an explicit public ID, for instance database table record import.

If you choose not to define the public ID property explicitly during initial API import, later you can query BillingCenter with other information. For example, you could pass a contact person's full name or taxpayer ID if you need to find its entity programmatically.

You can specify a new public ID for an object. From Gosu, set the `PublicID` property.

Creating Your Own Public IDs

Suppose a company called ABC has two external systems, each of which contains a record with an internal ID of 2224. Each system generates public ID by using the format "`{company}:{system}:{recordID}`" to create unique public ID strings such as "abc:s1:2224" and "abc:s2:2224".

To request BillingCenter automatically create a public ID for you rather than defining it explicitly, set the public ID to the empty string or to `null`. If a new entity's public ID is blank or `null`, BillingCenter generates a public ID. The ID is a two-character ID, followed by a colon, followed by a server-created number. For example, "bc:1234". Guidewire reserves for itself all public IDs that start with a two-character ID and then a colon.

Public IDs that you create must never conflict with BillingCenter-created public IDs. If your external system generates public IDs, you must use a naming convention that prevents conflict with Guidewire-reserved IDs and public IDs created by other external systems.

The prefix for auto-created public IDs is configurable using the `PublicIDPrefix` configuration parameter. If you change this setting, all explicitly-assigned public IDs must not conflict with the namespace of that prefix.

For BillingCenter, the maximum public ID length is 20.

IMPORTANT Integration code must never set a public IDs to a `String` that starts with a two-character ID and then a colon. Guidewire strictly reserves all such IDs. If you use the `PublicIDPrefix` configuration parameter, integration code that sets explicit public IDs also must not conflict with that namespace. Additionally, plan your public ID naming to support large (long) record numbers. Your system must support a significant number of records over time and stay within the public ID length limit.

See also

- “`PublicIDPrefix`” on page 46 in the *Configuration Guide*

part II

Web Services

Web Services Introduction

You can write web service APIs in Gosu and access them from remote systems using the standard web services protocol *SOAP*, the standard Simple Object Access Protocol. Web services provide a language-neutral, platform-neutral mechanism for invoking actions or requesting data from other applications across a network. The SOAP protocol defines request/response mechanisms for translating a function call and its response into XML-based messages, typically across the standard HTTP protocol. BillingCenter publishes its own built-in web service APIs that you can use.

This topic includes:

- “What are Web Services?” on page 27
- “What Happens During a Web Service Call?” on page 28
- “Reference of All Built-in Web Services” on page 29

What are Web Services?

Web services define request-and-response APIs that let you call an API on a remote computer, or even the current computer, using an abstracted well-defined interface. A data format called the Web Service Description Language (WSDL) describes available web services that other systems can call using the SOAP protocol. Many languages and third-party packages provide bindings implementations of WSDL and SOAP, including Gosu (built into BillingCenter), Java, Perl, and other languages.

Publish or Consume Web Services from Gosu

Gosu natively supports web services in two ways:

- **Publish your Gosu code as new web service APIs** – Write Gosu code that external systems call as a web service using the SOAP protocol. Simply add a single line of code before the definition of a Gosu class.
- **Call web service APIs that external applications publish from your Gosu code** – Write Gosu code that imports and calls web service APIs published by external systems. Gosu parses the WSDL for the service. Gosu uses the WSDL to create Gosu types that enable you to call the remote API. You can call methods on the API and access types from the WSDL, all with a natural Gosu syntax.

Finding the Best Web Service Documentation for Your Needs

BillingCenter supports WS-I web services. WS-I web services use the SOAP protocol and are compatible with the WS-I standard. For an overview of web services, including a list of all built-in web services, see “Web Services Introduction” on page 27.

| Topic | See |
|--|---|
| Overview of web services | “Web Services Introduction” on page 27 |
| Reference of all built-in web services | “Reference of All Built-in Web Services” on page 29 |
| Publishing a web service on the BillingCenter server | “Publishing Web Services” on page 31 |
| Consuming a web service | “Calling Web Services from Gosu” on page 65 |

What Happens During a Web Service Call?

For all types of web services, BillingCenter converts the server’s local Gosu objects to and from the flattened text-based format that the SOAP protocol requires. These processes are called *serialization* and *deserialization*.

For example:

- Suppose you write a web service in Gosu and publish it from BillingCenter and call it from a remote system. Gosu must deserialize the text-based request into a local object that your Gosu code can access. If one of your web service methods returns a result, BillingCenter serializes that local in-memory Gosu result object. BillingCenter serializes it into a text-based reply to the remote system.
- Suppose you use Gosu to call a web service hosted by an external system. Before calling the API, Gosu automatically serializes any API parameters to convert a local object into a flattened form to send to the API. If the remote API returns a result, BillingCenter deserializes the response into local Gosu objects for your code to examine.

Writing your own custom web service for each integration point is the best approach for maximum performance and maintainability. Guidewire strongly encourages you to write as many web services as necessary to elegantly provide APIs for each integration point.

For example, write new web services to communicate with a check printing service, a legacy financials system, reporting service, or document management system. External systems can query BillingCenter to calculate values, trigger actions, or to change data within the BillingCenter database.

Publishing a web service can be as simple as adding one special line of code called an *annotation* immediately before your Gosu class.

For consuming a web service, Gosu creates local objects in memory that represent the remote API. Gosu creates types for every object in the WSDL, and you can create these objects or manipulate properties on them.

See also

- For more details about publishing or consuming web services, see “Finding the Best Web Service Documentation for Your Needs” on page 28.

Reference of All Built-in Web Services

The following table lists all built-in web services.

| Web Service Name | Description |
|---------------------------------|---|
| Application web services | |
| BillingAPI | Performs various actions requested by external policy administration systems, such as creating billing instructions. If you configure PolicyCenter to connect to BillingCenter, PolicyCenter uses this web service automatically. See “Policy Administration System Core Web Service APIs (BillingAPI)” on page 85. |
| InvoiceDetailsAPI | Manipulates invoices. See “Invoice Details Web Service APIs (InvoiceDetailsAPI)” on page 106. |
| PaymentAPI | Makes payments, reversals, and adjustments. See “Payments Web Service APIs (PaymentAPI)” on page 107. |
| PaymentInstrumentAPI | Creates and retrieves payment instruments, such as credit cards, for accounts and producers, stored in an external and secure repository separate from BillingCenter. See “Payment Instrument Web Service APIs (PaymentInstrumentAPI)” on page 112. |
| BillingSummaryAPI | Retrieves billing summaries for accounts and policies, periods billed to accounts and policies, and invoices on an account. See “Billing Summary Web Service APIs (BillingSummaryAPI)” on page 113. |
| TroubleTicketAPI | Creates trouble tickets based on postal codes. See “Trouble Ticket Web Service APIs (TroubleTicketAPI)” on page 114. |
| BCAPI | Various actions such as manipulating account objects (such as notes, documents, activities) and changing billing methods. See “Other Billing Web Service APIs (BCAPI)” on page 115. |
| ContactAPI | Manipulates contacts. See “Contact Web Service APIs” on page 401. |
| General web services | |
| TypeListToolsAPI | Retrieves aliases for BillingCenter typecodes in external systems. See “Mapping Typecodes to External System Codes” on page 125. |
| ImportTools | Imports administrative data from an XML file. You must use this only with administrative database tables (entities such as User). This system does not perform complete data validation tests on any other type of imported data. See “Importing Administrative Data” on page 127. |
| MaintenanceToolsAPI | Starts and manages various background processes. The methods of this web service are available only when the server run level is maintenance or higher. See “Maintenance Tools Web Service” on page 128. |
| SystemToolsAPI | Performs various actions related to server run levels, schema and database consistency, module consistency, and server and schema versions. The methods of this web service are available regardless of the server run level. See “System Tools Web Services” on page 129. |
| WorkflowAPI | Performs various actions on a workflow, such as suspending and resuming workflows and invoking workflow triggers. See “Workflow Web Services” on page 130. |
| ProfilerAPI | Sends information to the built-in system profiler. See “Profiling Web Services” on page 131. |
| MessagingToolsAPI | Manages the messaging system remotely for message acknowledgements error recovery. The methods of this web service are available only when the server run level is multiuser. See “Web Services for Handling Messaging Errors” on page 360. |
| DataChangeAPI | Tool for rare cases of mission-critical data updates on running production systems. See “Data Change API Overview” on page 47 in the <i>System Administration Guide</i> . |

| Web Service Name | Description |
|------------------|---|
| TableImportAPI | Loads operational data from staging tables into operational tables. Typically you use this web service for large-scale data conversions, such as migrating accounts from a legacy system to BillingCenter prior to bringing BillingCenter into production. The methods of this web service are available only when the server run level is maintenance. See "Importing from Database Staging Tables" on page 367. |
| ZoneImportAPI | Imports geographic zone data from a comma separated value (CSV) file into a staging table, in preparation for loading zone data into the operational table. See "Importing from Database Staging Tables" on page 367. |
| LoginAPI | The WS-I web service LoginAPI is not used for authentication in typical use. It is only used to test authentication or force a server session for logging. For details, see "Login Authentication Confirmation" on page 58. |

Publishing Web Services

You can write web service APIs in Gosu and access them from remote systems using the standard web services protocol *SOAP*. The SOAP protocol defines request/response mechanisms for translating a function call and its response into XML-based messages typically sent across computer networks over the standard HTTP protocol. Web services provide a language-neutral and platform-neutral mechanism for invoking actions or requesting data from another application across a network. BillingCenter publishes its own built-in web service APIs that you can use.

This topic includes:

- “Web Service Publishing Overview” on page 32
- “Publishing and Configuring a Web Service” on page 36
- “Testing Web Services with Local WSDL” on page 41
- “Generating WSDL” on page 42
- “Adding Advanced Security Layers to a Web Service” on page 45
- “Web Services Authentication Plugin” on page 50
- “Checking for Duplicate External Transaction IDs” on page 51
- “Request or Response XML Structural Transformations” on page 51
- “Reference Additional Schemas in Your Published WSDL” on page 52
- “Validate Requests Using Additional Schemas as Parse Options” on page 52
- “Invocation Handlers for Implementing Preexisting WSDL” on page 53
- “Locale Support” on page 56
- “Setting Response Serialization Options, Including Encodings” on page 57
- “Exposing Typelists and Enumerations as String Values” on page 57
- “Transforming a Generated Schema” on page 58
- “Login Authentication Confirmation” on page 58
- “Stateful Session Affinity Using Cookies” on page 59
- “Calling a BillingCenter Web Service from Java” on page 59

See also

- For information about the WS-I standard and its Document Literal encoding, see “Calling Web Services from Gosu” on page 65.
- For information about PolicyCenter integration with BillingCenter, see the topic “Billing Integration” in the *PolicyCenter Integration Guide*. The integration information is in the PolicyCenter documentation set, not the BillingCenter documentation set.

Web Service Publishing Overview

Web services define request-and-response APIs that let you call an API on a remote computer, or even the current computer, using an abstracted well-defined interface. A data format called the Web Service Description Language (WSDL) describes available web services that other systems can call using the SOAP protocol. Many languages or third-party packages provide bindings implementations of WSDL and SOAP, including Gosu (built into BillingCenter), Java, Perl, and other languages.

Gosu natively supports web services in two different ways:

- **Publish your Gosu code as new web service APIs** – Write Gosu code that external systems call as a web service using the SOAP protocol. Simply add a single line of code before the definition of a Gosu class.
- **Call web service APIs that external applications publish from your Gosu code** – Write Gosu code that imports and calls web service APIs published by external systems. Gosu parses the WSDL for the service. Gosu uses the WSDL to create Gosu types that enable you to call the remote API. You can call methods on the API and access types from the WSDL, all with a natural Gosu syntax. For more information, see “Calling Web Services from Gosu” on page 65.

Designing Your Web Services

In both cases BillingCenter converts the server’s local Gosu objects to and from the flattened text-based format required by the SOAP protocol. This process is *serialization* and *deserialization*.

- Suppose you write a web service in Gosu and publish it from BillingCenter and call it from a remote system. Gosu must deserialize the text-based XML request into a local object that your Gosu code can access. If one of your web service methods returns a value, BillingCenter serializes that local in-memory Gosu object and serializes it into a text-based XML reply for the remote system.
- Suppose you use Gosu to call a web service hosted by an external system. Before calling the API, Gosu automatically serializes any API parameters to convert a local object into a flattened text-based XML format to send to the API. If the remote API returns a result, BillingCenter deserializes the XML response into local Gosu objects for your code to use.

Guidewire provides built-in web service APIs for common general tasks and tasks for business entities of BillingCenter. For a full list, see “Reference of All Built-in Web Services” on page 29. However, writing your own custom web service for each integration point is the best approach for maximum performance and maintainability. Guidewire strongly encourages you to write as many web services as necessary to elegantly provide APIs for each integration point.

For example, write new web services to communicate with a check printing service, a legacy financials system, reporting service, or document management system. External systems can query BillingCenter to calculate values, trigger actions, or to change data within the BillingCenter database.

Publishing a web service may be as simple as added one special line of code called an annotation immediately before your Gosu class. Additional customizations may require extra annotations. See “Publishing and Configuring a Web Service” on page 36 for details.

There are special additional tasks or design decisions that affect how you write your web services, for example:

- **Create custom structures to send only the subset of data you need** – For large Guidewire business data objects (entities), most integration points only need to transfer a subset of the properties and object graph. Do not pass large object graphs, and be aware of any objects that might be very large in your real-world deployed production system. In such cases, you must design your web services to pass your own objects containing only your necessary properties for that integration point, rather than pass the entire entity. For example, if an integration point only needs a record's main contact name and phone number, create a shell object containing only those properties and the standard public ID property. For details, see “Publishing and Configuring a Web Service” on page 36.

Web services published by Guidewire applications are forbidden to use a Guidewire entity type as an argument to a method or a return type. Instead, create other kinds of objects that store the important data for that integration point.

IMPORTANT WS-I web services cannot have arguments or return types that directly or indirectly access Guidewire entity types.

For example, you could change argument and return types to the following:

- **Gosu class instances** – Write Gosu classes that include the important properties for that integration point. For example, a check printing service might only need the amount and the mailing address, but not any associated metadata or notes about the check. To work with web services, you must add the `final` keyword on the class, and also add the `@WsExportable` annotation on the class. For example:

```
package example.wsi.myobjects
uses gw.xml.ws.annotation.WsExportable

@WsExportable
final class MyCheckPrintInfo {
    var checkID : String
    var checkRecipientDisplayName : String
}
```

- **XML types created with the Guidewire XML modeler** – Use the Guidewire XML modeler tool (also called the *GX modeler*) to generate a model that contains only the desired subset of properties for each entity for each integration point. Then you can import or export XML data using that GX modeler as needed. See “The Guidewire XML (GX) Modeler” on page 310 in the *Gosu Reference Guide*.
- **Be careful of big objects** – If your data set is particularly large, it may be too big to pass over the SOAP protocol in one request. You may need to refactor your code to accommodate smaller requests. If you try to pass too much data over the SOAP protocol in either direction, there can be memory problems that you must avoid in production systems.
- **Design your web service to be testable** – For an example of testing a web service, see “Testing Web Services with Local WSDL” on page 41.
- **Learn bundle and transaction APIs** – To change and commit entity data in the database, learn the bundle APIs. You do not need to use bundle APIs for SOAP APIs that simply get and return unchanged data. See “Committing Entity Data to the Database Using a Bundle” on page 34.

Guidewire provides a Java language binding for published web services. Using these bindings, you can call the published web services from Java program as easy as making a local method invocation. You can use other programming languages if it has a SOAP implementation and can access the web service over the Internet.

Because Java is the typical language for web service client code, this topic usually uses Java syntax and terminology to demonstrate APIs. In some cases, this topic uses Gosu to demonstrate examples as it relates to publishing or testing web services.

If you write APIs to integrate two or more Guidewire applications, you probably will write your web service code in Gosu using its native SOAP syntax rather than Java. See “Calling Web Services from Gosu” on page 65.

WARNING Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. Be careful about any SOAP calls to the same server.

Committing Entity Data to the Database Using a Bundle

There is no default writable bundle for web services. Bundles are the way that Guidewire applications track changes to entity data. You must define your own new writable bundle if your web service must change any entity data.

To create a new writable bundle, use the `runWithNewBundle` API. See “Running Code in an Entirely New Bundle” on page 351 in the *Gosu Reference Guide*.

Web services that only get entity data (never modify entity data) never need to create a new bundle.

Serializable Gosu Classes Must Be Final and Exportable

Web services can contain arguments and return values that contain or reference regular instances of Gosu classes, sometimes called Plain Old Gosu Objects (POGOs).

However, the Gosu class must have the following two special qualities:

- The class must be declared as `final`, which as a consequence means it has no subclasses.
- The class must have the annotation `@WsIExportable` or Gosu does not publish the service.

Web Service Publishing Quick Reference

The following table summarizes important qualities of WS-I web services.

| Feature | WS-I web service behavior |
|---|--|
| Basic publishing of a WS-I web service | Add the annotation <code>@WsIWebService</code> to the implementation class. This annotation supports one optional argument for the web service namespace. See “Declaring the Namespace for a Web Service” on page 37. If you do not declare the namespace, Gosu uses the default namespace <code>http://example.com</code> . |
| Does BillingCenter automatically generate WSDL files from a running server? | Yes. See “Getting WSDL from a Running Server” on page 42. |
| Does BillingCenter automatically generate WSDL files locally if you regenerate the SOAP API files? | Yes. See “Generating WSDL” on page 42. |
| Does BillingCenter automatically generate JAR files for Java SOAP client code if you regenerate the SOAP API files? | No, but it is easy to generate with the Java built-in utility <code>wsimport</code> . The documentation includes examples. See “Calling a BillingCenter Web Service from Java” on page 59. |
| Can serialize Gosu class instances, sometimes called POGOs: Plain Old Gosu Objects? | Yes, however the Gosu class must have two special qualities. See “Serializable Gosu Classes Must Be Final and Exportable” on page 34. |
| Can web services serialize or deserialize Guidewire entity instances? | No. To transfer entity data, create data transfer objects (DTOs) as that contain only the data you need. DTO objects can be either Gosu class instances or XML objects. See later in this table for “Can serialize Gosu class instances”. |
| Can serialize a XSD-based type | Yes, using the <code>XmlElement</code> APIs. |
| Can you use the Guidewire XML Modeler tool (GX Model) to generate XML types that can be WS-I arguments or return types? | Yes |
| Can it serialize a Java object as an argument type or return type? | No |
| Automatically throw <code>SOAPException</code> exceptions? | No. Declare the actual exceptions you want thrown. In general this requirement reduces typical WSDL size. |
| Bundle handling for changing entity data? | A <i>bundle</i> is a container for entities that helps BillingCenter track what changed in a database transaction. There is no default bundle for WS-I web services. See “Committing Entity Data to the Database Using a Bundle” on page 34. |

| Feature | WS-I web service behavior |
|---|---|
| Logging in to the server | Each WS-I request embeds necessary authentication and security information in each request. If you use Guidewire authentication, you must add the appropriate SOAP headers before making the connection. |
| Package name of web services from the SOAP API client perspective | The package in which you put your web service implementation class defines the package for the web service from the SOAP API client perspective. The biggest benefit from this change is reducing the chance of namespace collisions in general. In addition, the web service namespace URL helps prevent namespace collisions. See "Declaring the Namespace for a Web Service" on page 37. |
| Calling a local version of the web service from Gosu | Gosu creates types that use the original package name to avoid namespace collisions. API references in the package <code>wsi.local.ORIGINAL_PACKAGE_NAME</code> You must call the command line command to rebuild the local files: <code>BillingCenter/bin/gwbc regen-wsi-local</code> |

Web Service Publishing Annotation Reference

The following table lists annotations related to WS-I web service declaration and configuration:

| Annotation | Description | For more information |
|---|---|--|
| Web service implementation class annotations | | |
| <code>@WsiWebService</code> | Declare a class as implementing a WS-I web service | "Publishing and Configuring a Web Service" on page 36 |
| <code>@WsiAvailability</code> | Set the minimum server run level for this service | "Specifying Minimum Run Level for a Web Service" on page 37 |
| <code>@WsiPermissions</code> | Set the required permissions for this service | "Specifying Required Permissions for a Web Service" on page 38 |
| <code>@WsiExposeEnumAsString</code> | Instead of exposing typelist types and enumerations as enumerations in the WSDL, you can expose them as <code>String</code> values. | "Exposing Typelists and Enumerations as String Values" on page 57 |
| <code>@WsiWebMethod</code> | The <code>@WsiWebMethod</code> annotation can do two things: <ul style="list-style-type: none"> Override the web service operation name to something other than the default, which is the method name. Suppress a method from generating a web service operation even though the method has <code>public</code> access. | "Overriding a Web Service Method Name or Visibility" on page 38 |
| <code>@WsiSerializationOptions</code> | Add serialization options for web service responses, for example supporting encodings other than UTF-8. | "Setting Response Serialization Options, Including Encodings" on page 57 |
| <code>@WsiAdditionalSchemas</code> | Expose additional schemas to web service clients in the WSDL. Use this to provide references to schemas that might be required but are not automatically included. | "Reference Additional Schemas in Your Published WSDL" on page 52 |
| <code>@WsiParseOptions</code> | Add validation of incoming requests using additional schemas in addition to the automatically generated schemas. | "Validate Requests Using Additional Schemas as Parse Options" on page 52 |

| Annotation | Description | For more information |
|---|---|---|
| @WsiRequestTransform @WsiResponseTransform | Add transformations of incoming or outgoing data as a byte stream. Typically you would use this to add advanced layers of authentication or encryption. Contrast to the annotations in the next row, which operate on XML elements. | "Data Stream Transformations" on page 45 |
| @WsiRequestXml1Transform @WsiResponseXml1Transform | Add transformations of incoming or outgoing data as XML elements. Use this for transformations that do not require access to byte data. Contrast to the annotations in the previous row, which operate on a byte stream. | "Request or Response XML Structural Transformations" on page 51 |
| @WsiSchemaTransform | Transform the generated schema that BillingCenter publishes. | "Transforming a Generated Schema" on page 58 |
| @WsiInvocationHandler | Perform advanced implementation of a web service that conforms to externally-defined standard WSDL. This is for unusual situations only. This approach limits protection against some types of common errors. | "Invocation Handlers for Implementing Preexisting WSDL" on page 53 |
| @WsiCheckDuplicateExternalTransaction | Detect duplicate operations from external systems that change data | "Checking for Duplicate External Transaction IDs" on page 51 |
| Other annotations to support serialization | | |
| @WsiExportable | Add this annotation on a Gosu class to indicate that it supports serialization with WS-I web services. You must additionally add the <code>final</code> keyword to the class to prevent subclasses. | "Serializable Gosu Classes Must Be Final and Exportable" on page 34 |

Publishing and Configuring a Web Service

To publish a WS-I web service, use the `@WsiWebService` annotation on a class. Gosu publishes that class automatically when the server runs.

For example:

```
package example
uses gw.xml.ws.annotation.WsiWebService

@WsiWebService
class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }

}
```

Choose your package declaration carefully. The package in which you put your web service implementation class defines the package for the web service from the SOAP API client perspective. This reduces the chance of namespace collisions.

Declaring the Namespace for a Web Service

Each web service is defined within a *namespace*, which is similar to how a Gosu class or Java class is within a package. Specify the namespace as a URL in each web service declaration. The namespace is a `String` that looks like a standard HTTP URL but represents a namespace for all objects in the service's WSDL definition. The namespace is not typically a URL for an actual downloadable resource. Instead it is an abstract identifier that disambiguates objects in this service from objects in another service.

A typical namespace specifies your company and perhaps other meaningful disambiguating or grouping information about the purpose of the service, possibly including a version number. For example:

- "http://mycompany.com"
- "http://mycompany.com/xsds/messaging/v1"

You can specify the namespace for each web service by passing a namespace as a `String` argument to the `@WsiWebService` annotation.

For example:

```
package example
uses gw.xml.ws.annotation.WsiWebService

@WsiWebService("http://mycompany.com")
class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }

}
```

You can omit the namespace declaration entirely and provide no arguments to the annotation. If you omit the namespace declaration in the constructor, the default is "example.com".

Most tools that create stub classes for web services use the namespace to generate a package namespace for related types. For examples of how the Java `wsimport` tool uses the namespace, see "Calling a BillingCenter Web Service from Java" on page 59.

Specifying Minimum Run Level for a Web Service

To set the minimum run level for the service, add the annotation `@WsiAvailability` and pass the run level at which this web service is first usable. The choices include DAEMONS, MAINTENANCE, MULTIUSER, and STARTING. The default is NODAEMONS.

For example:

```
package example
uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.

@WsiAvailability(MAINTENANCE)
@WsiWebService
class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }

}
```

Specifying Required Permissions for a Web Service

To add required permissions, add the annotation `@WsiPermissions` and pass an array of permission types (`SystemPermissionType[]`). The default permission is SOAPADMIN. If you provide an explicit list of permissions, you can choose to include SOAPADMIN explicitly or omit it. If you omit SOAPADMIN from our list of permissions, your web service does not require SOAPADMIN permission. If you pass an empty list of permissions, your web service does not require authentication at all.

The only supported permission types are permissions that are role-based (static), rather than data-based (requires an object). In the *Security Dictionary*, view the desired permission. If the permission is role-based, the page says after the permission name the word "(static)".

The following example removes authentication for a simple service you might use for debugging:

```

package example

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiPermissions

@WsiPermissions({}) // A blank list means no authentication needed.
@WsiWebService

class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }
}

```

Overriding a Web Service Method Name or Visibility

You can override the names and visibility of individual web service methods in several ways.

Overriding Web Service Method Names

By default, the web service operation name for a method is simply the name of the method. You can override the name by adding the `@WsiWebMethod` annotation immediately before the declaration for the method on the implementation class. Pass a `String` value for the new name for the operation. For example:

```

@WsiWebMethod("newMethodName")
public function helloWorld() : String {
    return "Hello!"
}

```

Hiding Public Web Service Methods

By default, Gosu publishes one web service operation for each method marked with `public` availability. For `public` methods, you can exclude the method from the web service by adding the `@WsiWebMethod` annotation immediately before the declaration for the method on the implementation class. Pass the value `true` to exclude (suppress) publishing that public method. For example:

```

@WsiWebMethod(true)
public function helloWorld() : String {
    return "Hello!"
}

```

If you pass the value `false` instead of `true` to the `@WsiWebMethod` annotation, there is no different behavior compared to omitting the annotation. If you use this annotation on a non-public method, the exclusion behavior has no effect. It never forces a non-public method to be visible on the web service.

Overriding Method Names and Visibility with a Single `@WsiWebMethod` Annotation

You can combine both of these features of the `@WsiWebMethod` annotation by passing both arguments. For example:

```

@WsiWebMethod("newMethodName", true)
public function helloWorld() : String {
    return "Hello!"
}

```

Web Service Invocation Context

In some cases your web service may need additional context for incoming or outgoing information. For example, to get or set HTTP or SOAP headers. You can access this information in your implementation class by adding an additional method argument to your class of type `WsiInvocationContext`. Make this new object the last argument in the argument list.

Unlike typical method arguments on your implementation class, this method parameter does not become part of the operation definition in the WSDL. Your web service code can use the `WsiInvocationContext` object to get or set important information as needed.

The following table lists properties on `WsiInvocationContext` objects and whether each property is writable.

| Property | Description | Readable | Writable |
|---------------------------------------|--|----------|--|
| Request Properties | | | |
| <code>HttpServletRequest</code> | A servlet request of type <code>HttpServletRequest</code> | Yes | No |
| <code>MtomEnabled</code> | <p>A boolean value that specifies whether to optionally support sending MTOM attachments to the WS-I web service client in any data returned from an API. The W3C Message Transmission Optimization Mechanism (MTOM) is a method of efficiently sending binary data to and from web services as attachments outside the normal response body.</p> <p>If <code>MtomEnabled</code> is <code>true</code>, BillingCenter can send MTOM in results. Otherwise, MTOM is disabled. The default is <code>false</code>.</p> <p>This property does not affect MTOM data sent to a published web service. Incoming MTOM data is always supported.</p> <p>This property does not affect MTOM support where Gosu is the client to the web service request. See "MTOM Attachments with Gosu as Web Service Client" on page 81.</p> | Yes | Yes |
| <code>RequestHttpHeaders</code> | Request headers of type <code>HttpHeaders</code> | Yes | No |
| <code>RequestEnvelope</code> | Request envelope of type <code>Xmlelement</code> | Yes | No |
| <code>RequestSoapHeaders</code> | Request SOAP headers of type <code>Xmlelement</code> | Yes | No |
| Response Properties | | | |
| <code>ResponseHttpHeaders</code> | Response HTTP headers of type <code>HttpHeaders</code> | Yes | The property value is read-only, but you can modify the object it references |
| <code>ResponseSoapHeaders</code> | Response SOAP headers of type <code>List<Xmlelement></code> | Yes | The property value is read-only, but you can modify the object it references |
| Output serialization | | | |
| <code>Xmleserializationoptions</code> | <p>XML serialization options of type <code>Xmleserializationoptions</code>. For more information on XML serialization, including the properties of <code>Xmleserializationoptions</code> objects, see "Exporting XML Data" on page 283 in the <i>Gosu Reference Guide</i>.</p> | Yes | Yes |

You can use these APIs to modify response headers from your web service and read request headers as needed.

For example, suppose you want to copy a specific request SOAP header XML object and copy to the response SOAP header object. Create a private method to get a request SOAP header XML object:

```
private function getHeader(name : QName, context : WsiInvocationContext) : Xmlelement {
    for (h in context.RequestSoapHeaders.Children) {
        if (h.QName.equals(name))
            return h;
    }
    return null
}
```

From a web service operation method, declare the invocation context optional argument to your implementation class. Then you can use code with the invocation context such as the following to get the incoming request header and add it to the response headers:

```
function doWork(..., context : WsiInvocationContext) {
    var rtnHeader = getHeader(TURNAROUND_HEADER_QNAME, context)
    context.ResponseSoapHeaders.add(rtnHeader)

    // do your main work of your web service operation
}
```

Web Service Class Lifecycle and Request Local Scoped Variables

BillingCenter does not instantiate the web service implementation class on server startup. BillingCenter instantiates the web service implementation class upon the first request from an external web service client for that specific service. That one object instance is shared across all requests and sessions on that server for the lifetime of the BillingCenter application.

Because multiple threads may share this object, you must be careful with use of static variables or instance variables. You must use concurrency APIs to ensure thread safety. For most Gosu coding, you can use the standard Gosu concurrency classes `RequestVar` and `SessionVar` in package `gw.api.web` package. However, these classes do not work in web service implementation classes.

Instead, for web service request-based data storage, use the concurrency class `gw.xml.ws.WsiRequestLocal`. If you use this for web service implementation class instance variable, the object exists only once. However, the object has `get` and `set` methods that manage the variable life cycle for each request in a thread-safe way. In this context, a request refers to invocation of one web service operation (one method of your web service).

To define a web request local scoped variable

1. Decide what type of object you want to store in your request local variable. In your type declaration for the variable, parameterize the `WsiRequestLocal` class with the type you want to store. For example, if you plan to store a `String` object, declare your variable to have the type `WsiRequestLocal<String>`.
2. In your web service implementation class, define a private variable using the parameterized type:

```
private var _reqLocal = new WsiRequestLocal<String>()
```

3. In your implementation class, use the `get` and `set` methods to get or set the variable. For example:

```
var loc = _reqLocal.get()
```

```
...
```

```
_reqLocal.set("the new value")
```

If you call `get` before calling `set` in the same web service request, the `get` method returns the value `null`.

Testing Web Services with Local WSDL

For testing purposes only, you can call WS-I web services published from the same BillingCenter server. To call a WS-I web service on the same server, you must generate WSDL files into the class file hierarchy so Gosu can access the service. This permits you to write unit tests that access the WSDL files over the SOAP protocol from Gosu.

WARNING Guidewire does not support calls to SOAP APIs published on the same server in a production system. If you think you need to do so, please contact Customer Support.

To regenerate the WSDL for all local web services in the class hierarchy, open a command prompt and type:

```
BillingCenter/bin/gwbc regen-wsi-local
```

BillingCenter generates the WSDL in the following location in the class hierarchy. The WSDL is in the `wsi.local` package, followed by fully qualified name of the web service:

```
wsi.local.FQNAME.wsdl
```

To use the class, simply prefix the text `wsi.local.` (with a final period) to the fully-qualified name of your API implementation class.

For example, suppose the web service implementation class is:

```
mycompany.ws.v100.EchoAPI
```

The `regen-wsi-local` tool generates the following WSDL file:

```
BillingCenter/modules/configuration/gsrc/wsi/local/mycompany/ws/v100/EchoAPI.wsdl
```

Call this web service locally with the syntax:

```
var api = new wsi.local.mycompany.ws.v100.EchoAPI()
```

If you change the code for the web service and the change potentially changes the WSDL, regenerate the WSDL.

BillingCenter includes with WSDL for some local services in the default configuration. These WSDL files are in Studio modules other than `configuration`. If BillingCenter creates new local WSDL files from the `regen-wsi-local` tool, it creates new files in the `configuration` module.

WARNING The `wsi.local.*` namespace exists only to call web services from unit tests. It is unsafe to write production code that uses these `wsi.local.*` types.

To reduce the chance of accidental use of `wsi.local.*` types, Gosu prevents using these types in method signatures of published WS-I web services.

Writing Unit Tests for Your Web Service

It is good practice to design your web services to be testable. At the time you design your web service, think about the kinds of data and commands your service handles. Consider your assumptions about the arguments to your web service, what use cases your web service handles, and which use cases you want to test. Then, write a series of GUnit tests that use the `wsi.local.*` namespace for your web service.

For example, you created the following web service.

```
package example

uses gw.xml.ws.annotation.WsiWebService

@WsiWebService("http://mycompany.com")
class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }

}
```

The following sample Gosu code is a GUnit test of the preceding `HelloWorldAPI` web service.

```
package example

uses gw.testharness.TestBase

class HelloWorldAPITest extends gw.testharness.TestBase {

    construct() {

    }

    public function testMyAPI() {
        var api = new wsi.local.example.helloworldapi.HelloWorldAPI()

        api.Config.Guidewire.Authentication.Username = "su"
        api.Config.Guidewire.Authentication.Password = "gw"
    }
}
```

```
var res = api.helloWorld();
print("result is: " + res);
TestBase.assertEquals("Expected 'Hello!'", "Hello!", res)
print("we got to the end of the test without exceptions!")
}

}
```

For more thorough testing, test your web service from integration code on an external system. To assure your web service scales adequately, test your web service with as large a data set and as many objects as potentially exist in your production system. To assure the correctness of database transactions, test your web service to exercise all bundle-related code.

Generating WSDL

Getting WSDL from a Running Server

Typically, you get the WSDL for a WS-I web service from a running server over the network. This approach encourages all callers of the web services to use the most current WSDL. In contrast, generating the WSDL and copying the files around risks callers of the web service using outdated WSDLs. You can get the most current WSDL for a web service from any computer on your network that publishes the web service. In a production system, code that calls a web service typically resides on computers that are separate from the computer that publishes the web service.

BillingCenter publishes a WS-I web service WSDL at the following URL:

`SERVER_URL/WEB_APP_NAME/ws/WEB_SERVICE_PACKAGE/WEB_SERVICE_CLASS_NAME?WSDL`

A published WSDL may make references to schemas at the following URL:

`SERVER_URL/WEB_APP_NAME/ws/SCHEMA_PACKAGE/SCHEMA_FILENAME`

For example, BillingCenter generates and publishes the WSDL for web service class `ws.eg.WebService.TestWsService` at the location on a server with web application name bc:

`http://localhost:PORTNUM/bc/ws/eg/webservice/TestWsService?WSDL`

Java includes a built-in command called `wsimport` that generates Java from the WSDL published on a running server. For more information, see “Calling a BillingCenter Web Service from Java” on page 59

WSDL and Schema Browser

If you publish web services from a Guidewire application, you can view the WSDL and a schema browser available at the following URL:

`SERVER_URL/WEB_APP_NAME/ws`

For example:

`http://localhost:8580/bc/ws`

From there, you can browse for:

- Document/Literal Web Services
- Supporting Schemas
- Generated Schemas
- Supporting WSDL files

Example WSDL

The following example demonstrates how a simple Gosu web service translates to WSDL. This example uses the following simple example web service.

```
package example

uses gw.xml.ws.annotation.WsiWebService

@WsiWebService
class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }
}
```

BillingCenter publishes the WSDL for the HelloWorldAPI web service at this location:

<http://localhost:PORTNUMBER/bc/ws/example>HelloWorldAPI?WSDL>

BillingCenter generates the following WSDL for the HelloWorldAPI web service.

```
<?xml version="1.0"?>
<!-- Generated WSDL for example.HelloWorldAPI web service -->
<wsdl:definitions targetNamespace="http://example.com/example/HelloworldAPI"
    name="HelloWorldAPI" xmlns="http://example.com/example/HelloworldAPI"
    xmlns:gw="http://guidewire.com/xsd" xmlns:soap11="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

    <wsdl:types>
        <xss:schema targetNamespace="http://example.com/example/HelloworldAPI"
            elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
            <!-- helloWorld() : java.lang.String -->
            <xss:element name="helloWorld">
                <xss:complexType/>
            </xss:element>
            <xss:element name="helloWorldResponse">

                <xss:complexType>
                    <xss:sequence>
                        <xss:element name="return" type="xs:string" minOccurs="0"/>
                    </xss:sequence>
                </xss:complexType>
            </xss:element>
        </xss:schema>
    </wsdl:types>
    <wsdl:portType name="HelloWorldAPIPortType">

        <wsdl:operation name="helloWorld">
            <wsdl:input name="helloWorld" message="helloWorld"/>
            <wsdl:output name="helloWorldResponse" message="helloWorldResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldAPISoap12Binding" type="HelloWorldAPIPortType">
        <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <wsdl:operation name="helloWorld">
            <soap12:operation style="document"/>

            <wsdl:input name="helloWorld">
                <soap12:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="helloWorldResponse">
                <soap12:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:binding name="HelloWorldAPISoap11Binding" type="HelloWorldAPIPortType">

        <soap11:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <wsdl:operation name="helloWorld">
            <soap11:operation style="document"/>
            <wsdl:input name="helloWorld">
                <soap11:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="helloWorldResponse">
                <soap11:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
```

```

<wsdl:service name="HelloWorldAPI">
  <wsdl:port name="HelloWorldAPISoap12Port" binding="HelloWorldAPISoap12Binding">
    <soap12:address location="http://localhost:8180/bc/ws/example/HelloWorldAPI"/>
    <gw:address location="${bc}/ws/example/HelloWorldAPI"/>
  </wsdl:port>
  <wsdl:port name="HelloWorldAPISoap11Port" binding="HelloWorldAPISoap11Binding">
    <soap11:address location="http://localhost:8180/bc/ws/example/HelloWorldAPI"/>
    <gw:address location="${bc}/ws/example/HelloWorldAPI"/>
  </wsdl:port>
</wsdl:service>
<wsdl:message name="helloWorld">
  <wsdl:part name="parameters" element="helloWorld"/>
</wsdl:message>
<wsdl:message name="helloWorldResponse">
  <wsdl:part name="parameters" element="helloWorldResponse"/>
</wsdl:message>
</wsdl:definitions>

```

The preceding generated WSDL defines multiple *ports* for this web service. A *port* in the context of a web service is unrelated to ports in the TCP/IP network transport protocol. Web service ports are alternative versions of a published web service. The preceding WSDL defines a SOAP 1.1 version and a SOAP 1.2 version of the HelloWorldAPI web service.

Generating WSDL On Disk

Typically, clients of a WS-I web service get the WSDL from the server where the web service runs to ensure that they use the current WSDL, not an outdated version. Also, you can get that WSDL from anywhere on your network. In a production system, your SOAP client code typically runs on computers that are separate from the computer where your SOAP server code runs.

However, there are special situations in which you might want to generate the WSDL file locally on the server. Locally generated WSDL files require a special annotation in your web service code and a special regeneration tool step.

Special Annotation to Generate WSDL On Disk

To generate WSDL for a WS-I web service locally on the server where the service runs, you must add the annotation `@WsiGenInToolkit` to the web service implementation class. Most WS-I web services included with BillingCenter do not have the annotation. If you want to generate the WSDL locally for these web services, modify the implementation file in Studio to add the `@WsiGenInToolkit` annotation.

For example, the following web service implementation class has the annotation before the class declaration:

```

package example

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiGenInToolkit

@WsiWebService
@WsiGenInToolkit

class HelloWorldAPI {

  public function helloWorld() : String {
    return "Hello!"
  }
}

```

Command Line tool to Generate WSDL On Disk

To generate the WSDL for all the WS-I web services with the annotation `@WsiGenInToolkit`, at a command prompt type the command:

```
BillingCenter/bin/gwbc regen-soap-api
```

Look for the locally generated WSDL files in the directory:

```
BillingCenter/soap-api/wsi/wsdl
```

Generating SOAP APIs for Use with Unit Tests From Gosu

To support writing Gosu unit tests (JUnit tests), BillingCenter can generate WSDL within the class file hierarchy for any published WS-I web services. This permits you to write unit tests that access the WSDL files over the SOAP protocol from Gosu. Other than for the purpose of unit tests, Guidewire does not support calls from a BillingCenter server into the same server over the SOAP protocol.

See also

- “Testing Web Services with Local WSDL” on page 41

Adding Advanced Security Layers to a Web Service

For security options beyond simple HTTP authentication and Guidewire authentication, you can use an additional set of APIs to implement additional layers of security. For example, you might want to add additional layers of encryption, digital signatures, or other types of authentication or security. From the SOAP server side, you add advanced security layers to outgoing requests by applying transformations to the data stream of the request.

Data Stream Transformations

Transformations on Data Streams

You can transform the data stream by processing it incrementally, byte by byte. For example, you can implement an encryption security layer by transforming the request data stream incrementally. Alternatively, you can transform the data stream by processing it as an entire unit at one time. For example, you must implement a digital signature authentication layer by transforming the entire request data stream at one time.

You can apply multiple types of transformations to the request data stream to add multiple security layers to your web service. The order of your transformations is important. For example, an encryption transformation followed by a digital signature transformation produces a different request data stream than a digital signature transformation followed by an encryption transformation.

If your desired transformation operates more naturally on XML elements and not a byte stream, instead consider using the APIs in “Request or Response XML Structural Transformations” on page 51

Accessing Data Streams

To access the data stream for a request, Gosu provides an annotation (`WsiRequestTransform`) to inspect or transform an incoming request data stream. Gosu provides another annotation (`WsiResponseTransform`) to inspect or transform an outgoing response data stream. Both annotations take a Gosu block that takes an input stream (`java.io.InputStream`) and returns another input stream. Gosu calls the annotation block for every request or response, depending on the type of annotation.

Example of Data Stream Request and Response Transformations

The following example implements a request transform and a response transform to apply a simple encryption security layer to a web service.

The example applies the same incremental transformation to the request and the response data streams. The transformation processes the data streams byte by byte to change the bits in each byte from a 1 to a 0 or a 0 to a 1. The example transformation code uses the Gosu bitwise exclusive OR (XOR) logical operator (\wedge) to perform the bitwise changes on each byte. The XOR logical operator is a *symmetrical* operation. If you apply XOR operation to a data stream and then apply the operation again to the transformed data stream, you obtain the original data stream. Therefore, transforming the incoming request data stream by using the XOR operation encrypts the data. Conversely, transforming the outgoing response data stream by using the XOR operation decrypts the data.

```
package gw.webservice.example

uses gw.util.StreamUtil
uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiRequestTransform
uses gw.xml.ws.annotation.WsiResponseTransform
uses java.io.ByteArrayInputStream
uses java.io.InputStream

@WsiWebService

// Specify data stream transformations for web service requests and responses.
@WsiRequestTransform(WsiTransformTestService._xorTransform)
@WsiResponseTransform(WsiTransformTestService._xorTransform)

class WsiTransformTestService {

    // Declare a static variable to hold a Gosu block that encrypts and decrypts data streams.
    public static var _xorTransform(inputStream : InputStream) : InputStream = \ inputStream ->{

        // Get the input stream, and store it in a byte array.
        var bytes = StreamUtil.getContent(inputStream)

        // Encrypt the bits in each byte.
        for (b in bytes index idx) {
            bytes[idx] = (b  $\wedge$  17) as byte // XOR encryption with "17" as the encryption mask
        }

        return new ByteArrayInputStream(bytes)
    }

    function add(a : int, b : int) : int {
        return a + b
    }

}
```

In the preceding example, the request transformation and the response transformation use the same Gosu block for transformation logic because the block uses a symmetrical algorithm. In a typical production scenario however, the request transformation and the response transformation use different Gosu blocks because their transformation logic differs.

See also

- “Using WSS4J for Encryption, Signatures, and Other Security Headers” on page 47
- “Bitwise Exclusive OR (\wedge)” on page 71 in the *Gosu Reference Guide*

Applying Multiple Security Layers to a Web Service

Whenever you apply multiple layers of security to your web service, the order of substeps in your request and response transformation blocks is critical. Typically, the order of substeps in your response block reverses the order of substeps in your request block. For example, if you encrypt and then add a digital signature to the response data stream, remove the digital signature before decrypting the request data stream. If you remove a security layers from your web service, assure the remaining layers preserve the correct order of substeps in the transformation blocks.

Using WSS4J for Encryption, Signatures, and Other Security Headers

The following example uses the Java utility WSS4J to implement encryption, digital cryptographic signatures, and other security elements (a timestamp). This example has three parts.

Part 1 of the Example that Uses WSS4J

The first part of the example is a utility class called `demo.DemoCrypto` that implements an input stream encryption routine that adds a timestamp, then a digital signature, then encryption. To decrypt the input stream for a request, the utility class knows how to decrypt the input stream and then validate the digital signature.

Early in the encryption (`addCrypto`) and decryption (`removeCrypto`) methods, the code parses, or inflates, the XML request and response input streams into hierarchical DOM trees that represent the XML. The methods call the internal class method `parseDOM` to parse input streams into DOM trees.

Parsing the input streams in DOM trees is an important step. Some of the encryption information, such as timestamps and digital signatures, must be added in a particular place in the SOAP envelope. At the end of the encryption and decryption methods, the code serializes, or flattens, the DOM trees back into XML request and response input streams. The methods call the internal class method `serializeDOM` to serialize DOM trees back into input streams.

```
package gw.webservice.example

uses gw.api.util.ConfigAccess
uses java.io.ByteArrayInputStream
uses java.io.ByteArrayOutputStream
uses java.io.InputStream
uses java.util.Vector
uses java.util.Properties
uses java.lang.RuntimeException
uses javax.xml.parsers.DocumentBuilderFactory
uses javax.xml.transform.TransformerFactory
uses javax.xml.transform.dom.DOMSource
uses javax.xml.transform.stream.StreamResult
uses org.apache.ws.security.message./*
uses org.apache.ws.security./*
uses org.apache.ws.security.handler./*

// Demonstration input stream encryption and decryption functions.
// The layers of security are a timestamp, a digital signature, and encryption.
class DemoCrypto {

    // Encrypt an input stream.
    static function addCrypto(inputStream : InputStream) : InputStream {
        var crypto = getCrypto()

        // Parse the input stream into a DOM (Document Object Model) tree.
        var domEnvironment = parseDOM(inputStream)

        var securityHeader = new WSSecHeader()
        securityHeader.insertSecurityHeader(domEnvironment);

        var timeStamp = new WSSecTimestamp();
        timeStamp.setTimeToLive(600)
        domEnvironment = timeStamp.build(domEnvironment, securityHeader)

        var signer = new WSSecSignature();
        signer.setUserInfo("ws-client", "client-password")
        var parts = new Vector()
        parts.add(new WSEncryptionPart("Timestamp",
            "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd",
            "Element"))
        parts.add(new WSEncryptionPart("Body",
            gw.xsd.w3c.soap12_envelope.Body.$QNAME.NamespaceURI, "Element"));
        signer.setParts(parts);
        domEnvironment = signer.build(domEnvironment, crypto, securityHeader);

        var encrypt = new WSSecEncrypt()
        encrypt.setUserInfo("ws-client", "client-password")
        // encryptionParts=
        // {Element}{http://www.w3.org/2000/09/xmldsig#}Signature;{Content}
        // {http://schemas.xmlsoap.org/soap/envelope/}Body
        parts = new Vector()
        parts.add(new WSEncryptionPart("Signature", "http://www.w3.org/2000/09/xmldsig#", "Element"))

    }
}
```

```

parts.add(new WSEncryptionPart("Body", gw.xsd.w3c.soap12_envelope.Body.$QNAME.NamespaceURI,
    "Content"));
encrypt.setParts(parts)
domEnvironment = encrypt.build(domEnvironment, crypto, securityHeader);

// Serialize the modified DOM tree back into an input stream.
return new ByteArrayInputStream(serializeDOM(domEnvironment.DocumentElement))
}

// Decrypt an input stream.
static function removeCrypto(inputStream : InputStream) : InputStream {

// Parse the input stream into a DOM (Document Object Model) tree.
var envelope = parseDOM(inputStream)

var secEngine = WSSecurityEngine.getInstance();
var crypto = getCrypto()
var results = secEngine.processSecurityHeader(envelope, null, \ callbacks ->{
    for (callback in callbacks) {
        if (callback typeis WSPasswordCallback) {
            callback.Password = "client-password"
        }
    }
    else {
        throw new RuntimeException("Expected instance of WSPasswordCallback")
    }
}, crypto);

for (result in results) {
    var eResult = result as WSSecurityEngineResult
    // Note: An encryption action does not have an associated principal.
    // Only Signature and UsernameToken actions return a principal
    if (eResult.Action != WSConstants.ENCR) {
        print(eResult.Principal.Name);
    }
}

// Serialize the modified DOM tree back into an input stream.
return new ByteArrayInputStream(serializeDOM(envelope.DocumentElement))
}

// Private function to create a map of WSS4J cryptographic properties.
private static function getCrypto() : org.apache.ws.security.components.crypto.Crypto {

    var cryptoProps = new Properties()
    cryptoProps.put("org.apache.ws.security.crypto.merlin.keystore.alias", "ws-client")
    cryptoProps.put("org.apache.ws.security.crypto.merlin.keystore.password", "client-password")
    cryptoProps.put("org.apache.ws.security.crypto.merlin.keystore.type", "jks")
    cryptoProps.put("org.apache.ws.security.crypto.merlin.file", ConfigAccess.getConfigFile(
        "config/etc/client-keystore.jks").CanonicalPath)
    cryptoProps.put("org.apache.ws.security.crypto.provider",
        "org.apache.ws.security.components.crypto.Merlin")

    return org.apache.ws.security.components.crypto.CryptoFactory.getInstance(cryptoProps)
}

// Private function to parse an input stream into a hierarchical DOM tree.
private static function parseDOM(inputStream : InputStream) : org.w3c.dom.Document {

    var factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);

    return factory.newDocumentBuilder().parse(inputStream);
}

// Private function to serialize a hierarchical DOM tree into an input stream.
private static function serializeDOM(element : org.w3c.dom.Element) : byte[] {

    var transformerFactory = TransformerFactory.newInstance();
    var transformer = transformerFactory.newTransformer();
    var baos = new ByteArrayOutputStream();
    transformer.transform(new DOMSource(element), new StreamResult(baos));

    return baos.toByteArray();
}

```

```
 }  
  
 }
```

Part 2 of the Example that Uses WSS4J

The next part of this example is a WS-I web service implementation class written in Gosu. The following sample Gosu code implements the web service in the class `demo.DemoService`.

```
package gw.webservice.example  
  
uses gw.xml.ws.annotation.WsiWebService  
uses gw.xml.ws.annotation.WsiRequestTransform  
uses gw.xml.ws.annotation.WsiResponseTransform  
uses gw.xml.ws.annotation.WsiPermissions  
uses gw.xml.ws.annotation.WsiAvailability  
  
@WsiWebService  
@WsiAvailability(NONE)  
@WsiPermissions({})  
@WsiRequestTransform(\ inputStream ->DemoCrypto.removeCrypto(inputStream))  
@WsiResponseTransform(\ inputStream ->DemoCrypto.addCrypto(inputStream))  
  
// This web service computes the sum of two integers. The web service decrypts incoming SOAP  
// requests and encrypts outgoing SOAP responses.  
class DemoService {  
  
    // Compute the sum of two integers.  
    function add(a : int, b : int) : int {  
        return a + b  
    }  
  
}
```

Some things to notice about the preceding web service implementation class:

- The web service itself simply adds two numbers, but the service has a request and response transformation.
- The request transformation removes and confirms the cryptographic layer on the request, including the digital signature and encryption. The request transformation calls `DemoCrypto.removeCrypto(inputStream)`.
- The response transformation adds the cryptographic layer on the response. The response transformation calls `DemoCrypto.addCrypto(inputStream)`

Part 3 of the Example that Uses WSS4J

The third and final part of this example is code to test this web service.

```
var webService = new wsi.local.demo.demoservice.DemoService()  
webService.Config.RequestTransform = \ inputStream ->demo.DemoCrypto.addCrypto(inputStream)  
webService.Config.ResponseTransform = \ inputStream ->demo.DemoCrypto.removeCrypto(inputStream)  
print(webService.add(3, 5))
```

Paste this code into the Gosu Scratchpad and run it.

See also

- “Testing Web Services with Local WSDL” on page 41

Web Services Authentication Plugin

To handle the name/password authentication for a user connecting to WS-I web services, BillingCenter delegates this job to the currently registered implementation of the `WebservicesAuthenticationPlugin` plugin interface. There must always be a registered version of this plugin, otherwise web services that require permissions cannot authenticate successfully.

The `WebservicesAuthenticationPlugin` plugin interface supports WS-I web service connections only.

The Default Web Services Authentication Plugin Implementation

To authenticate web service requests, it is common to set up non-human Guidewire application users through regular BillingCenter user administration. These non-human users never use their usernames and passwords to sign into the ClaimCenter user interface. Instead, an external system passes the username and password in the request headers of its WS-I web service requests to authenticate.

The default configuration of BillingCenter has a registered built-in implementation of WS-I web service authentication plugin. The class is called `gw.plugin.security.DefaultWebservicesAuthenticationPlugin`. In the default configuration of BillingCenter, this class does two things:

1. Looks at HTTP request headers for WS-I authentication information.
2. Performs authentication against the local BillingCenter users in the database. This class calls the registered implementation of the `AuthenticationServicePlugin` plugin interface.

IMPORTANT If you write your own implementation of the `AuthenticationServicePlugin` plugin interface, be aware of this interaction with WS-I web service authentication. For example, you might want LDAP authentication for most users, but for web service authentication to authenticate against the current BillingCenter application administrative data.

To authenticate only some user names to Guidewire application credentials, your `AuthenticationServicePlugin` code must check the user name and compare to a list of special web service user names. If the user name matches, do not use LDAP and instead authenticate with the local application administrative data. To do this in your `AuthenticationServicePlugin` implementation, use the code:

```
_handler.verifyInternalCredentials(username, password)
```

For details of authentication handlers and the `AuthenticationServicePlugin` plugin interface, see “User Authentication Service Plugin” on page 211.

Writing an Implementation of the Web Services Authentication Plugin

Most customers do not need to write a new implementation of the `WebservicesAuthenticationPlugin` plugin interface. Typical changes to WS-I authentication logic are instead in your `AuthenticationServicePlugin` plugin implementation. See related discussion in “The Default Web Services Authentication Plugin Implementation” on page 50.

You can change the default web services authentication behavior to get the credentials from different headers. You can write your own `WebservicesAuthenticationPlugin` plugin implementation to implement custom logic.

The `WebservicesAuthenticationPlugin` interface has a single plugin method that your implementation must implement, called `authenticate`. The `authenticate` method takes one parameter of type `gw.plugin.security.WebservicesAuthenticationContext`. The object passed to your `authenticate` method contains authentication information, such as the name and password.

Properties on a Web Services Authentication Context

Important properties on a `WebservicesAuthenticationContext` include:

- `HttpHeaders` – HTTP headers of type `gw.xml.ws.HttpHeaders`. This includes a list of header names
- `HttpServletRequest` – the HTTP servlet request object, as the standard Java object `javax.servlet.http.HttpServletRequest`.
- `RequestSOAPHeaders` – The request SOAP headers, as an XML element (`XmlElement`)

Values to Return from Your Default Web Services Authentication Plugin

The value that your implementation returns from its `authenticate` method depend on whether authentication succeeds:

- If authentication succeeds, return the relevant `User` object from the BillingCenter database.
- If you cannot attempt to authenticate for some reason, such as network problems, return `null`.
- If authentication fails for other reasons, throw an exception of type `WsiAuthenticationException`.

Checking for Duplicate External Transaction IDs

To detect duplicate operations from external systems that change data, add the `@WsiCheckDuplicateExternalTransaction` annotation to your WS-I web service implementation class. To apply this feature for all operations on the service, add the annotation at the class level. To apply only to some operations, declare the annotation at the method level for individual operations.

If you apply this feature to an operation (or to the whole class), BillingCenter checks for the SOAP header `<transaction_id>` in namespace `http://guidewire.com/ws/soapheaders`. If the header exists, the text data is the external transaction ID that uniquely identifies the transaction. The recommended pattern for the transaction ID is to begin with an identifier for the external system, then a colon, then an ID that is unique to that external system. The most important thing is that the transaction ID be unique across all external systems.

If the web service changes any database data, the application stores the transaction ID in an internal database table for future reference. If in the future, some code calls the web service again with the same transaction ID, the database commit fails and throws the following exception:

```
com.guidewire.pl.system.exception.DBAlreadyExecutedException
```

The caller of the web service can detect this exception to identify the request as a duplicate transaction.

Because this annotation relies on database transactions (bundles), if your web service does not change any database data, this API has no effect.

If your WS-I client code is written in Gosu, to set the SOAP header see “Setting Guidewire Transaction IDs” on page 75.

If you apply this feature to an operation (or to the whole class), and the SOAP header `<transaction_id>` is missing, BillingCenter throws an exception.

Request or Response XML Structural Transformations

For advanced layers of security, you probably want to use transformations that use the byte stream. See “Data Stream Transformations” on page 45.

However, there are other situations where you might want to transform either the request or response XML data at the structural level of manipulating `Xmlelement` objects.

To transform the request envelope XML before processing, add the `@WsiRequestXmlTransform` annotation. To transform the response envelope XML after generating it, add the `@WsiResponseXmlTransform` annotation.

Each annotation takes a single constructor argument which is a Gosu block. Pass a block that takes one argument, which is the envelope. The block transforms the XML element in place using the Gosu XML APIs.

The envelope reference statically has the type `Xmlelement`. However, at runtime the type is one of the following, depending on whether SOAP 1.1 or SOAP 1.2 invoked the service:

- `gw.xsd.w3c.soap11_envelope.Envelope`
- `gw.xsd.w3c.soap12_envelope.Envelope`

Reference Additional Schemas in Your Published WSDL

If you need to expose additional schemas to the web service clients in the WSDL, you can use the `@WsiAdditionalSchemas` annotation to do them. Use this to provide references to schemas that might be required but are not automatically included.

For example, you might define an operation to take any object in a special situation, but actually accepts only one of several different elements defined in other schemas. You might throw exceptions on any other types. By using this annotation, the web service can add specific new schemas so web service client code can access them from the WSDL for the service.

The annotation takes one argument of the type `List<XmlSchemaAccess>`, which means a list of schema access objects. To get a reference to a schema access object, first put a XSD in your class hierarchy. Then from Gosu, determine the fully-qualified name of the XSD based on where you put the XSD. Next, get the `util` property from the schema, and on the result get the `SchemaAccess` property. To generate a list, simply surround one or more items with curly braces and comma-separate the list.

For example the following annotation adds the XSD that resides locally in the location `gw.xml.ws.wsimyschema`:

```
@WsiAdditionalSchemas({ gw.xml.ws.wsimyschema.util.SchemaAccess })
```

See also

- “Reference of XSD Properties and Types” on page 290 in the *Gosu Reference Guide*
- “Referencing Additional Schemas During Parsing” on page 287 in the *Gosu Reference Guide*
- “Transforming a Generated Schema” on page 58

Validate Requests Using Additional Schemas as Parse Options

You can validate incoming requests using additional schemas. To add an additional schema parse option, add the `@WsiParseOptions` annotation to your web service implementation class.

Before proceeding, be sure you have a reference to the XSD-based schema. For an XSD or WSDL, get the `SchemaAccess` property on the XSD type to get the schema reference. The argument for the annotation is an instance of type `XmlParseOptions`, which contains a property called `AdditionalSchemas`. That property must contain a list of schemas.

To add a single schema, you can use the following compact syntax:

```
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { YOUR_XSD_TYPE.util.SchemaAccess } })
```

For an XSD called `WS.xsd` in the source code file tree in the package `com.abc`, use the following syntax:

```
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { com.abc.ws.util.SchemaAccess } })
```

To include an entire WSDL file as an XSD, use the same syntax. For example, if the file is `WS.wsdl`:

```
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { com.abc.ws.util.SchemaAccess } })
```

See also

- “Introduction to the XML Element in Gosu” on page 279 in the *Gosu Reference Guide*
- “Schema Access Type” on page 309 in the *Gosu Reference Guide*
- “Reference Additional Schemas in Your Published WSDL” on page 52

Invocation Handlers for Implementing Preexisting WSDL

The implementation of a Gosu WS-I web service is a Gosu class. Typically, each WS-I operation corresponds directly to one method on the Gosu class that implements the web service. Gosu automatically determines and generates the output WSDL for that operation. For any method arguments and return types, Gosu uses the class definition and the method signatures to determine what to export in the WSDL.

However, if necessary you can use argument types and return values defined in a separate WSDL. This might be necessary for example if you are required to implement a preexisting service definition. Perhaps ten different systems implement this service, and only one is defined by Gosu. By using a standardized WSDL, some organization can ensure that all types sent and received conform to a standard WSDL definition for the service.

Adding an Invocation Handler for Preexisting WSDL

Only use the `@WsiInvocationHandler` annotation if you need to write a web service that conforms to externally-defined standard WSDL. Generally speaking, using this approach makes your code harder to read code and error prone because mistakes are harder to catch at compile time. For example, it is harder to catch errors in return types using this approach.

To implement preexisting WSDL, you define your web service very differently than for typical web service implementation classes.

First, on your web service implementation class add the annotation `@WsiInvocationHandler`. As an argument to this annotation, pass an invocation handler. An invocation handler has the following qualities:

- The invocation handler is an instance of a class that extends the type
`gw.xml.ws.annotation.DefaultWsiInvocationHandler`.
- Implement the invocation handler as an inner class inside your web service implementation class.
- The invocation handler class overrides the `invoke` method with the following signature:
`override function invoke(requestElement : XmlElement, context : WsiInvocationContext) : XmlElement`
- The `invoke` method does the actual dispatch work of the web service for all operations on the web service. Gosu does not call any other methods on the web service implementation. Instead, the invocation handler handles all operations that normally would be in various methods on a typical web service implementation class.
- Your `invoke` method can call its super implementation to trigger standard method calls for each operation based on the name of the operation. Use this technique to run custom code before or after standard method invocation, either for logging or special logic.

In the `invoke` method of your invocation handler, determine which operation to handle by checking the type of the `requestElement` method parameter. For each operation, perform whatever logic makes sense for your web service. Return an object of the appropriate type. Get the type of the return object from the XSD-based types created from the WSDL.

Finally, for the WSDL for the service to generate successfully, add the preexisting WSDL to your web service using the WS-I parse options annotation `@WsiParseOptions`. Pass the entire WSDL as the schema as described in that topic.

See also

- “Introduction to the XML Element in Gosu” on page 279 in the *Gosu Reference Guide*
- “XSD-based Properties and Types” on page 289 in the *Gosu Reference Guide*
- “Validate Requests Using Additional Schemas as Parse Options” on page 52

Example of an Invocation Handler for Preexisting WSDL

In the following example, there is a WSDL file at the resource path in the source code tree at the path:

`BillingCenter/configuration/gsrc/ws/weather.wsdl`

The schema for this file has the Gosu syntax: `ws.weather.util.SchemaAccess`. Its element types are available in the Gosu type system as objects in the package `ws.weather.elements`.

The method signature of the `invoke` method returns an object of type `XmLElement`, the base class for all XML elements. Be sure to carefully create the right subtype of `XmLElement` that appropriately corresponds to the return type for every operation. See “Introduction to the XML Element in Gosu” on page 279 in the *Gosu Reference Guide*.

The following example implements a web service that conforms to a preexisting WSDL and implements one of its operations.

```
package gw.xml.ws

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiInvocationHandler
uses gw.xml.XmLElement
uses gw.xml.ws.annotation.WsiPermissions
uses gw.xml.ws.annotation.WsiAvailability
uses gw.xml.ws.annotation.WsiParseOptions
uses gw.xml.XmlParseOptions
uses java.lang.IllegalArgumentException

@WsiWebService("http://guidewire.com/px/ws/gw/xml/ws/WsiImplementExistingWsdlTestService")
@WsiPermissions({})
@WsiAvailability(NONE)
@WsiInvocationHandler(new WsiImplementExistingWsdlTestService.Handler())
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { ws.weather.util.SchemaAccess } })

class WsiImplementExistingWsdlTestService {

    // The following line declares an INNER CLASS within the outer class.
    static class Handler extends DefaultWsiInvocationHandler {

        // Here we implement the "weather" wsdl with our own GetCityForecastByZIP implementation.
        override function invoke(requestElement : XmLElement, context : WsiInvocationContext)
            : XmLElement {

            // Check the operation name. If it is GetCityForecastByZIP, handle that operation.
            if (requestElement typeis ws.weather.elements.GetCityForecastByZIP) {
                var returnResult = new ws.weather.elements.GetCityForecastByZIPResponse() {

                    // The next line uses type inference to instantiate XML object of the correct type
                    // rather than specifying it explicitly.
                    :GetCityForecastByZIPResult = new() {
                        :Success = true,
                        :City = "Demo city name for ZIP ${requestElement.ZIP}"
                    }
                    return returnResult
                }

                // Check for additional requestElement values to handle additional operations.
                if ...

                else {
                    throw new IllegalArgumentException("Unrecognized element: " + requestElement)
                }
            }
        }
    }
}
```

First, the `invoke` method checks if the requested operation is the desired operation. An operation normally corresponding to a method name on the web service, but in this approach one method handles all operations. In this simple example, the `invoke` method handles only the operation in the WSDL called `GetCityForecastByZip`. If the requested operation is `GetCityForecastByZip`, the code creates an instance of the `GetCityForecastByZIPResponse` XML element.

Next, the example uses Gosu object creation initialization syntax to set properties on the element as appropriate. Finally, the code returns that XML object to the caller as the result from the API call.

For additional context of the WSI request, use the `context` parameter to the `invoke` method. The `context` parameter has type `WsiInvocationContext`, which contains properties such as servlet and request headers.

See also

- “Web Service Invocation Context” on page 39

Invocation Handler Responsibilities

If you write an invocation handler for a WS-I web service, by default you are bypassing some important WS-I features:

- The application does not enable profiling for method calls.
- The application does not check run levels even at the web service class level.
- The application does not check web service permissions, even at the web service class level.
- The application does not check for duplicate external transaction IDs if present.

However, you can support all these things in your web service even when using an invocation handler, and in typical cases it is best to do so.

To re-enable bypassed WS-I features from an invocation handler

1. In your web service implementation class, create separate methods for each web service operation. For each method, for the one argument and the one return value, use an `XmLElement` object. For example, the method signature:

```
static function myMethod(req : XmLElement) : XmLElement
```

2. In your invocation handler's `invoke` method, determine which method to call based on the operation name, as documented earlier.
3. Get a reference to the method info meta data for the method you want to call, using the `#` symbol to access meta data of a feature (method or property):

```
var MethodInfo = YourClassName#myMethod(XmLElement).MethodInfo
```

4. Before calling your desired method, get a reference to the `WsiInvocationContext` object that is a method argument to `invoke`. Call its `preExecute` method, passing the `requestElement` and the method info metadata as arguments. If you do not require checking method-specific annotations for run level or permissions, for the method info metadata argument you can pass `null`.

The `preExecute` method does several things:

- Enables profiling for the method you are about to call if profiling is available and configured
- Checks the SOAP headers looking for headers that set the locale. If found, sets the specified locale.
- Checks the SOAP headers for a unique transaction ID. This ID is intended to prevent duplicate requests. If this transaction has already been processed successfully (a bundle was committed with the same ID), `preExecute` throws an exception. See “Checking for Duplicate External Transaction IDs” on page 51.
- If the method info argument is non-null, `preExecute` confirms the run level for this service, checking both the class level `@WsiAvailability` annotation and any overrides for the method. As with standard WS-I implementation classes, the method level annotation supersedes the class level annotation. If the run level is not at the required level, `preExecute` throws an exception.
- If the method info argument is non-null, `preExecute` confirms user permissions for this service, checking both the class level `@WsiPermissions` annotation and any overrides for the method. As with standard WS-I implementation classes, the method level annotation supersedes the class level annotation. If the permissions are not satisfied for the web service user, `preExecute` throws an exception.

5. Call your desired method from the invocation handler.

Assuming that you want to check the method-specific annotations for run level or permissions, one potential approach is to set up a map to store the method information. The map key is the operation name. The map value is the method info metadata required by the `preExecute` method.

The following example demonstrates this approach

```
@WsiInvocationHandler( new WsiImplementExistingWsdlTestService.Handler() )
class WsiImplementExistingWsdlTestService {

    // The following line declares an INNER class within the outer class.
    static class Handler extends DefaultWsiInvocationHandler {

        var _map = { "myOperationName1" -> WsiImplementExistingWsdlTestService#methodA(XmlElement).MethodInfo,
                    "myOperationName2" -> WsiImplementExistingWsdlTestService#methodB(XmlElement).MethodInfo
                }

        override function invoke( requestElement : XmlElement, context : WsiInvocationContext )
            : XmlElement {

            // get the operation name from the request element
            var opName = requestElement.QName

            // get the local part (short name) from operation name, and get the method info for it
            var method = _map.get(opName.LocalPart)

            // call preExecute to enable some WS-I features otherwise disabled in an invocation handler
            context.preExecute(requestElement, method)

            // call your method using the method info and return its result
            return method.CallHandler.handleCall(null, {requestElement}) as XmlElement
        }
    }

    // After defining your invocation handler inner class, define the methods that do your work
    // as separate static methods

    // example of overriding the default permissions
    @WsiPermissions( { /* add a list of permissions here */ } )
    static function methodA(req : XmlElement) : XmlElement {
        /* do whatever you do, and return the result */ return null
    }
    static function methodB(req : XmlElement) : XmlElement {
        /* do whatever you do, and return the result */ return null
    }
}
```

This is the only supported use of a `WsiInvocationContext` object's `preExecute` method. Any use other than calling it exactly once from within an invocation handler `invoke` method is unsupported.

Locale Support

By default, WS-I web services use the default server locale.

WS-I web service clients can override this behavior and set a locale to use while processing this web service request. To set the locale, the client can add a SOAP header element type `<gwsoap:locale>` with namespace "`http://guidewire.com/ws/soapheaders`", with the element containing the locale code.

For example, the following SOAP envelope contains a SOAP header that sets the `fr_FR` locale for French language in France:

```
<soap12:Envelope xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
<soap12:Header>
    <gwsoap:locale xmlns:gwsoap="http://guidewire.com/ws/soapheaders">fr_FR</gwsoap:locale>
</soap12:Header>
<soap12:Body>
    <getPaymentInstrumentsFor xmlns="http://example.com/gw/webservice/bc/bc700/PaymentAPI">
        <accountNumber>123</accountNumber>
    </getPaymentInstrumentsFor>
</soap12:Body>
</soap12:Envelope>
```

In this case, the web service would set the `fr_FR` locale before processing this web service request.

On a related topic, you can quickly configure the locale in a web service client in Gosu, such as from another Guidewire application. See “Setting Locale in a Guidewire Application” on page 78. That configuration information sets the SOAP header mentioned in this section.

Setting Response Serialization Options, Including Encodings

You can customize how BillingCenter serializes the XML in the web service response to the client.

The most commonly customized serialization option is changing character encoding. Outgoing web service responses by default use the UTF-8 character encoding. You might want to use another character encoding for your service to improve Asian language support or other technical reasons.

Note: Incoming web service requests support any valid character encodings recognized by the Java Virtual Machine. The web service client determines the encoding that it uses, not the server or its WSDL.

To support additional serializations, add the `@WsiSerializationOptions` annotation to the web service implementation class. As an argument to the annotation, pass a list of `XmlSerializationOptions` objects. The `XmlSerializationOptions` class encapsulates various options for serializing XML, and that includes setting its `Encoding` property to a character encoding of type `java.nio.charset.Charset`.

The easiest way to get the appropriate character set object is to use the `Charset.forName(ENCODING_NAME)` static method. That method returns the desired static instance of the character set object.

For example, add the following the annotation immediately before the web service implementation class to specify the Big5 Chinese character encoding

```
@WsiSerializationOptions( new() { :Encoding = Charset.forName( "Big5" ) } )
```

For more information about other XML serialization options, such as indent levels, pretty printing, line separators, and element sorting, see “Exporting XML Data” on page 283 in the *Gosu Reference Guide*.

If the web service client is a Guidewire product, you can configure the character encoding for the request in addition to the server response. See “Setting XML Serialization Options” on page 77.

Exposing Typelists and Enumerations as String Values

For each typelist type or enumeration (Gosu or Java), the web service by default exposes this data as an enumeration value in the WSDL. This applies to both requests and responses for the service.

For example:

```
<xs:simpleType name="HouseType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="apartment"/>
    <xs:enumeration value="house"/>
    <xs:enumeration value="shack"/>
    <xs:enumeration value="mobilehome"/>
  </xs:restriction>
</xs:simpleType>
```

The published web service validates any incoming values against the set of choices and throws an exception for unknown values. Depending on the client implementation, the web service client might check if responses contain only allowed enumeration values during de-serialization. For typical cases, this approach is the desired behavior for both server and client.

For example, suppose you add new codes to a typelist or enumeration for responses. If the service returns an unexpected value in a response, it might be desirable that the client throws an exception. System integrators would quickly detect this unexpected change. The client system can explicitly refresh the WSDL and carefully check how the system explicitly handles any new codes.

However, in some cases you might want to expose enumerations as `String` values instead:

- The WSDL for a service that references typelist or enumeration types can grow in size significantly. This is true especially if some typelists or enumerations contain a vast number of choices.
- Changing enumeration values even slightly can cause incompatible WSDL. Forcing the web service client to refresh the WSDL might exacerbate upgrade issues on some projects. Although the client can refresh the WSDL from the updated service, sometimes this is an undesirable requirement. For example, perhaps new enumeration values on the server are predictably irrelevant to an older external system.
- In some cases, your actual WS-I web service client code might be middleware that simply passes through `String` values from another system. In such cases, you may not require explicit detection of enumeration changes in the web service client code.

To expose typelist types and enumerations (from Gosu or Java) as a `String` type, add the `@WsiExposeEnumAsString` annotation before the web service implementation class. In the annotation constructor, pass the typelist or enumeration type as the argument.

You can expose multiple types as `String` values by repeating the annotation multiple types on separate lines, each providing a different type as the argument.

For example, to expose the `HouseType` enumeration as a `String`, add the following line before the web service implementation class declaration:

```
@WsiExposeEnumAsString( HouseType )
```

Transforming a Generated Schema

BillingCenter generates WSDL and XSDs for the web service based on the contents of your implementation class and any of its configuration-related annotations. In typical cases, the generated files are appropriate for consuming by any web service client code.

In rare cases, you might need to do some transformation. For example, if you want to specially mark certain fields as required in the XSD itself, or to add other special comment or marker information.

You can do any arbitrary transformation on the schema by adding the `@WsiSchemaTransform` annotation. The one argument to the annotation constructor is a block that does the transformation. The block takes two arguments:

- a reference to the entire WSDL XML. From Gosu, this object is an `XmLElement` object and strongly typed to match the `<definitions>` element from the official WSDL XSD specification. From Gosu, this type is `gw.xsd.w3c.wsdl.Definitions`.
- A reference to the main generated schema for the operations and its types. From Gosu this object is an `XmLElement` object strongly typed to match the `<schema>` element from the official XSD metaschema. From Gosu this type is `gw.xsd.w3c.xmlschema.Schema`. There may be additional schemas in the WSDL that are unrepresented by this parameter but are accessible through the WSDL parameter.

The following example modifies a schema to force a specific field to be required. In other words, it strictly prohibits a `null` value. This example transformation finds a specific field and changes its XSD attribute `MinOccurs` to be 1 instead of 0:

```
@WsiSchemaTransform( \ wsdl\, schema ->{
    schema.Element.firstWhere( \ e ->e.Name == "myMethodSecondParameterIsRequired"
        ).ComplexType.Sequence.Element[1].MinOccurs = 1
} )
```

You can also change the XML associated with the WSDL outside the schema element.

Login Authentication Confirmation

In typical cases, WS-I web service client code sets up authentication and calls desired web services, relying on catching any exceptions if authentication fails. You do not need to call a specific WS-I web service as a precondition for login authentication. In effect, WS-I authentication happens with each API call.

However, if your web service client code wants to explicitly test specific authentication credentials, BillingCenter publishes the built-in `Login` web service. Call this web service's `login` method, which takes a user name as a `String` and a password as a `String`. If authentication fails, the API throws an exception.

If the authentication succeeds, that server creates a persistent server session for that user ID and returns the session ID as a `String`. The session persists after the call completes. In contrast, a normal WS-I API call creates a server session for that user ID but clears the session as soon as the API call completes.

If you call the `login` method, you must call the matching `logout` method to clear the session, passing the session ID as an argument. If you were merely trying to confirm authentication, call `logout` immediately.

However, in some rare cases you might want to leave the session open for logging purposes to track the owner of multiple API calls from one external system. After you complete your multiple API calls, finally call `logout` with the original session ID.

If you fail to call `logout`, all application servers are configured to time out the session eventually.

Stateful Session Affinity Using Cookies

By default, WS-I web services do not ask the application server to generate and return session cookies. However, BillingCenter supports cookie-based load-balancing options for web services. This is sometimes referred to as *session affinity*.

The WS-I web services layer can generate a cookie for a series of API calls. You can configure load balancing routers to send consecutive SOAP calls in the same conversation to the same server in the cluster. This feature simplifies things like customer portal integration. Repeated page requests by the same user (assuming they successfully reused the same cookie) go to the same node in the cluster.

By using session affinity, you can improve performance by ensuring that caches for that node in the cluster tend to already contain recently used objects and any session-specific data.

To create cookies for the session, append the text "`?stateful`" (with no quotes) to the WS-I API URL.

From Gosu client code, you can use code similar to following to append text to the URL:

```
uses gw.xml.ws.WsdlConfig
uses java.net.URI
uses wsi.remote.gw.webservice.ab.ab800.abcontactapi.ABContactAPI

// get a reference to an API object
var api = new ABContactAPI()

// get URL and override URL to force creation of local cookies to save session for load balancing
api.Config.ServerOverrideUrl = new URI(ABContactAPI.ADDRESS + "?stateful")

// call whatever API method you need as you normally would...
api.getReplacementAddress("12345")
```

The code might look very different depending on your choice of web service client programming language and SOAP library.

Calling a BillingCenter Web Service from Java

For WS-I web services, there are no automatically generated libraries to generate stub classes for use in Java. You must use your own procedures for converting WSDL for the web service into APIs in your preferred language. There are several ways to create Java classes from the WSDL:

- Java 6 (Java 1.6) includes a built-in utility to generate Java-compatible stubs using the `wsimport` tool.
- The CXF open source tool.
- The Axis2 open source tool.

Calling Web Services using Java 1.6 and wsimport

Java 6 (Java 1.6) includes a built-in utility that generates Java-compatible stubs from WSDL. This tool is called `wsimport` tool. This documentation uses this built-in Java tool and its output to demonstrate creating client connections to the BillingCenter web services.

You can get the WSDL for the WS-I web service from one of two sources:

- “Getting WSDL from a Running Server” on page 42
- “Generating WSDL On Disk” on page 44

The `wsimport` tool supports getting WSDL over the Internet published directly from a running application. This is the approach that this documentation uses to demonstrate how to use `wsimport`.

Note: The `wsimport` tool also supports using local WSDL files. Refer to the `wsimport` tool documentation for details.

To generate Java classes that make SOAP client calls to a SOAP API

1. Launch the server that publishes the WS-I web services.
2. On the computer from which you will run your SOAP client code, open a command prompt.
3. Change the working directory to a place on your local disk where you want to generate Java source files and `.class` files.
4. Decide the name of the subdirectory of the current directory where you want to place the Java source files. For example, you might choose the folder name `src`. Assure the subdirectory already exists before you run the `wsimport` tool in the next step. If the directory does not already exist, create the directory now. This topic calls this the `SUBDIRECTORY_NAME` directory.
5. Type the following command:

```
wsimport WSDL_LOCATION_URL -s SUBDIRECTORY_NAME
```

For `WSDL_LOCATION_URL`, type the HTTP path to the WSDL. See “Getting WSDL from a Running Server” on page 42.

For example:

```
wsimport http://localhost:PORTNUMBER/bc/ws/example/HelloWorldAPI?WSDL -s src
```

IMPORTANT You must assure the `SUBDIRECTORY_NAME` directory already exists before running the command. If the directory does not already exist, the `wsimport` action fails with the error “`directory not found`”.

6. The tool generates Java source files and compiled class files. Depending on what you are doing, you probably need only the class files or the source files, but not both.
 - The `.java` files are in the `SUBDIRECTORY_NAME` subdirectory. To use these, add this directory to your Java project’s class path, or copy the files to your Java project’s `src` folder. The location of the files represent the hierarchical structure of the web service namespace (in reverse order) followed by the fully qualified name.

The namespace is a URL that each published web service specifies in its declaration. It represents a namespace for all the objects in the WSDL. A typical namespace would specify your company domain name and perhaps other meaningful disambiguating or grouping information about the purpose of the service. For example, “`http://mycompany.com`”. You can specify the namespace for each web service by passing a namespace as a `String` argument to the `@WebService` annotation. If you do not override the namespace when you declare the web service, the default is “`http://example.com`”. For more details, see “Declaring the Namespace for a Web Service” on page 37.

The path to the Java source file has the following structure:

`CURRENT_DIRECTORY/SUBDIRECTORY_NAME/REVERSED_NAMESPACE/FULLY_QUALIFIED_NAME.java`

For example, suppose your web service fully qualified name is `com.mycompany.HelloWorld` and the namespace is the default "`http://example.com`". Suppose you use the `SUBDIRECTORY_NAME` value "src".

The `wsimport` tool generates the `.java` file at the following location:

`CURRENT_DIRECTORY/src/com/example/com/mycompany/HelloWorld.java`

- The compiled `.class` files are placed in a hierarchy (by package name) with the same basic naming convention as the `.java` files but with no `SUBDIRECTORY_NAME`. In other words, the location is:
`CURRENT_DIRECTORY/SUBDIRECTORY_NAME/REVERSED_NAMESPACE/FULLY_QUALIFIED_NAME`
- Continuing our example, the `wsimport` tool generates the `.class` files at the following location:
`CURRENT_DIRECTORY/com/example/com/mycompany/HelloWorld.class`
- To use the Java class files, add this directory to your Java project's class path, or copy the files to your Java project's `src` folder.

7. The next step depends on whether you want just the `.class` files to compile against, or whether you want to use the generated Java files.
 - To use the `.java` files, copy the `SUBDIRECTORY_NAME` subdirectory into your Java project's `src` folder.
 - To use the `.class` files, copy the files to your Java project's `src` folder or add that directory to your project's class path.

8. Write Java SOAP API client code that compiles against these new generated classes.

To get a reference to the API itself, access the WSDL port (the service type) with the syntax:

```
new API_INTERFACE_NAME().getAPI_INTERFACE_NAMESoap11Port();
```

For example, for the API interface name `HelloWorldAPI`, the Java code looks like:

```
HelloWorldAPIPortType port = new HelloWorldAPI().getHelloWorldAPISoap11Port();
```

Once you have that port reference, you can call your web service API methods directly on it.

You can publish a web service that does not require authentication by overriding the set of permissions necessary for the web service. See “Specifying Required Permissions for a Web Service” on page 38. The following is a simple example to show calling the web service without worrying about the authentication-specific code.

```
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPIPortType;

public class WsiTestNoAuth {

    public static void main(String[] args) throws Exception {
        // get a reference to the SOAP 1.1 port for this web service
        HelloWorldAPIPortType port = new HelloWorldAPI().getHelloWorldAPISoap11Port();

        // call API methods on the WS-I web service
        String res = port.helloWorld();

        // print result
        System.out.println("Web service result = " + res);
    }
}
```

See also

- “Adding HTTP Basic Authentication in Java” on page 61
- “Adding SOAP Header Authentication in Java” on page 62

Adding HTTP Basic Authentication in Java

In previous topics the examples showed how to connect to a service without authentication. The following code shows how to add HTTP Basic authentication to your WS-I client request. HTTP Basic authentication is the easiest type of authentication to code to connect to a BillingCenter WS-I web service.

```
import com.example.example.helloworldapi.HelloWorldAPI;
```

```

import com.example.example.helloworldapi.HelloWorldAPIPortType;
import com.sun.xml.internal.ws.api.message.Headers;
import javax.xml.ws.BindingProvider;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.ws.BindingProvider;
import java.util.Map;

public class WsiTest02 {

    public static void main(String[] args) throws Exception {
        System.out.println("Starting the web service client test...");

        // get a reference to the SOAP 1.1 port for this web service
        HelloWorldAPIPortType port = new HelloWorldAPI().getHelloWorldAPISoap11Port();

        // cast to BindingProvider so the following lines are easier to understand
        BindingProvider bp = (BindingProvider) port;

        // "HTTP Basic" authentication
        Map<String, Object> requestContext = bp.getRequestContext();
        requestContext.put(BindingProvider.USERNAME_PROPERTY, "su");
        requestContext.put(BindingProvider.PASSWORD_PROPERTY, "gw");

        System.out.println("Calling the service now...");
        String res = port.helloWorld();
        System.out.println("Web service result = " + res);
    }
}

```

Adding SOAP Header Authentication in Java

Although HTTP authentication is the easiest to code for most integration programmers, BillingCenter also supports optionally authenticating WS-I web services using custom SOAP headers.

For Guidewire applications, the structure of the required SOAP header is:

- An element <authentication> with the namespace `http://guidewire.com/ws/soapheaders`.
That element contains two elements:
 - <username> – Contains the username text
 - <password> – Contains the password text

This SOAP header authentication option is also known as Guidewire authentication.

The Guidewire authentication XML element looks like the following:

```

<?xml version="1.0" encoding="UTF-16"?>
<authentication xmlns="http://guidewire.com/ws/soapheaders"><username>su</username><password>gw
</password></authentication>

```

The following code shows how to use Java client code to access a web service with the fully-qualified name `example.helloworldapi.HelloWorld` using a custom SOAP header.

After authenticating, the example calls the `helloWorld` SOAP API method.

```

import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPIPortType;
import com.sun.org.apache.xml.internal.serialize.DOMSerializerImpl;
import com.sun.xml.internal.messaging.saaj.soap.ver1_1.Header1_1Impl;
import com.sun.xml.internal.messaging.saaj.soap.ver1_1.HeaderElement1_1Impl;
import com.sun.xml.internal.ws.developer.WSBindingProvider;
import com.sun.xml.internal.ws.api.message.Headers;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilderFactory;

public class WsiTest01 {

```

```
private static final QName AUTH = new QName("http://guidewire.com/ws/soapheaders",
    "authentication");
private static final QName USERNAME = new QName("http://guidewire.com/ws/soapheaders", "username");
private static final QName PASSWORD = new QName("http://guidewire.com/ws/soapheaders", "password");

public static void main(String[] args) throws Exception {
    System.out.println("Starting the web service client test...");

    // Get a reference to the SOAP 1.1 port for this web service.
    HelloWorldAPIPortType port = new HelloWorldAPI().getHelloWorldAPISoap11Port();

    // cast to WSBindingProvider so the following lines are easier to understand
    WSBindingProvider bp = (WSBindingProvider) port;

    // Create XML for special SOAP headers for Guidewire authentication of a user & password.
    Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
    Element authElement = doc.createElementNS(AUTH.getNamespaceURI(), AUTH.getLocalPart());
    Element usernameElement = doc.createElementNS(USERNAME.getNamespaceURI(),
        USERNAME.getLocalPart());
    Element passwordElement = doc.createElementNS(PASSWORD.getNamespaceURI(),
        PASSWORD.getLocalPart());

    // Set the username and password, which are content within the username and password elements.
    usernameElement.setTextContent("su");
    passwordElement.setTextContent("gw");

    // Add the username and password elements to the "Authentication" element.
    authElement.appendChild(usernameElement);
    authElement.appendChild(passwordElement);

    // Uncomment the following lines to see the XML for the authentication header.
    DOMSerializerImpl ser = new DOMSerializerImpl();
    System.out.println(ser.writeToString(authElement));

    // Add our authentication element to the list of SOAP headers.
    bp.setOutboundHeaders(Headers.create(authElement));

    System.out.println("Calling the service now...");
    String res = port.helloWorld();
    System.out.println("Web service result = " + res);
}

}
```


Calling Web Services from Gosu

Gosu code can import web services (SOAP APIs) from external systems and call these services as a SOAP client (an API consumer). The Gosu language handles all aspects of object serialization, object deserialization, basic authentication, and SOAP fault handling.

This topic includes:

- “Consuming Web Service Overview” on page 65
- “Adding Configuration Options” on page 72
- “One-Way Methods” on page 79
- “Asynchronous Methods” on page 80
- “MTOM Attachments with Gosu as Web Service Client” on page 81

Consuming Web Service Overview

Gosu supports calling WS-I compliant web services. WS-I is an open industry organization that promotes industry-wide best practices for web services interoperability among diverse systems. The organization provides several different profiles and standards. The WS-I Basic Profile is the baseline for interoperable web services and more consistent, reliable, and testable APIs.

Gosu offers native WS-I web service client code with the following features:

- Call web service methods with natural Gosu syntax for method calls
- Call web services optionally asynchronously. See “Asynchronous Methods” on page 80.
- Support one-way web service methods. See “One-Way Methods” on page 79.
- Separately encrypt requests and responses. See “Adding Configuration Options” on page 72.
- Process attachments that use the multi-step MTOM protocol. See “MTOM Attachments with Gosu as Web Service Client” on page 81.
- Sign incoming responses with digital signatures. See “Implementing Advanced Web Service Security with WSS4J” on page 78.

One of the big differences between WS-I and older styles of web services is how the client and server encodes API parameters and return results.

When you use the WS-I standards, you can use the encoding called Document Literal encoding (`document/literal`). The document-literal-encoded SOAP message contains a complete XML document for each method argument and return value. The schema for each of these documents is an industry-standard XSD file. The WSDL that describes how to talk to the published WS-I service includes a complete XSD describing the format of the embedded XML document. The outer message is very simple, and the inner XML document contains all of the complexity. Anything that an XSD can define becomes a valid payload or return value.

The WS-I standard supports a mode called RPC Literal (`RPC/literal`) instead of Document Literal. Despite the similarity in name, WS-I RPC Literal mode is not closely related to RPC encoding (RPCE). Gosu supports the WS-I RPC Literal mode for Gosu web service client code. Gosu supports RPC Literal mode by automatically converting WSDL in RPC Literal mode to equivalent WSDL in Document Literal mode.

Loading WSDL Locally Using Studio Web Service Collections

The recommended way of consuming WS-I web services is to use a web service collection in Studio. A web service collection encapsulates one or more web service endpoints, and any WSDL or XSD files they reference. If you ever want to refresh the downloaded WSDL or XSD files in the collection, simply navigate to the web service collection editor in Studio and click **Fetch Updates**.

IMPORTANT To create a web service collection in Studio, see “Using the Web Service Editor” on page 116 in the *Configuration Guide* for details.

For adding configuration options to the connection, such as authentication and security settings, there are multiple approaches. See “Adding Configuration Options” on page 72.

Web service collection (.wsc) files encapsulate the set of resources necessary to connect to a web service on an external system. If you view a web service collection in Studio and click the **Fetch Updates** button, Studio retrieves WSDL and XSD files from the servers that publish those web services. You can trigger the **Fetch Updates** process from a command line tool called `regen-from-wsc`:

- First, be sure that your `suite-config.xml` file correctly refers to your server names and port numbers for your Guidewire applications. See “Suite Configuration File Sets URLs to Guidewire Applications” on page 74.
- Next, from a command prompt in the `BillingCenter/bin` directory, type the following:
`gwbc regen-from-wsc`

The tool refreshes all of your .wsc files in your Studio configuration.

Loading WSDL Directly into the File System

To consume an external web service, you must load the WSDL and XML schema files (XSDs) for the web service. You must fetch copies of WSDL files, as well as related WSDL and XSD files, from the web services server. Fetch the copies into an appropriate place in the Gosu class hierarchy on your local system. Place them in the directory that corresponds to the package in which you want new types to appear. For example, place the WSDL and XSD files next to the Gosu class files that call the web service, organized in package hierarchies just like class files.

To automatically fetch WSDLs and XSDs from a web service server

1. Within Studio, navigate within the `Classes → wsi` hierarchy to a package in which you want to store your collection of WSDL and XSD files. Guidewire recommends that you place your web service collection in the `Resources → Configuration → Classes → wsi → remote` hierarchy.
2. Right-click and choose **New → Webservice Collection**. Studio prompts you for a name for the web service collection. Enter a name for the web service collection and click **OK**.

3. Click **Add Resource...**
4. Enter the URL of the WSDL for the external web service. This is also called the *web service endpoint URL*. For example, enter *WebServiceURL/WebServiceName.wso?wsdl*. Studio indicates whether the URL is valid. You cannot proceed until you enter a valid URL. After determining that the URL is valid, click **OK**.
5. Studio indicates that you modified the list of resource URLs and offers to fetch updated resources. Click **Yes**. You can click **Fetch Updates** at any time to refresh the WSDL from the web service server.
6. Studio retrieves the WSDL for that service. You see the resource URL in the editor's **Resources** pane.
7. Click **Fetched Resources** to view the WSDL and its associated resources.

Web Service Collections in the File System

When you fetch WSDL resources, Studio retrieves the WSDL from the web service and stores it locally at the path location in Studio where you defined your web service collection. Studio downloads the WSDL and any related XSDs into a subdirectory of that location. The subdirectory has the same name as your web service collection.

The path of the WSDL is *PACKAGE_NAME/web_service_name.wsdl*. Studio creates the value for *web_service_name* from the last part of the web service endpoint URL, not from the WSDL. For example, the URL is *MY_URL/info.wsdl* or *MY_URL/info.wso?wsdl*. Gosu creates all the types for the web service in the namespace *PACKAGE.info*.

Sample Code to Manually Fetch WSDLs and XSDs from a Web Service Server

The following sample Gosu code shows how to manually fetch web service WSDLs for test purposes or for command-line use from a web service server.

```
uses gw.xml.ws.*  
uses java.net.URL  
uses java.io.File  
  
// -- set the web service endpoint URL for the web service WSDL --  
var urlStr = "http://www.aGreatWebService.com/GreatWebService?wsdl"  
  
// -- set the location in your file system for the web service WSDL --  
var loc = "/wsi/remote/GreatWebService"  
  
// -- load the web service WSDL into Gosu --  
Wsd12Gosu.fetch(new URL(urlStr), new File(loc))
```

The first long string (*urlStr*) is the URL to the web service WSDL. The second long string (*loc*) is the path on your file system where the fetched WSDL is stored. You can run your version of the preceding sample Gosu code in the Gosu Scratchpad.

Security and Authentication

The WS-I basic profile requires support for some types of security standards for web services, such as encryption and digital signatures (cryptographically signed messages). See “Adding Configuration Options” on page 72.

Types of Client Connections

From Gosu, there are three types of WS-I web service client connections:

- Standard round trip methods (synchronous request and response)
- Asynchronous round trip methods (send the request and immediately return to the caller, and check later to see if the request finished). See “Asynchronous Methods” on page 80.
- One-way methods, which indicate a method defined to have no SOAP response at all. See “One-Way Methods” on page 79.

How Does Gosu Process WSDL?

As mentioned before, Studio adds a WSDL file to the class hierarchy automatically for each registered web service in the Web Services editor in Studio. Gosu creates all the types for your web service in the namespace `ws.web_service_name`. For this example, assume that the name of your web service is `MyService`. Gosu creates all the types for your web service in the namespace `gw.config.webservices.MyService`.

Suppose you add a Web Service in Studio and you name the web service `MyService`. Gosu creates all the types for your web service in the namespace:

```
ws.myservice.*
```

Suppose you add a WSDL file directly to your class hierarchy called `MyService.wsdl` in the package `example.p1.gs.wsic`. Gosu creates all the types for your web service in the namespace:

```
example.p1.gs.wsic.myservice.*
```

The name of `MyService` becomes lowercase `myservice` in the package hierarchy for the XML objects because the Gosu convention for package names is lowercase. There are other name transformations as Gosu imports types from XSDs.

For details, see “Normalization of Gosu Generated XSD-based Names” on page 292 in the *Gosu Reference Guide*.

The structure of a WSDL comprises the following:

- One or more services
- For each service, one or more ports

A port represents a protocol or other context that might change how the WSDL defines that service. For example, methods might be defined differently for different versions of SOAP, or an additional method might be added for some ports. WSDL might define one port for SOAP 1.1 clients, one port for SOAP 1.2 clients, one port for HTTP non-SOAP access, and so on. See discussion later in this topic for what happens if multiple ports exist in the WSDL.

- For each port, one or more methods

A method, also called an operation or action, performs a task and optionally returns a result. The WSDL includes XML schemas (XSDs), or it imports other WSDL or XSD files. Their purposes are to describe the data for each method argument type and each method return type.

Suppose the WSDL looks like the following:

```
<wsdl>
  <types>
    <schemas>
      <import schemaLocation="yourschema.xsd"/>
      <!-- now define various operations (API methods) in the WSDL ... -->
```

The details of the web service APIs are omitted in this example WSDL. Assume the web service contains exactly one service called `SayHello`, and that service contains one method called `helloWorld`. Let us assume for this first example that the method takes no arguments, returns no value, and is published with no authentication or security requirements.

In Gosu, you can call the remote service represented by the WSDL using code such as:

```
// get a reference to the web service API object in the namespace of the WSDL
// warning: this object is not thread-safe. Do not save in a static variable or singleton instance var.
var service = new .myservice.SayHello()

// call a method on the service
service.helloWorld()
```

Of course, real APIs need to transfer data also. In our example WSDL, notice that the WSDL refers a secondary schema called `yourschema.xsd`.

Studio adds any attached XSDs into the `web_service_name.wsdl.resources` subdirectory.

Let us suppose the contents of your `yourschema.xsd` file looks like the following:

```
<schema>
  <element name="Root" type="xsd:int"/>
</schema>
```

Note that the element name is "root" and it contains a simple type (`int`). This XSD represents the format of an element for this web service. The web service could declare a `<root>` element as a method argument or return type.

Now let us suppose there is another method in the `SayHello` service called `doAction` and this method takes one argument that is a `<root>` element.

In Gosu, you can call the remote service represented by the WSDL using code similar to the following:

```
// get a Gosu reference to the web service API object
// warning: this object is not thread-safe. Do not save in a static variable or singleton instance var.
var service = new ws.myservice.SayHello()

// create an XML document from the WSDL using the Gosu XML API
var x = new ws.myservice.Root()

// call a method that the web service defines
var ret = service.doAction( x )
```

The package names are different if you place your WSDL file in a different part of the package hierarchy.

Note: If you use Guidewire Studio, you do not need to manipulate the WSDL file manually. Studio automatically gets the WSDL and saving it when you add a Web Service in the user interface.

For each web service API call, Gosu first evaluates the method parameters. Internally, Gosu serializes the root `Xmlelement` instance and its child elements into XML raw data using the associated XSD data from the WSDL. Next, Gosu sends the resulting XML document to the web service. In the SOAP response, the main data is an XML document, whose schema is contained in (or referenced by) the WSDL.

Be sure to read the warnings in the section “Web Service API Objects Are Not Thread Safe” on page 69.

Web Service API Objects Are Not Thread Safe

To create a WS-I API object, use a `new` expression to instantiate the right fully-qualified type:

```
var service = new .myservice.SayHello()
```

Be warned that the WS-I API object is not thread-safe in all cases.

WARNING It is dangerous to modify any configuration when another thread might have a connection open. Also, some APIs may directly or indirectly modify the state on the API object itself.

For example, the `initializeExternalTransactionIdForNextUse` method saves information that is specific to one request, and then resets the transaction ID after one use. See “Setting Guidewire Transaction IDs” on page 75.

It is safest to follow these rules:

- Instantiate the WS-I API object each time it is needed. This careful approach allows you to modify the configuration and use API without concerns for thread-safety.
- Do not save API objects in static variables.
- Do not save API objects in instance variables of classes that are singletons, such as plugin implementations. Each member field of the plugin instance becomes a singleton and needs to be shared across multiple threads.

Learning Gosu XML APIs

All WS-I method argument types and return types are defined from schemas (the XSD embedded in the WSDL). From Gosu, all these objects are instances of subclasses of `Xmlelement`, with the specific subclass defined by the schemas in the WSDL. From Gosu, working with WS-I web service data requires that you understand Gosu XML APIs.

In many cases, Gosu hides much of the complexity of XML so you do not need to worry about it. For example, for XSD-types, in Gosu you do not have to directly manipulate XML as bytes or text. Gosu handles common types like number, date, or Base64 binary data. You can directly get or set values (such as a Date object rather than a serialized xsd:date object). The `XmlElement` class, which represents an XML element hide much of the complexity.

Other notable tips to working with XML in Gosu:

- When using a schema (an XSD, or a WSDL that references an XSD), Gosu exposes shortcuts for referring to child elements using the name of the element. See “XSD-based Properties and Types” on page 289 in the *Gosu Reference Guide*.
- When setting properties in an XML document, Gosu creates intermediate XML element nodes in the graph automatically. Use this feature to significantly improve the readability of your XML-related Gosu code. For details, see “Automatic Creation of Intermediary Elements” on page 303 in the *Gosu Reference Guide*.
- For properties that represent child elements that can appear more than once, Gosu exposes that property as a list. For list-based types like this, there is a special shortcut to be aware of. If you assign to the list index equal to the size of the list, then Gosu treats the index assignment as an insertion. This is also true if the size of the list is zero: use the [0] array/list index notation and set the property. This inserts the value into the list, which is equivalent to adding an element to the list. However, you do not have to worry about whether the list exists yet. If you are creating XML objects in Gosu, by default the lists do not yet exist.

In other words, use the syntax:

```
element.PropertyName[0] = childElement
```

If the list does not exist yet for a list property at all, Gosu creates the list upon the first insertion. In other words, suppose an element contains child elements that represent an address and the child element has the name `Address`. If the XSD declares the element could exist more than once, the `element.Address` property is a list of addresses. The following code creates a new `Address`:

```
element.Address[0] = new my.package.xsdname.elements.Address()
```

See also

- “Automatic Insertion into Lists” on page 294 in the *Gosu Reference Guide*
- “XSD-based Properties and Types” on page 289 in the *Gosu Reference Guide*
- “Gosu and XML” on page 275 in the *Gosu Reference Guide*

What Gosu Creates from Your WSDL

Within the namespace of the package of the WSDL, Gosu creates some new types.

For each service in the web service, Gosu creates a service by name. For example, if the external service has a service called `GeocodeService` and the WSDL is in the package `examples.gosu.wsdl`, then the service has the fully-qualified type `examples.gosu.wsdl.GeocodeService`. Create a new instance of this type, and you then you can call methods on it for each method.

For each operation in the web service, generally speaking Gosu creates two local methods:

- One method with the method name in its natural form, for example suppose a method is called `doAction`
- One method with the method name with the `async_` prefix, for example `async_doAction`. This version of the method handles asynchronous API calls. For details, see “Asynchronous Methods” on page 80.

Special Behavior For Multiple Ports

Gosu automatically processes ports from the WSDL identified as either SOAP 1.1 or SOAP 1.2. If both are available for any service, Gosu ignores the SOAP 1.1 ports. In some cases, the WSDL might define more than one available port (such as two SOAP 1.2 ports with different names).

For example, suppose you add a WSDL file to your class hierarchy called `MyService.wsdl` in the package `example.pl.gs.wsic`. Gosu chooses a default port to use and creates types for the web service at the following path:

```
ROOT_PACKAGE.WSDL_NAME_NORMALIZED.NORMALIZED_SERVICE_NAME
```

The `NORMALIZED_SERVICE_NAME` name of the package is the name of the service as defined by the WSDL, with capitalization and conflict resolution as necessary. For example, if there are two services in the WSDL named `Report` and `Echo`, then the API types are in the location

```
example.pl.gs.wsic.myservice.Report  
example.pl.gs.wsic.myservice.Echo
```

Gosu chooses a default port for each service. If there is a SOAP 1.2 version, Gosu prefers that version.

Additionally, Gosu provides the ability to explicitly choose a port. For example, if there is a SOAP 1.1 port and a SOAP 1.2 port, you could explicitly reference one of those choices. Gosu creates all the types for your web service ports within the `ports` subpackage, with types based on the name of each port in the WSDL:

```
ROOT_PACKAGE.WSDL_NAME_NORMALIZED.ports.SERVICE_AND_PORT_NAME
```

The `SERVICE_AND_PORT_NAME` is the service name, followed by an underscore, followed by the port name.

For example, suppose the ports are called `p1` and `p2` and the service is called `Report`. Gosu creates types within the following packages:

```
example.pl.gs.wsic.myservice.ports.Report_p1  
example.pl.gs.wsic.myservice.ports.Report_p2
```

Additionally, if the port name happens to begin with the service name, Gosu removes the duplicate service name before constructing the Gosu type. For example, if the ports are called `ReportP1`, and `helloP2`, Gosu creates types within the following packages:

```
example.pl.gs.wsic.myservice.ports.Report_P1      // NOTE: it is not Report_ReportP1  
example.pl.gs.wsic.myservice.ports.Report_helloP2  // not a duplicate, so Gosu does not remove "Hello"
```

Each one of those hierarchies would include method names for that port for that service.

A Real Example: Weather

There is a public free web service that provides the weather. You can get the WSDL for this web service at the URL <http://wsf.cdyne.com/WeatherWS/Weather.asmx?wsdl>. This web service does not require authentication or encryption.

In Studio, navigate to the package `ws.weather` in the `Classes` hierarchy. Choose `New → Web Service Collection`. Enter the previously mentioned URL in the top field.

IMPORTANT There are other fields in the Studio user interface for configuring the web service collection in Studio. Refer to “Using the Web Service Editor” on page 116 in the *Configuration Guide* for details.

The following Gosu code gets the weather in San Francisco:

```
var ws = new ws.weather.Weather()  
var r = ws.GetCityWeatherByZIP(94114)  
print( r.Description )
```

Depending on the weather, your result might be something like:

```
Mostly Sunny
```

Request XML Complexity Affects Appearance of Method Arguments

A WS-I operation defines a request element. If the request element is simply a sequence of elements, Gosu converts these elements into multiple method arguments for the operation. For example, if the request XML has a sequence of five elements, Gosu exposes this operation as a method with five arguments.

If the request XML definition uses complex XML features into the operation definition itself, Gosu does not extract individual arguments. Instead Gosu treats the entire request XML as a single XML element based on an XSD-based type.

For example, if the WSDL defines the operation request XML with restrictions or extensions, Gosu exposes that operation in Gosu as a method with a single argument. That argument contains one XML element with a type defined from the XSD.

Use the regular Gosu XML APIs to navigate that XML document from the XSD types in the WSDL. See “Introduction to the XML Element in Gosu” on page 279 in the *Gosu Reference Guide*.

Adding Configuration Options

If a web service does not need encryption, authentication, or digital signatures, you can just instantiate the service object and call methods on it:

```
// get a reference to the service in the package namespace of the WSDL
var api = new example.gosu.wsdl.myservice.SayHello()

// call a method on the service
api.helloWorld()
```

If you need to add encryption, authentication, or digital signatures, there are two approaches

- **Configuration objects** – Set the configuration options directly on the configuration object for the service instance. The service instance is the newly-instantiated object that represents the service. In the previous example, the `api` variable holds a reference for the service instance. That object has a `Config` property that contains the configuration object. For details, see “Directly Modifying the WSDL Configuration Object for a Service” on page 72.
- **Configuration providers** – Add one or more WS-I web service configuration providers in the Studio web service collection `Settings` tab. A web service configuration provider centralizes and encapsulates the steps required to add encryption, authentication, or digital signatures to your web service client code. For example, instead of adding encryption and authentication in your code each time that you call out to a single service, you can centralize that code. This has the side effect of making your web service client code look cleaner. This approach separates the code that requests the web service call from the transport-layer authentication and security configuration.

[Directly Modifying the WSDL Configuration Object for a Service](#)

To add authentication or security settings to a web service you can do so by modifying the options on the service object. To access the options from the API object (in the previous example, the object in the variable called `api`), use the syntax `api.Config`. That property contains the API options object, which has the type `gw.xml.ws.WsdlConfig`.

The WSDL configuration object has properties that add or change authentication and security settings. The `WsdlConfig` object itself is not an XML object (it is not based on `XmlElement`), but some of its subobjects are defined as XML objects. Fortunately, in typical code you do not need to really think about that difference. Instead, simply use a straightforward syntax to set authentication and security parameters. The following subtopics describe `WsdlConfig` object properties that you can set on the WSDL configuration object.

Note: For XSD-generated types, if you set a property several levels down in the hierarchy, Gosu adds any intermediate XML elements if they did not already exist. This makes your XML-related code look concise. See also “Automatic Creation of Intermediary Elements” on page 303 in the *Gosu Reference Guide*.

Adding Configuration Provider Classes (To Centralize Your WSDL Configuration)

You can add one or more WS-I web service configuration providers in the Studio web service collection **Settings** tab. A web service configuration provider centralizes and encapsulates the steps required to add encryption, authentication, or digital signatures to your web service client code.

For example, instead of adding encryption and authentication in your code each time that you call out to a single service, you can centralize that code. Using a configuration provider has the side effect of making your web service client code look cleaner. This approach separates the code that requests the web service call from the transport-layer authentication and security configuration.

A configuration provider is a class that implements the interface

`gw.xml.ws.IWsWebServiceConfigurationProvider`. This interface defines a single method with signature:

```
function configure( serviceName : QName, portName : QName, config : WsdlConfig )
```

The arguments are as follows:

- The service name, as a `QName`. This is the service as defined in the WSDL for this service.
- The port name, as a `QName`. Note that this is not a TCP/IP port. This is a port in the sense of the WSDL specification.
- A WSDL configuration object, which is the `WsdlConfig` object that an API service object contains each time you instantiate the service. For more details, see “[Directly Modifying the WSDL Configuration Object for a Service](#)” on page 72.

You can write zero, one, or more than one configuration providers and attach them to a web service collection. This means that for each new connection to one of those services, each configuration provider has an opportunity to (optionally) add configuration options to the `WsdlConfig` object.

For example, you could write one configuration provider that adds all configuration options for web services in the collection, or write multiple configuration providers that configure different kinds of options. For an example of multiple configuration providers, you could:

- Add one configuration provider that knows how to add authentication
- Add a second configuration provider that knows how to add digital signatures
- Add a third configuration provider that knows how to add an encryption layer

Separating out these different types of configuration could be advantageous if you have some web services that share some configuration options but not others. For example, perhaps all your web service collections use digital signatures and encryption, but the authentication configuration provider class might be different for different web service collections.

The list of configuration provider classes in the Studio editor is an ordered list. For typical systems, the order is very important. For example, performing encryption and then a digital signature results in different output than adding a digital signature and then adding encryption. You can change the order in the list by clicking a row and clicking **Move Up** or **Move Down**.

WARNING The list of configuration providers in the Studio editor is an ordered list. If you use more than one configuration provider, carefully consider the order you want to specify them.

The following is an example of a configuration provider:

```
package wsi.remote.gw.webservice.ab

uses javax.xml.namespace.QName
uses gw.xml.ws.WsdlConfig
uses gw.xml.ws.IWsWebServiceConfigurationProvider

class ABConfigurationProvider implements IWsWebServiceConfigurationProvider {

    override function configure(serviceName : QName, portName : QName, config : WsdlConfig) {
        config.Guidewire.Authentication.Username = "su"
        config.Guidewire.Authentication.Password = "gwg"
    }
}
```

```
}
```

In this example, the configuration provider adds Guidewire authentication to every connection that uses that configuration provider. Any web service client code for that web service collection does not need to bother with adding these options with each use of the service. (For more information about Guidewire authentication, see “Guidewire Authentication” on page 74.)

HTTP Authentication

To add simple HTTP authentication to an API request, use the basic HTTP authentication object at the path as follows. Suppose *api* is a reference to a SOAP API that you have already instantiated with the new operator. The properties on which to set the name and password are on the object:

```
api.Config.Http.Authentication.Basic
```

That object has a **Username** property for the user name and a **Password** property for the password. Set those two values with the desired user name and password.

For example:

```
// Get a reference to the service in the package namespace of the WSDL.  
var service = new example.gosu.wsi.myservice.SayHello()  
  
service.Config.Http.Authentication.Basic.Username = "jms"  
service.Config.Http.Authentication.Basic.Password = "b5"  
  
// Call a method on the service.  
service.helloWorld()
```

Guidewire Authentication

To add Guidewire application authentication to API request, use the Guidewire authentication object at the path as follows. In this example, *api* is a reference to a SOAP API that you have already instantiated with the new operator: *api.Config.Guidewire.Authentication*.

That object has a **Username** property for the user name and a **Password** property for the password. Set those two values with the desired user name and password.

For example:

```
// Get a reference to the service in the package namespace of the WSDL.  
var service = new example.gosu.wsi.myservice.SayHello()  
  
service.Config.Guidewire.Authentication.Username = "jms"  
service.Config.Guidewire.Authentication.Password = "b5"  
  
// Call a method on the service.  
service.helloWorld()
```

Suite Configuration File Sets URLs to Guidewire Applications

BillingCenter uses a single configuration file for configuring all web service URLs to other Guidewire InsuranceSuite applications. For Guidewire InsuranceSuite integrations that use WS-I web services, always use this suite configuration file to set the URLs. The suite configuration file is called `suite-config.xml`. In Studio, find it under **Configuration** → **config** → **suite**, then click `suite-config.xml`. Alternatively, type Shift-Ctrl+N and type the file name.

By default, all subelements are commented out. If you have multiple Guidewire products on one server for testing, the file might look like the following:

```
<suite-config xmlns="http://guidewire.com/config/suite/suite-config">  
  <product name="cc" url="http://localhost:8080/cc"/>  
  <product name="pc" url="http://localhost:8180/pc"/>  
  <product name="ab" url="http://localhost:8280/ab"/>  
  <product name="bc" url="http://localhost:8580/bc"/>  
</suite-config>
```

In production, the applications would be on separate physical servers with different domain names for each application.

You can use the system environment setting `env` to trigger different values within the file. Each `<product>` element supports the `env` attribute to specifies the element is enabled only for that environment setting. See “Defining the Application Server Environment” on page 14 in the *System Administration Guide*.

The following example shows a URL with different values for development and production `env` settings:

```
<suite-config xmlns="http://guidewire.com/config/suite/suite-config">
  <product name="ab" url="http://localhost:8080/cc" env="development"/>
  <product name="ab" url="http://ProductionClaimCenterServer:8080/cc" env="production"/>
</suite-config>
```

When importing WS-I WSDL, BillingCenter does the following:

1. Checks to see whether there is a WSDL port of element type `<gwwsdl:address>`. If this is found, BillingCenter ignores any other ports on the WSDL for this service.
2. If so, it looks for shortcuts of the syntax `${PRODUCT_NAME_SHORTCUT}`. For example: `${bc}`
3. If that product name shortcut is in the `suite-config.xml` file, BillingCenter substitutes the URL from the XML file to replace the text `${PRODUCT_NAME_SHORTCUT}`. If the product name shortcut is not found, Gosu throws an exception during WSDL parsing.

For web services that Guidewire applications publish, all WSDL documents have the `<gwwsdl:address>` port in the WSDL. The Guidewire application automatically specifies the application that published it using the standard two-letter application shortcut. For example, for BillingCenter the abbreviation is `bc`. For example:

```
<wsdl:port name="TestServiceSoap11Port" binding="TestServiceSoap11Binding">
  <soap11:address location="http://172.24.11.41:8480/bc/ws/gw/test/TestService/soap11" />
  <gwwsdl:address location=" ${bc}/ws/gw/test/TestService/soap11" />
</wsdl:port>
```

IMPORTANT When integrating with Guidewire InsuranceSuite applications, always use the `suite-config.xml` file to centrally configure domain names and port numbers of all applications.

Accessing the Suite Configuration File Using APIs

BillingCenter exposes the suite configuration file as APIs that you can access from Gosu. The `gw.api.suite.GuidewireSuiteUtil` class can look up entries in the file by the product code. This class has a static method called `getProductInfo` that takes a `String` that represents the product. Pass the `String` that is the `<product>` element’s `name` attribute. The method returns the associated URL from the `suite-config.xml` file.

For example:

```
uses gw.api.suite.GuidewireSuiteUtil
var x = GuidewireSuiteUtil.getProductInfo("cc")
print(x.Url)
```

For the earlier `suite-config.xml` example file, this code prints:

```
http://localhost:8080/cc
```

Setting Guidewire Transaction IDs

If the web service you are calling is hosted by a Guidewire application, there is an optional feature to detect duplicate operations from external systems that change data. The service must add the `@WsiCheckDuplicateExternalTransaction` annotation. For implementation details, see “Checking for Duplicate External Transaction IDs” on page 51.

If this feature is enabled for an operation, BillingCenter checks for the SOAP header `<transaction_id>` in namespace `http://guidewire.com/ws/soapheaders`.

To set this SOAP header, call the `initializeExternalTransactionIdForNextUse` method on the API object and pass the transaction ID as a `String` value.

IMPORTANT BillingCenter sends the transaction ID with the next method call and applies only to that one method call. If you require subsequent method calls with a transaction ID, call `initializeExternalTransactionIdForNextUse` again before each external API that requires a transaction ID. If your next call to an web service external operation does not require a transaction ID, there is no need for you to call the `initializeExternalTransactionIdForNextUse` method.

For example:

```
uses gw.xsd.guidewire.soapheaders.TransactionID
uses gw.xml.ws.WsdlConfig
uses java.util.Date
uses wsi.local.gw.services.wsidbupdateservice.faults.DBAlreadyExecutedException

function callMyWebService {

    // Get a reference to the service in the package namespace of the WSDL.
    var service = new example.gosu.wsi.myservice.SayHello()

    service.Config.Guidewire.Authentication.Username = "su"
    service.Config.Guidewire.Authentication.Password = "gw"

    // create a transaction ID that has an external system prefix and then a guaranteed unique ID
    // If you are using Guidewire messaging, you may want to use the Message.ID property in your ID.
    transactionIDString = "MyExtSys1:" + getUniqueTransactionID() // somehow create your own unique ID

    // Set the transaction ID for the next method call (and only the next method call) to this service
    service.initializeExternalTransactionIdForNextUse(transactionIDString)

    // Call a method on the service -- a transaction ID is set only for the next operation
    service.helloWorld()

    // Call a method on the service -- NO transaction ID is set for this operation!
    service.helloWorldMethod2()

    transactionIDString = "MyExtSys1:" + getUniqueTransactionID() // somehow create your own unique ID

    // Call a method on the service -- a transaction ID is set only for the next operation
    service.helloWorldMethod3()

}
```

Setting a Timeout

To set the timeout value (in milliseconds), set the `CallTimeout` property on the `WsdlConfig` object for that API reference.

For example:

```
// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.CallTimeout = 30000 // 30 seconds

// call a method on the service
service.helloWorld()
```

Custom SOAP Headers

SOAP HTTP headers are essentially XML elements attached to the SOAP envelope for the web service request or its response. Your code might need to send additional SOAP headers to the external system, such as custom headers for authentication or digital signatures. You also might want to read additional SOAP headers on the response from the external system.

To add SOAP HTTP headers to a request that you initiate, first construct an XML element using the Gosu XML APIs (`XmLElement`). Next, add that `XmLElement` object to the list in the location `api.Config.RequestSoapHeaders`. That property contains a list of `XmLElement` objects, which in generics notation is the interface type `java.util.List<XmLElement>`.

To read (get) SOAP HTTP headers from a response, it only works if you use the asynchronous SOAP request APIs described in “Asynchronous Methods” on page 80. There is no equivalent API to get just the SOAP headers on the response, but you can get the response envelope, and access the headers through that. You can access the response envelope from the result of the asynchronous API call. This is an `gw.xml.ws.AsyncResponse` object. On this object, get the `ResponseEnvelope` property. For SOAP 1.2 envelopes, the type of that response is type `gw.xsd.w3c.soap12_envelope.Envelope`. For SOAP 1.1, the type is the same except with "soap11" instead of "soap12" in the name.

From that object, get the headers in the `Header` property. That property contains a list of XML objects that represent all the headers.

Server Override URL

To override the server URL, for example for a test-only configuration, set the `ServerOverrideUrl` property on the `WsdlConfig` object for your API reference. It takes a `java.net.URI` object for the URL.

For example:

```
// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.ServerOverrideUrl = new URI("http://testingserver/xx")

// call a method on the service
service.helloWorld()
```

Setting XML Serialization Options

To send a SOAP request to the SOAP server, BillingCenter takes an internal representation of XML and serializes the data to actual XML data as bytes. For typical use, the default XML serialization settings are sufficient. If you need to customize these settings, you can do so.

The most common serialization option to set is changing the character encoding to something other than the default, which is UTF-8.

You can change serialization settings by getting the `XmLSerializationOptions` property on the `WsdlConfig` object, which has type `gw.xml.XmLSerializationOptions`. Modify properties on that object to set various serialization settings.

For full information about XML serialization options, such as encoding, indent levels, pretty printing, line separators, and element sorting, see “Exporting XML Data” on page 283 in the *Gosu Reference Guide*.

The easiest way to get the appropriate character set object for the encoding is to use the `Charset.forName(ENCODING_NAME)` static method. That method returns the desired static instance of the character set object.

For example, to change the encoding to the Chinese encoding Big5:

```
uses java.nio.charset.Charset

// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.XmLSerializationOptions.Encoding = Charset.forName( "Big5" )

// call a method on the service
service.helloWorld()
```

This API sets the encoding on the outgoing request only. The SOAP server is not obligated to return the response XML in the same character encoding.

If the web service is published from a Guidewire product, you can configure the character encoding for the response. See “Setting Response Serialization Options, Including Encodings” on page 57.

Setting Locale in a Guidewire Application

WS-I web services published on a Guidewire application support setting a specific international locale to use (to override) while processing this web services request.

For example:

```
// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.ws.i.myservice.SayHello()

// set the locale to the French language in the France region
service.Config.Guidewire.Locale = "fr_FR"

// call a method on the service
service.helloWorld()
```

This creates a SOAP header similar to the following:

```
<soap12:Envelope xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
<soap12:Header>
  <gwsoap:locale xmlns:gwsoap="http://guidewire.com/ws/soapheaders">fr_FR</gwsoap:locale>
</soap12:Header>
<soap12:Body>
  <getPaymentInstrumentsFor xmlns="http://example.com/gw/webservice/bc/bc700/PaymentAPI">
    <accountNumber>123</accountNumber>
  </getPaymentInstrumentsFor>
</soap12:Body>
</soap12:Envelope>
```

Implementing Advanced Web Service Security with WSS4J

For security options beyond HTTP Basic authentication and optional SOAP header authentication, you can use an additional set of APIs to implement whatever additional security layers.

For example, you might want to add additional layers of encryption, digital signatures, or other types of authentication or security.

From the SOAP client side, the way to add advanced security layers to outgoing requests is to apply transformations of the stream of data for the request. You can transform the data stream incrementally as you process bytes in the stream. For example, you might implement encryption this way. Alternatively, some transformations might require getting all the bytes in the stream before you can begin to output any transformed bytes. Digital signatures would be an example of this approach. You may use multiple types of transformations. Remember that the order of them is important. For example, an encryption layer followed by a digital signature is a different output stream of bytes than applying the digital signature and then the encryption.

Similarly, getting a response from a SOAP client request might require transformations to understand the response. If the external system added a digital signature and then encrypted the XML response, you need to first decrypt the response, and then validate the digital signature with your keystore.

The standard approach for implementing these additional security layers is the Java utility WSS4J, but you can use other utilities as needed. The WSS4J utility includes support for the WSS security standard.

Outbound Security

To add a transformation to your outgoing request, set the `RequestTransform` property on the `WsdlConfig` object for your API reference. The value of this property is a Gosu block that takes an input stream (`InputStream`) as an argument and returns another input stream. Your block can do anything it needs to do to transform the data.

Similarly, to transform the response, set the `ResponseTransform` property on the `WsdlConfig` object for your API reference.

The following simple example shows you could implement a transform of the byte stream. The transform is in both outgoing request and the incoming response. In this example, the transform is an XOR (exclusive OR) transformation on each byte. In this simple example, simply running the transformation again decodes the request.

The following code implements a service that applies the transform to any input stream. The code that actually implements the transform is as follows. This is a web service that you can use to test this request.

The class defines a static variable that contains a field called `_xorTransform` that does the transformation.

```
package gw.xml.ws

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiRequestTransform
uses java.io.ByteArrayInputStream
uses gw.util.StreamUtil
uses gw.xml.ws.annotation.WsiResponseTransform
uses gw.xml.ws.annotation.WsiAvailability
uses gw.xml.ws.annotation.WsiPermissions
uses java.io.InputStream

@WsiWebService
@WsiAvailability( NONE )
@WsiPermissions( {} )
@WsiRequestTransform( WsiTransformTestService._xorTransform )
@WsiResponseTransform( WsiTransformTestService._xorTransform )
class WsiTransformTestService {

    // THE FOLLOWING DECLARES A GOSU BLOCK THAT IMPLEMENTS THE TRANSFORM
    public static var _xorTransform( is : InputStream ) : InputStream = \ is ->{
        var bytes = StreamUtil.getContent( is )
        for ( b in bytes index idx ) {
            bytes[ idx ] = ( b ^ 17 ) as byte // xor encryption
        }
        return new ByteArrayInputStream( bytes )
    }

    function add( a : int, b : int ) : int {
        return a + b
    }
}
```

The following code connects to the web service and applies this transform on outgoing requests and the reply.

```
package gw.xml.ws

uses gw.testng.TestBase
uses gw.testng.RunLevel
uses org.xml.sax.SAXParseException

@RunLevel( NONE )
class WsiTransformTest extends TestBase {

    function testTransform() {
        var ws = new wsi.local.gw.xml.ws.wsitransformtestservice.WsiTransformTestService()
        ws.Config.RequestTransform = WsiTransformTestService._xorTransform
        ws.Config.ResponseTransform = WsiTransformTestService._xorTransform
        ws.add( 3, 5 )
    }
}
```

One-Way Methods

A typical WS-I method invocation has two parts: the SOAP request, and the SOAP response. Additionally, WS-I supports a concept called *one-way methods*. A one-way method is a method defined in the WSDL to provide no SOAP response at all. The transport layer (HTTP) may send a response back to the client, however, but it contains no SOAP response.

Gosu fully supports calling one-way methods. Your web service client code does not have to do anything special to handle one-way methods. Gosu handles them automatically if the WSDL specifies a method this way.

IMPORTANT Be careful not to confuse one-way methods with asynchronous methods. For more information about asynchronous methods, see “Asynchronous Methods” on page 80.

Asynchronous Methods

Gosu supports optional asynchronous calls to web services. Gosu exposes alternate web service methods signatures on the service with the `async_` prefix. Gosu does not generate the additional method signature if the method is a one-way method. The asynchronous method variants return an `AsyncResponse` object. Use that object with a polling design pattern (check regularly whether it is done) to choose to get results later (synchronously in relation to the calling code).

See the introductory comments in “Consuming Web Service Overview” on page 65 for related information about the basic types of connections for a method.

IMPORTANT Be careful not to confuse one-way methods with asynchronous methods. For more information about one-way methods, see “One-Way Methods” on page 79.

The `AsyncResponse` object contains the following properties and methods:

- `start` method – initiates the web service request but does not wait for the response
- `get` method – gets the results of the web service, waiting (blocking) until complete if necessary
- `RequestEnvelope` – a read-only property that contains the request XML
- `ResponseEnvelope` – a read-only property that contains the response XML, if the web service responded
- `RequestTransform` – a block (an in-line function) that Gosu calls to transform the request into another form. For example, this block might add encryption and then add a digital signature.
- `ResponseTransform` – a block (an in-line function) that Gosu calls to transform the response into another form. For example, this block might validate a digital signature and then decrypt the data.

The following is an example of calling the a synchronous version of a method contrasted to using the asynchronous variant of it.

```
var ws = new ws.weather.Weather()

// Call the REGULAR version of the method.
var r = ws.GetCityWeatherByZIP("94114")
print( "The weather is " + r.Description )

// Call the **asynchronous** version of the same method
// -- Note the "async_" prefix to the method
var a = ws.async_GetCityWeatherByZIP("94114")

// By default, the async request does NOT start automatically.
// You must start it with the start() method.
a.start()

print("the XML of the request for debugging... " + a.RequestEnvelope)
print("")

print ("in a real program, you would check the result possibly MUCH later...")

// Get the result data of this asynchronous call, waiting if necessary.
var laterResult = a.get()
print("asynchronous reply to our earlier request = " + laterResult.Description)

print("response XML = " + a.ResponseEnvelope.asUTFString())
```

MTOM Attachments with Gosu as Web Service Client

The W3C Message Transmission Optimization Mechanism (MTOM) is a method of efficiently sending binary data to and from web services as attachments outside the normal response body.

The main response contains placeholder references for the attachments. The entire SOAP message envelope for MTOM contains multiple parts. The raw binary data is in other parts of the request than the normal SOAP request or response. Other parts can have different MIME encoding. For example, it could use the MIME encoding for the separate binary data. This allows more efficient transfer of large binary data.

When Gosu is the web service client, MTOM is not supported in the initial request. However, if an external web service uses MTOM in its response, Gosu automatically receives the attachment data. There is no additional step that you need to perform to support MTOM attachments.

On a related topic, you can support MTOM support when publishing a web service. See “Web Service Invocation Context” on page 39.

Billing Web Services

BillingCenter publishes web services for external systems to manipulate billing information.

This topic includes:

- “Billing Web Services Overview” on page 83
- “Policy Administration System Core Web Service APIs (BillingAPI)” on page 85
- “Invoice Details Web Service APIs (InvoiceDetailsAPI)” on page 106
- “Payments Web Service APIs (PaymentAPI)” on page 107
- “Payment Instrument Web Service APIs (PaymentInstrumentAPI)” on page 112
- “Billing Summary Web Service APIs (BillingSummaryAPI)” on page 113
- “Trouble Ticket Web Service APIs (TroubleTicketAPI)” on page 114
- “Other Billing Web Service APIs (BCAPI)” on page 115

See also

- For important background and usage information about web services, see “Web Services Introduction” on page 27.
- For information about the PolicyCenter integration with BillingCenter, see “Billing Integration” in the *PolicyCenter Integration Guide*. The integration information is in the PolicyCenter documentation set, not the BillingCenter documentation set.

Billing Web Services Overview

BillingCenter publishes web services for external systems to manipulate billing information. Some of the web services are used for the BillingCenter integration with PolicyCenter. However, you can use any of the web services with other policy administration systems.

See also

- For information about the PolicyCenter integration with BillingCenter, see “Billing Integration” in the *PolicyCenter Integration Guide*. The integration information is in the PolicyCenter documentation set, not the BillingCenter documentation set.

BillingCenter Data Transfer Objects (DTOs)

The WS-I web services do not support entity data directly as method arguments or return values. Instead, BillingCenter web service APIs use separate *data transfer objects* (DTOs) to encapsulate entity data. Carefully read this section before extending the data model.

BillingCenter uses two types of DTOs to represent data that billing web services receive or send back:

- Gosu class DTOs
- XML DTOs

Both types of DTOs encapsulate entity and other data for transfer between BillingCenter and external systems. A DTO transfers only the data required by a web service method, rather than the entire set of data for an entity instance. In most cases, a DTO property represents a simple entity property, such as a Date property, and both DTO property and entity property have the same names.

For entity properties that represent links to related objects, the definitions and names of DTO properties differ slightly from their corresponding entity definitions. BillingCenter implements DTO properties that link to related objects in one of two ways:

- The DTO property includes the related objects directly.
- The DTO property includes the related objects indirectly by reference to their unique identifiers, typically their public IDs.

The name of a DTO property helps you determine how the property implements its representation of the links to related objects:

- If the name of a DTO property ends with the singular `Info`, typically it is a DTO that represents a related object. The related object is of the entity type named without the suffix.

For example, a DTO property named `InvoiceStreamInfo` typically is a DTO that represents a related instance of the `InvoiceStream` entity type.

- If the name of a DTO property ends with the plural `Infos`, typically it is a list of DTOs that represents related objects. The related objects are of the entity type named without the suffix.

For example, a DTO property named `ChargeInfos` typically is a list of DTOs that represent related instances of the `Charge` entity type.

- If the name of a DTO property ends with the singular `PublicID` or `ID`, it is a simple `String` property that references the public ID for the related object. The related object is of the entity type named without the suffix.

- If the name of a DTO property ends with the plural `IDs`, it is a list of simple `String` properties that reference the public IDs for related objects. The related objects are of the entity type named without the suffix.

Gosu Class DTOs

Guidewire implements many DTOs as Gosu classes that you can view and edit in Studio. If a Gosu class name ends with `DTO`, it is likely a Gosu class DTO for the entity type named without the suffix. For example, a Gosu class called `AddressDTO` is likely a DTO for the `Address` entity type. However, some Gosu class DTOs do not have the `DTO` suffix.

If you extend the entity data model with a property that a web service must receive or send back, you must also extend the corresponding Gosu class DTO. Add to the set of properties in the corresponding Gosu class named with the DTO suffix. For example, if you add an important property to the `Address` entity, also add the property to the Gosu class `AddressDTO`.

IMPORTANT Before editing Gosu class DTO files, carefully review the introductory comments at the top of each DTO file.

XML DTOs

Guidewire implements some DTOs as XML/XSD types defined in the XSD file called `entity.xsd`.

If you extend the entity data model with a property that a web service must receive or send back, also extend the corresponding XML DTO. Add to the set of properties in the corresponding XML/XSD type definition. For example, if you add an important property to `Address` entity, also add the property to the `Address` part of the `entity.xsd` file.

Some of the XML/XSD types in `entity.xsd` have Gosu enhancements. Gosu enhancements provide additional functionality, such as validating received data or moving data between DTOs and entity instances. If you add properties to the XML/XSD type for an XML DTO, you may need to modify its Gosu enhancements to manage the DTO properties that you added.

Policy Administration System Core Web Service APIs (BillingAPI)

The BillingAPI web service provides methods that external policy systems can call to send BillingCenter important policy and account changes, send BillingCenter billing information, and query BillingCenter for information. The BillingAPI web service defines most of the BillingCenter side of the built-in integration with PolicyCenter. Other external policy administration systems can also call these APIs.

This topic includes:

- “Overview of the BillingAPI Web Service” on page 86
- “Account APIs in BillingAPI” on page 90
- “Policy Period APIs in BillingAPI” on page 91
- “Producer APIs in BillingAPI” on page 97
- “Payment Plan APIs in BillingAPI” on page 100
- “Payment Allocation Plans API in BillingAPI” on page 101
- “Bill and Commission Plan APIs in BillingAPI” on page 102
- “Final Audit APIs in BillingAPI” on page 104
- “Update the Written Date on a Charge in BillingAPI” on page 105
- “Get Account Unapplied Funds Methods in BillingAPI” on page 105
- “Contact API in BillingAPI” on page 105

See also

- For information about the PolicyCenter integration with BillingCenter, see “Billing Integration” in the *PolicyCenter Integration Guide*. The integration information is in the PolicyCenter documentation set, not the BillingCenter documentation set.

Overview of the BillingAPI Web Service

If you use the built-in integration with PolicyCenter and you need to modify BillingCenter behavior, modify the web services implementation class in BillingCenter for BillingAPI in Studio.

There may be multiple versions of BillingAPI visible in your copy of Studio. Carefully look at the package names, which indicate the version of BillingCenter that introduced that version of the API. For example, if the package includes bc801 or bc.801, that represents BillingCenter 8.0.1.

If you use another policy administration system, follow the design patterns of the BillingAPI web service to design a custom but similar web service.

For example, suppose a policy administration system issues a new policy. The external policy administration system calls the BillingCenter BillingAPI web service method `issuePolicyPeriod`. The request includes:

- The policy itself, including associated contacts
- The billing implications of the policy change, in the form of charges

BillingCenter uses this information to send bills appropriately for the new policy.

Identifying Policy Periods in BillingAPI

For BillingAPI methods that need a policy period, most methods take a DTO that contains multiple properties representing the period:

- `PCPolicyPublicID` – public ID (`String`) of the `Policy` entity instance in PolicyCenter.
- `TermNumber` – term number (`int`)
- `PolicyNumber` – policy number (`String`)

Generally speaking, BillingCenter uses the following algorithm to find the policy period:

1. BillingCenter tries to find the policy using `PCPolicyPublicID` and `TermNumber`.
2. If that does not succeed, BillingCenter tries to find the policy using `PolicyNumber` and `TermNumber`.

For some methods, instead of taking those properties in a DTO, they are direct method arguments. However, the search algorithm is the same.

Understanding Billing Instruction Subtypes

BillingCenter internally uses what it calls *billing instructions* to track requests to modify billing information. The following table lists the billing instruction subtypes, including notes about how these billing instructions are used in the built-in PolicyCenter integration.

Note: For detailed information about PolicyCenter integration with BillingCenter, the most complete information is the separate PolicyCenter documentation set, not the BillingCenter documentation set. In the PolicyCenter documentation, refer to the “Billing Integration” topic of the *PolicyCenter Integration Guide*.

The `BillingInstruction` entity is the root of several billing instruction subtypes. Multiple levels of subtypes exist, such as `AccountGeneral`, which is a subtype of `AcctBillingInstruction`, which is a subtype of the root entity. The following table summarizes the `BillingInstruction` subtypes. The arrow symbol (→) and indentation define lower subtype levels.

| BillingCenter BillingInstruction subtype | Description and important additional properties |
|---|---|
| <code>AcctBillingInstruction</code> | Root for account-related billing instruction. The <code>Account</code> property of this instruction must be specified and reference the <code>PublicID</code> of the relevant account. |
| → <code>AccountGeneral</code> | Specifies the effective date of the <code>AcctBillingInstruction</code> . If the <code>AccountGeneral</code> instruction is used, the <code>BillingInstructionDate</code> property must specify the effective date. |

| BillingCenter BillingInstruction subtype | Description and important additional properties |
|---|--|
| → CtlBillingInstruction | Root for collateral-related billing instructions. |
| → CollateralBI | Specifies the collateral requirement that generated the associated charge. Note: In the base configuration, PolicyCenter does not trigger code in BillingCenter that creates this billing instruction. Instead, PolicyCenter sends deposit requirements with the policy job. In other words, the DepositRequirement property within the main policy billing instruction contains the information. |
| → SegregatedCollReqBI | Specifies the segregated collateral requirement that owns this charge. |
| P1cyBillingInstruction | Root for policy-related billing instruction. Includes a property to specify the deposit requirement that exists after the billing instruction is executed. Also includes an array of PaymentPlanModifier objects to apply to the existing payment plan. |
| → BaseGeneral | Specifies the policy period and the effective date of the billing instruction. Both properties are required. The Policy property must specify the PublicID of the PolicyPeriod. |
| → ExistingPlcyPeriodBI | Lets you associate the charges with an existing policy period and provide an effective date (ModificationDate in BillingCenter) for the charges. |
| → Audit | Used to make changes to an existing policy as part of an audit. Properties specify whether this is a final audit and whether the charge represents the total or incremental premium for the policy. BillingCenter does not support total premium in its default integration, but it can be customized to do so. |
| → Cancellation | Policy cancellation. Includes additional properties: <ul style="list-style-type: none">• CancellationReason – String that describes the reason for the cancellation.• CancellationType – Values are defined in the CancellationType typelist, which has Flat, Prorata, and Shortrate in the base configuration.• HoldUnbilledPremiumCharges – Boolean field specifying whether all unbilled premium charges on the policy will be held at the end of the cancellation. |
| → General | A “catch-all” general-purpose billing instruction for an existing policy. Policy administration systems may use this subtype for billing instructions that do not fit any of the other subtypes. |
| → PolicyChange | Used to specify changes to an existing policy. |
| → PremiumReportBI | Used to send premium report information. |
| → Reinstate | Used to reinstate a policy. |
| → NewPlcyPeriodBI | All billing instructions for policy transactions that create a new policy period. This subtype has properties with a list of producer codes for the new period. |
| → Issuance | Attach a new PolicyPeriod and link it to an existing Account. Automatically creates the new Policy for that PolicyPeriod. |
| → NewRenewal | Lets you attach a new PolicyPeriod. It also has a link to an Account. The difference between this billing instruction and the Renewal billing instruction is that this one is for a policy that is a renewal for the carrier but new for BillingCenter. This instruction also creates a new Policy entity for the policy period. |
| → Renewal | Lets you attach a new PolicyPeriod and link it to an existing Account. It expects to have a prior policy period. It links the new period to the existing policy. |
| → Rewrite | Rewrite a policy. In the built-in integration, the Rewrite instruction duplicates the behavior of the Renewal billing instruction. Carriers who required specialized policy-period behavior can customize this instruction. |
| → PremiumReportDueDate | Informs BillingCenter that a premium report is due on a certain date for a policy. BillingCenter handles delinquency if it does not receive the report on time. Note: In the base configuration, PolicyCenter does not trigger code in BillingCenter that creates this billing instruction. |
| ReversalBillingInstruction | Reserved for Guidewire internal use. |

At the time that PolicyCenter binds a job that can generate premium transactions, PolicyCenter sends a billing instruction to BillingCenter. Even if there are no charges, BillingCenter may need to know about a change to the policy period, such as knowing whether the period was canceled or reinstated. The following table summarizes what type of billing instruction PolicyCenter sends for different situations. A factor that determines what to send is whether PolicyCenter created a new policy and/or new period.

| PolicyCenter Policy Transaction | BillingCenter Billing Instruction Subtype | Conditions and comments |
|------------------------------------|--|---|
| submission | Issuance | Even if bind only (bind and bill, with delayed issuance), PolicyCenter generates charges for billing |
| issuance | PolicyChange | PolicyCenter sends this billing instruction if the issuance creates at least one premium transaction or the following properties change: PeriodStart, PeriodEnd, or Producer Code of Record. Although PolicyCenter calls this an Issuance, because the submission already establishes the period, then PolicyCenter simply sends adjusting charges for an existing period. |
| policy change | PolicyChange | PolicyCenter sends this billing instruction if there is at least one premium transaction or one of the following properties change: PeriodStart, PeriodEnd, or Producer Code of Record. |
| cancellation | Cancellation | All cancellations |
| reinstatement | Reinstatement | All reinstatements |
| renewal (regular) | Renewal | For regular renewals. Policy must already exist in BillingCenter Note: Not used in a <i>conversion on renewal</i> scenario |
| renewal (new renewal) | NewRenewal | For new renewals. This is the policy period in PolicyCenter. Typically you need to create the policy in BillingCenter and create a new period. Note: Used in a <i>conversion on renewal</i> scenario |
| rewrite | Rewrite | A rewrite establishes a new period for the same policy. |
| final audit | Audit | All final audits. |
| premium report | PremiumReportBI | All premium reports. |

IMPORTANT For detailed information about the PolicyCenter integration with BillingCenter, the best source of information is in the PolicyCenter documentation. Specifically, refer to the “Billing Integration” topic of the *PolicyCenter Integration Guide*.

Billing Instructions Contain Charges

The `entity.xsd` file defines the DTOs for billing instructions in `BillingAPI`. That file contains a declaration for an element called `BillingInstructionInfo`. The `BillingInstructionInfo` element is equivalent to the root class for all BillingCenter billing instruction DTO objects.

Each billing instruction encapsulates a list of charges. A charge is the cost of one part of the policy, such as a premium or a fee. The `BillingInstructionInfo` object stores the list of charges in its `ChargeInfos` property. The `ChargeInfos` property contains a list of `ChargeInfo` DTO objects. Each `ChargeInfo` DTO object encapsulates a single charge, similar to the `Charge` entity type.

IMPORTANT Each charge contains an `Amount` property that is the amount of the charge. In the web service API, BillingCenter represents this as a `String` value not an actual numerical type. The `Amount` value represents the `MonetaryAmount` type.

A MonetaryAmount is a String object that encapsulates two types of data:

- a BigDecimal value (converted to a String)
- a currency (a Currency typecode as a String)

Note that the web services do not take a list of invoice items. BillingCenter generates invoice items as appropriate from the charges.

Each ChargeInfo DTO object contains a charge pattern property (ChargePattern). Set this property to the public ID of an existing charge pattern within BillingCenter. To create and manage charge patterns in BillingCenter, administrative users click the Administration tab, then click on the Charge Patterns navigation item. If you create a new charge pattern in the BillingCenter user interface, each charge pattern has a user-specified charge name and a charge pattern type. Select the pattern type from a finite number of choices as you create the charge pattern. Each charge pattern in that screen represents a ChargePattern entity. Do not confuse the ChargePattern entity with the ChargePattern typelist.

Configuration Logic Before Executing Billing Instructions

You can add additional application logic before BillingCenter executes a billing instruction.

The IBillingInstruction plugin interface contains methods that BillingCenter calls each time it calls a billing instruction execute method. Implement this plugin interface to add your own logic. For more information, see “Billing Instruction Execution Customization Plugin” on page 167.

Policy System Transaction IDs in BillingAPI

Some web service methods use a *policy system transaction ID*, which is a String value that uniquely identifies this request from the policy system. BillingCenter tracks this transaction ID and determines whether this request already occurred. In other words, BillingCenter determines if this is a duplicate request if it already processed that transaction ID.

If the policy system transaction ID matches a previous request, BillingCenter ignores the duplicate request. This behavior is intended to recover from messaging issues, temporary integration issues, or special error conditions on the policy system.

The caller of these web service APIs must carefully create these transaction IDs to ensure all of the following:

- Policy system transaction IDs must be unique compared to all previous transaction IDs.
- Policy system transaction IDs must be unique compared to all transaction IDs generated from other systems.
- Transaction IDs must contain an initial alphabetic (letters-only) prefix, then a hyphen, and then some other unique ID. The prefix indicates the system that generated the ID, to prevent namespace conflicts. Guidewire strictly reserves all 2-letter prefixes, so you must not use such prefixes. Write your own transaction ID generator class that you can call from your own web services client code that needs a transaction ID:

```
String transactionID = MyTransactionIDGenerator.getNewTransactionID();
```

Guidewire recommends that within the transaction ID after the hyphen that you use a guaranteed unique sequence generator to generate an ID. Additionally, consider appending the date stamp and time stamp also. This can help diagnose integration issues and more easily spot anomalies. The built-in integrations between Guidewire applications use both these approaches.

For example, if your company name was ABC, your ID might be:

```
abc-2009-12-7-0207pm-123278
```

IMPORTANT Your transaction ID must be a prefix, followed by a hyphen, followed by a unique ID. Guidewire strictly reserves all two-letter prefixes for itself, such as bc, pc, cc, and capital letter versions.

For more information, see “Checking for Duplicate External Transaction IDs” on page 51.

Account APIs in BillingAPI

The `BillingAPI` web service includes the following methods that act on accounts:

- “Search for Accounts” on page 90
- “Get Account Invoice Streams” on page 90
- “Get All Subaccounts” on page 90
- “Check If an Account Exists” on page 91
- “Create an Account from an External System” on page 91
- “Update an Account” on page 91

See also

- Many of the account APIs in `BillingAPI` use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Search for Accounts

To search for accounts from an external system, call the `BillingAPI` web service method `searchForAccounts`.

Note: This method uses Gosu class DTOs without the `DTO` suffix.

The `searchForAccounts` method takes a search criteria object as a `BCAccountSearchCriteria` DTO. The properties include:

- `AccountNumber` – account number
- `AccountName` – account name
- `AccountNameKanji` – account name in Kanji
- `IsListBill` – a Boolean that specifies whether to restrict to list bill accounts only
- `Currency` – a `Currency` typecode, represented as a `String`.

The method also takes an `Integer` argument that represents the maximum number of results to return. If this value is `null`, BillingCenter uses a default limit of 50.

The method returns account search results as an array of `BCAccountSearchResult` objects. Each `BCAccountSearchResult` is a Gosu class DTO that contains an account number, account name, and the name of the primary payer.

Get Account Invoice Streams

To get invoice streams associated with an account, call the `BillingAPI` web service method `getAccountInvoiceStreams`.

Note: This method uses Gosu class DTOs without the `DTO` suffix.

The `getAccountInvoiceStreams` method takes as arguments an account number and an optional `Currency` value. The currency code is a `Currency` typecode as a `String`. If the currency value is `null`, the method uses the identified account to determine the currency.

The method returns results as an array of `InvoiceStreamInfo` DTO objects. `InvoiceStreamInfo` DTO object encapsulates data from the `InvoiceStream` entity type. Refer to the *Data Dictionary* for the `InvoiceStream` entity type for details of each property.

If the account does not exist, the method returns an empty list rather than throwing an exception.

Get All Subaccounts

To get all subaccounts of an account, call the `BillingAPI` web service method `getAllSubAccounts`. The search is recursive and returns all subaccounts under the hierarchy (tree) of accounts.

The only `getAllSubAccounts` method argument is an account number.

The method returns account search results as an array of `BCAccountSearchResult` objects. Each `BCAccountSearchResult` is a Gosu class DTO that contains an account number, account name, and the name of the primary payer. If the account number specified in the method's argument does not exist, the method returns an empty array.

Check If an Account Exists

To check if an account exists, call the `BillingAPI` web service method `isAccountExist`.

Create an Account from an External System

To create an account in BillingCenter from an external system, call the `BillingAPI` web service method `createAccount`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The method takes the following arguments:

- `PCAccountInfo` DTO – Represents data in a `PCAccountInfo` entity. This DTO includes optional links to other DTOs such as `PCContactInfo`. You could set the account number in the `PCAccountInfo` object optionally. If you do not set the account number, the method creates a new account number, which is also the method return value.
- Currency typecode, as a `String` – The currency for the account.
The default implementation of multicurrency integration with PolicyCenter sets the value to the preferred settlement currency of the PolicyCenter account.
- Transaction ID, as a `String`

Most of the necessary work for using the `createAccount` method is populating properties in the `PCAccountInfo` DTO. If a billing level is not specified then policy-level billing with cash separation is assigned by default.

The method returns a `String` that represents the account number (not the public ID) of the new account. The default implementation of multicurrency integration with PolicyCenter returns the account number for what will become the primary affiliated account if subsequently policies with alternative currencies are bound.

See also

- “Policy System Transaction IDs in BillingAPI” on page 89
- “Multicurrency Integration between BillingCenter and PolicyCenter” on page 405

Update an Account

To update an account from an external system, call the `BillingAPI` web service method `updateAccount`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The method takes a `PCAccountInfo` DTO. In the DTO, set the account number and any other properties that you want to change, leaving others blank.

The method returns the account public ID as a `String`.

Policy Period APIs in BillingAPI

The `BillingAPI` web service includes the following methods that act on policy periods:

- “Issue a New Policy Period” on page 92
- “Change a Policy” on page 92
- “Transfer Policy Periods” on page 93

- “Cancel a Policy” on page 93
- “Issue a Reinstatement” on page 94
- “Issue a Premium Report” on page 94
- “Renew a Policy Period” on page 94
- “Confirm a Policy Period” on page 95
- “Rewrite an Existing Policy Period” on page 95
- “Add Collateral Charges” on page 96
- “Get Policy Period Details” on page 96

See also

- Many of the policy period APIs in BillingAPI use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

[Issue a New Policy Period](#)

An external system can tell BillingCenter about a policy issuance (and its billing implications) with the BillingAPI web service method `issuePolicyPeriod`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main argument for the method is a `IssuePolicyInfo` DTO object, which represents data in the billing instruction entity `Issuance`. For the list of all properties, see:

- The `entity.xsd` file definitions of `IssuePolicyInfo`, `BillingInstructionInfo`, and `PCPolicyPeriodInfo`.
- The *Data Dictionary* for the `Issuance` entity.

The method returns the public ID for the new policy period.

The `IssuePolicyInfo` object contains the following key properties:

- `AccountNumber` – Account number, as a `String`
- `AssignedRisk` – Assigned risk, `boolean`
- `BillingMethodCode` – Billing method code, as a `String`
- `ModelDate` – Model date, as a `java.util.Calendar`
- `OfferNumber` – Offer number, as a `String`
- `PaymentPlanPublicId` – Payment plan public ID, as a `String`
- `ProducerCodeOfRecordId` – Producer code of record public ID, as a `String`. The specified ID must exist or the method throws a `BadIdentifierException`.
- `ProductCode` – Product code, as a `String`
- `TermNumber` – Term number, as an `int`
- `UWCompanyCode` – Underwriting company code, as a `String`

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

[Change a Policy](#)

An external system can tell BillingCenter about a policy change and its billing implications with the BillingAPI web service method `changePolicyPeriod`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main argument for the method is a `PolicyChangeInfo` object. It is a DTO that represents data in the billing instruction entity `PolicyChange`. For the list of all properties, see:

- The `entity.xsd` file definitions of `PolicyChangeInfo`, `BillingInstructionInfo`, and `PCPolicyPeriodInfo`.
- The *Data Dictionary* for the `PolicyChange` entity.

This method returns the public ID of the new billing instruction for the policy change.

At the payment page of the PolicyCenter policy change wizard, there is an option called **Override Billing Allocation**. BillingCenter does not support this feature in BillingCenter-PolicyCenter integration in the default configuration.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Transfer Policy Periods

An external system can instruct BillingCenter to transfer policy periods to a different account with the `BillingAPI` web service method `transferPolicyPeriods`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The `transferPolicyPeriods` method takes the following main arguments:

- An array of one or more `PCPolicyPeriodInfo` DTO objects. Each DTO identifies which policy period to transfer. For the search algorithm, see “Identifying Policy Periods in BillingAPI” on page 86.
- The account number of the destination account

For each policy period, if the BillingCenter destination account currency matches the currency of the policy period, BillingCenter directly transfers the policy period to that account.

However, if the BillingCenter destination account currency does not match the currency of the policy period, BillingCenter instead transfers to the sibling account for that currency. If a sibling account for that currency does not yet exist, first BillingCenter creates the sibling account for that currency. Next, BillingCenter transfers the policy period to the new sibling account.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Cancel a Policy

An external system can tell BillingCenter about a cancellation and its billing implications with the `BillingAPI` web service method `cancelPolicyPeriod`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main argument is a `CancelPolicyInfo` DTO. `CancelPolicyInfo` is a DTO that represents the billing instruction entity `Cancellation`. For this API, the required properties in the DTO are:

- `TermNumber`
- One of the following:
 - `PCPolicyPublicID`
 - `PolicyNumber`
- `EffectiveDate`

For the list of all properties, see:

- The `entity.xsd` file definitions of `CancelPolicyInfo`, `BillingInstructionInfo`, and `PCPolicyPeriodInfo`.
- The *Data Dictionary* for the `Cancellation` entity.

This method returns the public ID of the new billing instruction entity instance.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Issue a Reinstatement

After a cancellation billing instructions, you might need to create a billing instruction for *reinstatement* of a policy. A reinstatement reverses or undoes the cancellation a policy. An external system can tell BillingCenter about a reinstatement (and its billing implications) with the `BillingAPI` web service method `reinstatePolicyPeriod`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main argument is a `ReinstatementInfo` DTO. `ReinstatementInfo` is a DTO that represents the billing instruction entity `Reinstatement`. For the list of all properties, see:

- The `entity.xsd` file definitions of `ReinstatementInfo`, `BillingInstructionInfo`, and `PCPolicyPeriodInfo`.
- The *Data Dictionary* for the `Reinstatement` entity.

This method returns the public ID of the new billing instruction entity instance.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Issue a Premium Report

An external system can tell BillingCenter about a premium report and its billing implications with the `BillingAPI` web service method `issuePremiumReport`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main argument is a `PremiumReportInfo` DTO. `PremiumReportInfo` is a DTO that represents the billing instruction entity `PremiumReportBI`. For the list of all properties, see:

- The `entity.xsd` file definitions of `PremiumReportInfo`, `BillingInstructionInfo`, and `PCPolicyPeriodInfo`.
- The *Data Dictionary* for the `PremiumReportBI` entity.

This method returns the public ID of the new billing instruction entity instance.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Renew a Policy Period

An external system can tell BillingCenter about a renewal and its billing implications with the `BillingAPI` web service method `renewPolicyPeriod`. This method creates either a `Renewal` or `NewRenewal` billing instruction, depending on whether the policy already exists.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

This method performs differently depending on whether the specified policy period already exists:

- If this policy period already exists, BillingCenter renews the policy period with the standard `Renewal` billing instruction.
- Otherwise, BillingCenter issues a `NewRenewal` billing instruction. Use a `NewRenewal` billing instruction to renew policies previously managed by a legacy billing system other than BillingCenter. A `NewRenewal` creates a policy that is new to BillingCenter.

The main method argument is a `RenewalInfo` DTO. `RenewalInfo` is a DTO that represents the billing instruction entity `Renewal`. For the list of all properties, see:

- The `entity.xsd` file definitions of `RenewalInfo`, `BillingInstructionInfo`, and `PCPolicyPeriodInfo`.
- The *Data Dictionary* for the `Renewal` entity.

This method returns the public ID of the new billing instruction entity instance.

IMPORTANT When storing the public ID in an external system, remember that the entity type may be either `Renewal` or `NewRenewal`.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Confirm a Policy Period

There are several different options you can use for renewal flows between your policy administration system and BillingCenter. If you use *confirmed renewals*, the `PolicyPeriod.TermConfirmed` property becomes relevant. The property indicates whether the customer has met the conditions for the policy term/period to be officially bound. A typical condition is receipt of a payment. Other conditions might include required actions, such as a formal refusal of a certain type of coverage.

An external system can update a `BillingCenter.PolicyPeriod.TermConfirmed` property by calling the `BillingAPI` web service method `updatePolicyPeriodTermConfirmed`. The method accepts the following arguments.

- Policy number
- BillingCenter policy period number
- The boolean value to assign to the `PolicyPeriod.TermConfirmed` property

Rewrite an Existing Policy Period

An external system can tell BillingCenter about a policy rewrite and its billing implications with the `BillingAPI` web service method `rewritePolicyPeriod`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main method argument is a `RewriteInfo` DTO. `RewriteInfo` is a DTO that represents the billing instruction entity `Rewrite`.

Because of the similarity of rewrite and renewal, the `RewriteInfo` DTO contains the same properties as the DTO used for the similar API for renewal, the `RenewalInfo` object. For the list of all properties, see:

- The `entity.xsd` file definitions of `RewriteInfo`, `BillingInstructionInfo`, and `PCPolicyPeriodInfo`.
- The *Data Dictionary* for the `Rewrite` entity.

This method returns the public ID of the new billing instruction entity instance.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Add Policy Period General Charges

To add general charges like fees to a policy period, an external system can call the `BillingAPI` web service method `addPolicyPeriodGeneralCharges`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main method argument is a `PolicyPeriodGeneralInfo` DTO. `PolicyPeriodGeneralInfo` is a DTO that represents the billing instruction entity `General1`. For the list of all properties, see:

- The `entity.xsd` file definitions of `PolicyPeriodGeneralInfo`, `BillingInstructionInfo`, and `PCPolicyPeriodInfo`.

- The *Data Dictionary* for the **General** entity.

Identify the policy period using the following required properties:

- Either of the following: **PolicyNumber** or **PCPolicyPublicID** (the public ID of the PolicyCenter Policy)
- **TermNumber**

If the charge category in the DTO is not **Fee** or **General**, the method throws an exception.

The method returns the public ID of the new billing instruction entity instance.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Add Collateral Charges

For Collateral or unsegregated CollateralRequirement

To add collateral charges to a **Collateral** or unsegregated **CollateralRequirement**, call the **BillingAPI** web service method **addCollateralCharges**.

Note: This method uses XML DTOs with declarations in the XSD file **entity.xsd**.

The main method argument is a **CollateralInfo** DTO containing information for creating the **CollateralBI** for an existing **Collateral** or **CollateralRequirement**. The method returns the public ID of the created **CollateralBI**.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

For segregated CollateralRequirement

To add collateral charges to a segregated **CollateralRequirement**, call the **BillingAPI** web service method **addSegregatedCollateralCharges**.

Note: This method uses XML DTOs with declarations in the XSD file **entity.xsd**.

The main method argument is a **CollateralInfo** DTO containing information for creating the **SegregatedCollReqBI** for an existing **CollateralRequirement**. The method returns the public ID of the created **SegregatedCollReqBI**.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Get Policy Period Details

An external system can retrieve a full policy period including current payment plan and billing method using the **BillingAPI** web service method **getPolicyPeriod**.

Note: This method uses XML DTOs with declarations in the XSD file **entity.xsd**.

The **PCPolicyPeriodInfo** is a DTO that represents the root billing instruction class **BillingInstruction**, with extra properties used by various APIs. The method uses the following properties to identify the policy: **PCPolicyPublicID**, **TermNumber**, and **PolicyNumber**. For the search algorithm, see “Identifying Policy Periods in BillingAPI” on page 86.

After BillingCenter finds the policy period, BillingCenter extracts the payment plan and the billing method of the policy period.

The method sets the following fields in the result DTO object of type **IssuePolicyInfo**:

- **PaymentPlanPublicId** – Payment plan public ID
- **BillingMethodCode** – Indicates either agency bill or direct bill:

- For agency bill, the `String` value "AgencyBill"
- For direct bill, the `String` value "DirectBill"
- `AltBillingAccountNumber` – alternate billing account number, if it exists
- `InvoiceStreamId` – invoice stream ID
- `Currency` – the currency code, a `Currency` typecode as a `String`

Producer APIs in BillingAPI

The `BillingAPI` web service includes the following methods that act on producers:

- “Get Billing Methods for a Producer” on page 97
- “Create or Update a Producer” on page 97
- “Create or Update a Producer Code” on page 98
- “Check if a Producer or Producer Code Exists” on page 99
- “Get Billing Methods for a Producer” on page 97

See also

- Many of the producer APIs in `BillingAPI` use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.
- “Multicurrency Integration between BillingCenter and PolicyCenter” on page 405

[Get Billing Methods for a Producer](#)

An external system can get the list of all billing methods that a producer supports with the `BillingAPI` web service method `getAvailableBillingMethods`. For example, suppose you use a policy administration system to enter a submission. The policy system calls this `BillingCenter` method to request a list of billing methods from the billing system. Next, the policy system could offer a user-selectable list of available billing methods, or decide programmatically which to use for the submission.

The method takes a producer code public ID (a `String`) and a `Currency` typecode.

The method returns a list of the billing methods that the specified producer supports. The list is in the form of an array of `String` values. Possible `String` values are: “`DirectBill`” and “`AgencyBill`”.

If the method cannot find the producer code, it throws a `BadIdentifierException`. If more than one producer code is associated with the specified producer code public ID, the method throws a `ServerStateException`.

[Create or Update a Producer](#)

The `BillingAPI` web service has methods to create new producers or update existing producers from an external system.

Note: The producer code methods that follow use XML DTOs with declarations in the XSD file `entity.xsd`.

[Create a New Producer](#)

To create a new producer in `BillingCenter` from an external system, call the `BillingAPI` web service method `createProducer`.

The main method argument is a `PCNewProducerInfo` DTO. The `PCNewProducerInfo` DTO is which is a subclass of `PCProducerInfo` that requires a `PreferredCurrency`. This DTO represents information in the `BillingCenter` `Producer` entity:

- `PrimaryContact` – optional (nullable) property is a primary contact in a `PCCContactInfo` DTO.

- PreferredCurrency – The currency that the producer prefers to do business in. Currency code is `Currency` typecode as a `String`.
- Currencies – optional (nullable) property with a list of currencies that the producer supports
- AgencyBillPlanIDs – optional (nullable) property with a list of bill plan public IDs of the `AgencyBillPlan` objects that the Producer uses. Any duplicate IDs in the list are ignored. These must use a supported currency.
- Tier – optional (nullable) tier of the producer, as a `String`

The method returns the public ID (a `String`) for the new producer.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Update an Existing Producer

To update an existing producer call the `BillingAPI` web service method `updateProducer`. The one argument has the type `PCProducerInfo`. The `updateProducer` method uses the public ID property (`PCProducerInfo.PublicID`) to identify which producer to update. Next, the `updateProducer` method updates the other producer properties using properties in the `PCProducerInfo` object.

The `PCProducerInfo` DTO object represents information in the `BillingCenter Producer` entity:

- PrimaryContact – optional (nullable) property is a primary contact in a `PCContactInfo` DTO.
- PreferredCurrency – optional (nullable) property. The currency that the producer prefers to do business in. Currency code is `Currency` typecode as a `String`.
- Currencies – optional (nullable) property with a list of currencies that the producer supports
- AgencyBillPlanIDs – optional (nullable) property with a list of bill plan public IDs of the `AgencyBillPlan` objects that the Producer uses. Any duplicate IDs in the list are ignored. These must use a supported currency.
- Tier – optional (nullable) tier of the producer, as a `String`

The method returns the public ID (a `String`) for the updated producer.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

[Create or Update a Producer Code](#)

The `BillingAPI` web service has methods to create new producer codes and update existing producer codes from external system.

Note: The producer code methods that follow use XML DTOs with declarations in the XSD file `entity.xsd`.

Create a New Producer Code

To create a new producer code from an external system, call the `BillingAPI` web service method `createProducerCode`.

The main method argument is a `NewProducerCodeInfo` DTO. This DTO represents information in the `BillingCenter ProducerCode` entity. This DTO is based on the related `ProducerCodeInfo` DTO.

The `NewProducerCodeInfo` is a DTO container for links to a commission plan and a producer. Set the following properties in the object:

- Active – A boolean that indicates if the producer code is active
- Code – The code string to be associated with the producer code
- ProducerPublicID – The unique public ID of a related `Producer` entity instance, as a `String`

- `CommissionPlanIDs` – An optional list of public ID values of commission plans
- `Currencies` – An optional list of one or more currencies that the producer code supports

The method returns the unique public ID (a `String`) for the new producer code.

The `createProducerCode` method throws an error under the following conditions:

- Any commission plan for a specified currency does not exist in BillingCenter
- A commission plan with a specified public ID does not exist in BillingCenter or more than one identified commission plan specifies the same currency

You can create a producer code without specifying a commission plan in single currency and multiple currency configurations. However, you must configure BillingCenter with at least one active commission plan for each supported currency. BillingCenter then chooses an existing commission plan for a new producer code if no commission plans are specified, based on the currency specified for the producer code.

In a multiple currency configuration, you can create affiliated producer codes for separate currencies without specifying a commission plan. BillingCenter chooses a commission plan for the affiliated producer codes based on the currencies specified for the producer. If no commission plan exists in BillingCenter for a specified currency, an affiliated producer code for that currency is not created. If you define more than one commission plan in a given currency, BillingCenter selects a commission plan based on plan order.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Update an Existing Producer Code

To update an existing producer code, call the `BillingAPI` web service method `updateProducerCode`.

Its main argument has type `ProducerCodeInfo`, which has the same properties as the `NewProducerCodeInfo` object that the `createProducerCode` method uses. However, the only properties on producer code that you can update with the `updateProducerCode` method are `Active` and `Code`.

In multicurrency mode, this method updates the same producer code on sibling producers.

BillingCenter uses the `ProducerCodeInfo.PublicID` property to identify which producer code to update. Next, BillingCenter uses the `Active` and `Code` properties in the `ProducerCodeInfo` object to update the producer code.

The method returns the public ID (a `String`) for the new producer code.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

Check if a Producer or Producer Code Exists

To check from an external system if a producer exists, call the `BillingAPI` web service method `isProducerExist`. The method takes one argument that is a producer public ID as a `String`. The method returns `true` if and only if the producer already exists in BillingCenter.

Similarly, to check if a producer code exists for a producer, call the `BillingAPI` web service method `isProducerCodeExist`. The method takes a producer public ID (as a `String`) and a code (as a `String`). The method returns `true` if and only if the producer code exists for that producer in BillingCenter.

Get Producer Codes Commission Plans

To get commission plans from an external system, call the `BillingAPI` web service method `getProducerCodeInfo`. Pass a producer code public ID as an argument. The method returns a `ProducerCodeInfo` object that contains a property called `CommissionPlanInfos`. That property contains a list of `CommissionPlanInfo` objects associated with the producer code.

Payment Plan APIs in BillingAPI

The `BillingAPI` web service includes the following methods that act on payment plans:

- “Preview a Payment Plan” on page 100
- “Get All Payment Plans” on page 101
- “Get Payment Plans for an Account” on page 101

See also

- The payment plan APIs in `BillingAPI` use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Preview a Payment Plan

An external system can preview the payment plan that BillingCenter would create for a new policy or a policy change using one of two methods of the `BillingAPI` web service. For example, an insurance company policy system or web portal can request BillingCenter to determine the payment plan if the insured binds the policy. BillingCenter uses its rules to generate the plan, but BillingCenter does not save the previewed data to its database.

Note: These methods use XML DTOs with declarations in the XSD file `entity.xsd`.

The two method names vary only slightly in spelling:

- For a new policy, call the `previewInstallmentsPlanInvoices` method. Its main argument is a `IssuePolicyInfo` DTO.
- For a policy change, call the similarly-named `previewInstallmentPlanInvoices` method. Its main argument is a `PolicyChangeInfo` DTO.

Both methods return an array of invoice preview DTOs of type `InvoiceItemPreview`. Use each invoice preview item to examine the installment schedule. Each `InvoiceItemPreview` object has the following properties:

- `Amount` – Amount, as a `BigDecimal`
- `ChargeName` – Charge name, as a `String`
- `InvoiceDate` – Invoice date, as a `java.util.Calendar` object
- `InvoiceDueDate` – Invoice due date, as a `java.util.Calendar` object
- `Type` – the invoice type, as an `InvoiceItemType` typecode such as:
 - `commissionadjustment` – Commission adjustment from a change in `Commission`
 - `commissionremainder` – Commission remainder from distribution across invoice items
 - `deposit` – the down payment
 - `depositadjustment` – Down payment adjustment from change in the payment plan
 - `depositdunning` – Down payment on separate invoice with dunning due date
 - `depositseparate` – Down payment on separate invoice with standard due date
 - `installment` – Installment
 - `onetime` – One-time payment

Get All Payment Plans

An external system can get the list of all currently available payment plans with the BillingAPI web service method `getAllPaymentPlans`. For example, suppose you use a policy administration system to enter a submission. The policy system can call this BillingCenter method to request a list of payment plans from the billing system. Next, the policy system could offer a user-selectable list of available payment plans, or decide programmatically which to use for the submission.

IMPORTANT This method returns only the plans that are available at the current date.

The method takes no arguments.

The method returns an array of `PaymentPlanInfo` objects. This object has `PublicID` and `Name` properties like the bill plans. The `AllowedPaymentMethods` property will include only payment methods relevant for recurring payments, such as `ACH`, `CreditCard`, and `Responsive`. Additionally, the `PaymentPlanInfo` DTO has the `Reporting` property, which is a boolean that specifies whether the payment plan is a reporting plan.

The following `PaymentPlanInfo` properties are unpopulated in the BillingCenter base configuration: `Total`, `DownPayment`, `Installment`, and `Notes`.

Get Payment Plans for an Account

To get the available payment plans for an account and optionally filtered by the given currency, call the BillingAPI web service method `getPaymentPlansForAccount`. The method returns only the payment plans that are available and effective on the current date.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The method arguments are an account number (`String`) and a currency typecode (a `Currency` typecode as a `String`) object. If the currency argument is non-null, the method only displays payment plans for that currency. If the currency argument is `null`, the method returns all payment plans for the account.

The method returns an array of `PaymentPlanInfo` DTO objects. The `PaymentPlanInfo` DTO represents data in a `PaymentPlan` entity instance. Refer to the *Data Dictionary* for details of the entity. However, the DTO properties have many differences from the entity. DTO properties include:

- `Reporting` – is the payment plan reporting
- `DownPayment` – the down payment, a decimal value
- `Installment` – the installment payment, a decimal value
- `Total` – the total, a decimal value
- `Notes` – notes on the payment plan, as a `String`
- `InvoiceFrequency` – frequency of the payment plan, as a `String`
- `AllowedPaymentMethods` – a list of `String` objects, one for each payment method. Only payment methods relevant for recurring payments, such as `ACH`, `CreditCard`, and `Responsive`, are included.
- `Currency` – currency of the payment plan, a `Currency` typecode as a `String`.

If you need to customize this service, note that there is a Gosu enhancement that enhances the DTO type. In Studio, refer to the enhancement type `PaymentPlanInfoEnhancement`.

Payment Allocation Plans API in BillingAPI

For an external system to get a list of all payment allocation plans, call the BillingAPI web service method `findAllPaymentAllocationPlans`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The method takes no arguments.

The method returns an array of `PaymentAllocationPlanInfo` DTO objects. This DTO represents data in a `PaymentAllocationPlan` entity instance. The DTO simpler than the entity type and contains primarily the following properties:

- `PublicID` – a payment allocation plan public ID (`String`)
- `Name` – a name (`String`)

See also

- For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Bill and Commission Plan APIs in BillingAPI

The `BillingAPI` web service includes the following methods that act on bill plans and commission plans:

- “Get Agency Bill Plans and Commission Plans” on page 102
- “Preview the Applicable Commission Subplan” on page 103
- “Preview Commission Rate Overrides” on page 103

See also

- The bill and commission plan APIs in `BillingAPI` use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Get Agency Bill Plans and Commission Plans

The `BillingAPI` web service contains two similar methods for getting agency bill plans and commission plans.

Note: The bill and commission plan methods that follow use XML DTOs with declarations in the XSD file `entity.xsd`.

Get Agency Bill Plans

For agency bill, an external system can get the list of all agency bill plans with the `BillingAPI` web service method `getAllAgencyBillPlans`. For example, suppose you use a policy administration system to enter a submission. The policy system can call this BillingCenter method to request a list of bill plans from the billing system. Next, the policy system could offer a user-selectable list of available bill plans, or decide programmatically which to use for the submission.

This method takes no arguments and returns an array of `AgencyBillPlanInfo` objects. These `AgencyBillPlanInfo` objects are simple DTOs with:

- Public ID (`PublicID` property)
- Plan name (`Name` property)
- Currency (Currency typecode, represented as a `String`.)

It is important to note that `AgencyBillPlanInfo` objects do not contain the full set of information in an `AgencyBillPlan` entity instance.

Get Commission Plans

Get the list of all commission plans in BillingCenter by using the `BillingAPI` web service method `getAllCommissionPlans`.

This method returns an array of `CommissionPlanInfo` DTO objects. Just as with `AgencyBillPlanInfo`, the `CommissionPlanInfo` DTO includes the properties `PublicID`, `Name`, and `Currency`. The currency code is a `Currency` typecode represented as a `String`. Additionally, the `CommissionPlanInfo` object includes an `AllowedTiers` property, which is an array of `String` values. In the base configuration, the `AllowedTiers` can be “Gold,” “Silver,” or “Bronze.”

Preview the Applicable Commission Subplan

To get the `CommissionSubPlan` that best matches a new policy period for a producer code, external systems can call the `BillingAPI` web service method `previewApplicableCommissionSubPlan`. It is important to note that this method does not persist any changes. The method only returns a preview of the eventual commission that a producer might expect.

Note: This method uses two different types of DTOs. The argument `IssuePolicyInfo` is an XML DTO with declarations in the XSD file `entity.xsd`. The return result type `CommissionSubPlanInfo` is a Gosu class DTO.

The method takes two arguments:

- A `IssuePolicyInfo` DTO, which represents a new policy period to create, including all the charges. The `IssuePolicyInfo` DTO is an XSD-based DTO that encapsulates many properties that corresponds to the billing instruction entity `Issuance`. Refer to the *Data Dictionary* for `Issuance` for additional information about the properties. If the policy period already exists based on the DTO properties `PCPolicyPublicID` and `TermNumber`, the method throws the `BadIdentifierException` exception.
- A producer code public ID (`String`).

The method returns a commission subplan in the form of a `CommissionSubPlanInfo` DTO object, which is a Gosu class DTO that represents data in a `CommissionSubPlan` entity instance. See the class in Studio for details of its properties.

Preview Commission Rate Overrides

To preview commission rate overrides for a list of charges, from an external system you can call the `BillingAPI` web service method `previewCommissionRates`. It is important to note that this method does not persist any changes to the database.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main method argument is a `IssuePolicyInfo` DTO, which represents a new policy period to create, including all the charges. The `IssuePolicyInfo` DTO is an XSD-based DTO that encapsulates many properties that corresponds to the billing instruction entity `Issuance`. Refer to the *Data Dictionary* for `Issuance` for additional information about the properties. If the policy period already exists based on the DTO properties `PCPolicyPublicID` and `TermNumber`, the method throws the `BadIdentifierException` exception.

The second argument is a `String` policy role code. In the base configuration, only the policy role `TC_PRIMARY` is supported.

The return result includes the list of charges (`ChargeInfo` objects) from the incoming policy with populated commission rate overrides.

If there is no existing `CommissionSubPlan` with the given public ID, or a policy role does not exist with the given code, the method throws a `BadIdentifierException`.

You can combine this method with the `previewApplicableCommissionSubPlan` method to preview the expected commission rates that a new policy would generate.

Final Audit APIs in BillingAPI

The BillingAPI web service provides several methods to support final audits. The methods use data transfer objects (DTOs) to represent entity type data. For important information about DTOs, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Notification of a Final Audit

When an external system schedules a final audit, the best practice recommends notifying BillingCenter of the audit by calling the BillingAPI web service method `scheduleFinalAudit`. The method prevents the policy period from being closed until the audit has completed. If the policy period was already closed, it is reopened. The closure status of the policy period is set to `OpenLocked`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

Its main argument is a `PCPolicyPeriodInfo` DTO. `PCPolicyPeriodInfo` is a non-entity mirror of the root `BillingInstruction` class with additional properties to identify the relevant policy period. The important properties are `periodID` and `PolicyNumber`. For more information, see “Identifying Policy Periods in BillingAPI” on page 86.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

The method returns the policy period’s public ID.

Issue a Final Audit Billing Instruction

An external system can instruct BillingCenter to issue an `Audit` billing instruction of subtype `FinalAudit` by calling the BillingAPI web service method `issueFinalAudit`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main argument is a `FinalAuditInfo` DTO. `FinalAuditInfo` is a DTO that represents the billing instruction entity `FinalAudit`. For the list of all properties, see:

- The `entity.xsd` file definitions of `FinalAuditInfo`, `BillingInstructionInfo`, and `PCPolicyPeriodInfo`.
- The *Data Dictionary* for the `FinalAudit` entity.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

The method returns the public ID of the issued `Audit` billing instruction.

Waive a Final Audit

When an external system determines that a final audit is not going to occur, the best practice recommends notifying BillingCenter by calling the BillingAPI web service method `waiveFinalAudit`. The method enables the policy period to be closed by changing its closure status from `OpenLocked` to `Open`. Also, any holds on charges associated with the final audit are released.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

Its main argument is a `PCPolicyPeriodInfo` DTO. `PCPolicyPeriodInfo` is a non-entity mirror of the root `BillingInstruction` class with additional properties to identify the relevant policy period. The important properties are `periodID` and `PolicyNumber`. For more information, see “Identifying Policy Periods in BillingAPI” on page 86.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

The method returns the policy period’s public ID.

Update the Written Date on a Charge in BillingAPI

For an external system to update the written date on a charge, call the `BillingAPI` web service method `updateChargeWrittenDate`. The method takes a public ID for the charge and a `java.util.Date` object to represent the new written date. The method returns nothing.

Get Account Unapplied Funds Methods in BillingAPI

The `BillingAPI` web service includes two methods, `getUnappliedFunds` and `getUnapplieds`, to get the list of account unapplied funds in a multicurrency or a single currency integration.

There are two `BillingAPI` web service methods to get a list of account unapplied funds from an external system.

These methods both take an account number as an argument. The `getUnappliedFunds` method also takes a currency as an argument.

Both methods return an array of `PCUnappliedInfo` DTO objects. An `PCUnappliedInfo` DTO object is a Gosu class that represents one `PCUnappliedInfo` entity instance. The DTO contains only a public ID (the `PublicID` property) and a description (the `Description` property).

Note: These method uses a Gosu class DTO.

The following table compares the available methods.

| To get account unapplied funds | Call this method | Arguments | Description |
|--------------------------------|--------------------------------|--------------------------------|--|
| Single Currency Integration | <code>getUnapplieds</code> | • account number | <ul style="list-style-type: none">Retrieves list of unapplied funds associated with the specified account. |
| Multicurrency Integration | <code>getUnappliedFunds</code> | • account number • currency | <ul style="list-style-type: none">Retrieves list of unapplied funds associated with the specified account and currency.Call once for each currency to retrieve all unapplied funds associated with the specified account. |

Note: If you are using BillingCenter in a multicurrency integration and you use the `BillingAPI` web service method `getUnapplieds`, you will not get the expected results. You will get a list of unapplied funds only for the currency associated with the primary affiliated account in BillingCenter. This method is useful only in a single currency integration.

See also

- For more information about multicurrency integration and affiliated accounts, see “Billing and Policy Multicurrency Account Correspondence” on page 417 in the *Application Guide*.
- For important information about data transfer objects (DTOs) generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Contact API in BillingAPI

For an external system to update contact information on a policy, call the `BillingAPI` web service method `updateContact`.

Note: This method uses XML DTOs with declarations in the XSD file `entity.xsd`.

The main method argument is a `PCContactInfo` DTO. `PCContactInfo` is a DTO that represents the contact entity type called `Contact`. The most important property is the public ID property `PCContactInfo.PublicID`. BillingCenter first uses the public ID to find the contact you want to update. Then, BillingCenter uses the other properties in the `PCContactInfo` object to update that contact with new information.

The method returns the public ID (`String`) of the updated contact entity instance.

The method takes an additional transaction ID argument. See “Policy System Transaction IDs in BillingAPI” on page 89.

See also

- For important information about data transfer objects (DTOs) generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Invoice Details Web Service APIs (InvoiceDetailsAPI)

For an external system to get invoice details, call methods on the `InvoiceDetailsAPI` web service. Some methods let external systems get invoices for accounts, policy periods, or invoice streams. Other methods let external systems get individual invoice items on invoices or policy periods.

See also

- The invoice details APIs use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Get Invoices Methods

There are multiple `InvoiceDetailsAPI` web service methods to get invoices from an external system. All the methods return an array of `InvoiceDTO` objects. An `InvoiceDTO` object is a DTO that represents data in an `Invoice` entity instance. For all methods, the array is ordered by bill date (the event date).

These methods take as an argument an optional invoice status as a `InvoiceStatus` typecode. If `null`, the method returns invoices with any status. Otherwise, returns only items with the given status.

Note: The get invoices methods return Gosu class DTOs with the `DTO` suffix.

The following table compares the available methods.

| To get invoice items from this object | Ordered by | Call this method | Arguments |
|---------------------------------------|------------|---|---|
| account | bill date | <code>getInvoicesOnAccountSortedByBillDate</code> | <ul style="list-style-type: none"> invoice public ID optional invoice status |
| policy period | bill date | <code>getInvoicesOnPolicyPeriodSortedByBillDate</code> | <ul style="list-style-type: none"> policy period public ID optional invoice status |
| invoice stream | bill date | <code>getInvoicesOnInvoiceStreamSortedByBillDate</code> | <ul style="list-style-type: none"> invoice stream public ID optional invoice status |

Get Invoice Items Methods

There are multiple `InvoiceDetailsAPI` web service methods to get invoice items from an external system. All the methods return an array of `InvoiceItemDTO` objects. An `InvoiceItemDTO` object is a DTO that represents data in an `InvoiceItem` entity instance.

Note: The get invoice items methods return Gosu class DTOs with the `DTO` suffix.

The following table compares the available methods.

| To get invoice items from this object | Ordered by | Call this method | Arguments |
|---------------------------------------|------------|--|-------------------------|
| invoice | unordered | getInvoiceItemsOnInvoice | invoice public ID |
| policy period | bill date | getInvoiceItemsOnPolicyPeriodSortedByEventDate | policy period public ID |

Payments Web Service APIs (PaymentAPI)

For an external system to make payments, reverse payments, and manipulate producer unapplied accounts, call methods on the `PaymentAPI` web service.

This topic contains:

- “Payment Receipt Record DTOs” on page 107
- “Make a Direct Bill Payment” on page 108
- “Make a Suspense Payment” on page 109
- “Reverse a Payment” on page 109
- “Make an Account Balance Adjustment” on page 110
- “Producer Unapplied Methods” on page 110
- “Make an Agency Payment” on page 111

See also

- For information on how to securely handle payment instruments such as credit cards, see “Payment Instrument Web Service APIs (PaymentInstrumentAPI)” on page 112.

Payment Receipt Record DTOs

Various methods in the `PaymentAPI` web service take arguments of the `PaymentReceiptRecord` DTO type.

The `PaymentReceiptRecord` DTO is a Gosu class that represents properties on the `PaymentReceipt` entity delegate type and all entity types that implement the `PaymentReceipt` delegate. For example, the suspense payment (`SuspensePayment`) entity type implements the `PaymentReceipt` delegate. Therefore, the `PaymentReceiptRecord` DTO includes relevant properties that are specific to suspense payments.

Other notes about the `PaymentReceiptRecord` DTO:

- The DTO contains a `PaymentReceiptType` property that contains an enumeration (defined in that class) that indicates the billing type, such as `AGENCYBILLMONEYRECEIVED`, `DIRECTBILLMONEYDETAILS`, `SUSPENSEPAYMENT`.
- The DTO contains a `PaymentInstrumentRecord` property that contains a `PaymentInstrumentRecord` DTO. The `PaymentInstrumentRecord.PaymentMethod` property defines the payment method, such as cash or check.
- See the *Data Dictionary* for more details about `PaymentReceipt`.
- For the full list of properties on the DTO, see the `PaymentReceiptRecord` Gosu class in Studio.
- The DTO is a Gosu class DTO that does not have the suffix DTO.

See also

- For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Make a Direct Bill Payment

To make a direct bill payment from an external system, call the `PaymentAPI` web service methods `makeDirectBillPayment` or `makeDirectBillPaymentToPolicyPeriod` or `makeDirectBillPaymentToInvoice`.

Make a Direct Bill Payment to an Account

To make a direct payment to an account, call the `PaymentAPI` web service method `makeDirectBillPayment`. The method takes a single parameter, which is the money details as a `PaymentReceiptRecord` DTO. See “Payment Receipt Record DTOs” on page 107. The `PaymentReceiptRecord.PaymentReceiptType` property must have the enumeration value `DirectBillMoneyDetails`.

BillingCenter distributes the payment to the policies on the account using standard payment distribution. BillingCenter adds the amount to the account’s default Unapplied T-account, regardless of the account’s billing level. (See “Billing Levels and Unapplied T-Accounts” on page 302 in the *Application Guide* for further details.) Later, the batch processes `NewPayment` and `AutoDisbursements` distribute the money. Note that an account may or may not have multiple policies. For details of the distribution algorithm, see “Distributing Direct Bill Payments” on page 293 in the *Application Guide*.

The method returns the public ID (`String`) of the new payment.

Make Direct a Bill Payment to a Policy

To make a direct payment to a policy period, call the `PaymentAPI` web service method `makeDirectBillPaymentToPolicyPeriod`.

The method takes two parameters:

- The money details as a `PaymentReceiptRecord` DTO. See “Payment Receipt Record DTOs” on page 107. The `PaymentReceiptRecord.PaymentReceiptType` property must have the enumeration value `DirectBillMoneyDetails`.
- A policy period public ID (a `String`)

BillingCenter distributes the payment to the policy using standard payment distribution. If extra money remains from the payment, BillingCenter reads the account payment option property (`Account.PolicyLevelPaymentOption`) to determine how to handle this situation. The payment option property values contains one of the values of the `PolicyLevelPaymentOption` typelist:

- `distributeall` – Do not apply policy level payment to the specific policy. Instead, use account level distribution.
- `distributeextra` – Apply the portion of the payment necessary to cover the amount due on the policy. Distribute the remainder using account level distribution.

For details of the distribution algorithm, see “Distributing Direct Bill Payments” on page 293 in the *Application Guide*.

The method returns the public ID (`String`) of the new payment.

Make a Direct Bill Payment to an Invoice

To make a direct bill payment to an invoice, call the `PaymentAPI` web service method `makeDirectBillPaymentToInvoice`.

The method takes two parameters:

- the money details as a `PaymentReceiptRecord` DTO. See “Payment Receipt Record DTOs” on page 107. The `PaymentReceiptRecord.PaymentReceiptType` property must have the enumeration value `DirectBillMoneyDetails`.
- an invoice public ID (a `String`) – Optional. (see note below)

If the invoice public ID is non-null, BillingCenter makes the payment to that specific `Invoice`.

If the invoice public ID is `null`, BillingCenter distributes the payment to the invoices of the account using standard payment distribution. For details of the distribution algorithm, see “Distributing Direct Bill Payments” on page 293 in the *Application Guide*.

The method returns the public ID (`String`) of the new payment.

Make a Suspense Payment

A suspense payment is a payment for an account or policy that does not yet exist in BillingCenter. Without an existing account or policy, BillingCenter cannot determine where to distribute the payment. The nonexistent account or policy will be created presumably in the near future. In the meantime, BillingCenter can accept the payment and store it in a temporary suspense account by making a suspense payment.

Typically, you invoke make a suspense payment an attempt to make `makeSuspensePayment` after a call to another `PaymentAPI` method fails because of a nonexisting account or policy number. To make a suspense payment from an external system, call the `PaymentAPI` web service method `makeSuspensePayment`. If your external system is certain that the account or policy does not exist in BillingCenter, call `makeSuspensePayment` immediately.

The `makeSuspensePayment` method takes a single argument of the `PaymentReceiptRecord` DTO type. You must specify an amount, a payment date, a payment method, a check number, and an invoice number. In addition, three semi-optional fields exist in `PaymentReceiptRecord`. One, and only one, of these three fields must be specified when calling the method. The fields are:

- **Account number**
- **Policy number**
- **Offer number**

After the relevant account or policy is created in BillingCenter, Suspense Payment batch processing associates the suspense payment with its account or policy.

Before your code calls the `makeSuspensePayment` method after it catches an exception on another `PaymentAPI` method, confirm that the payment date and amount are valid. The exception can mask other validation errors, such as an invalid payment date or amount. A subsequent call to `SuspensePayment` will fail for the same reasons.

See also

- “Payment Receipt Record DTOs” on page 107.
- For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Reverse a Payment

You can instruct BillingCenter to reverse an incoming payment from an external system, for example to handle insufficient funds (NSF) for a check payment from an external system. To reverse a payment from an external system, call the `PaymentAPI` web service method `reverseDirectBillPayment`.

Pass to this method two arguments:

- the public ID (`String`) of a payment, which is an entity instance of type `DirectBillMoneyRcvd`
- a reason code typecode from the `PaymentReversalReason` typelist:
 - `accountclosed` – account closed
 - `returnedcheck` – insufficient funds

The method returns nothing.

Make an Account Balance Adjustment

If an account has Unapplied funds, the funds can be processed as an account adjustment. An external system initiates the account adjustment by calling the `PaymentAPI` web service `makeAccountAdjustment` method.

WARNING This web service method triggers resource-intensive code on the server. On a production server, use caution because of the implications on server performance. If frequent or complex reporting is required, use a separate reporting system rather than this web service method.

BillingCenter records the adjustment as a transaction, not a charge or distribution. The adjustment functions like a Negative Write-off transaction.

A typical use case for an account adjustment occurs during integration testing. A dummy account might be set up to temporarily hold misallocated funds stored in BillingCenter for the sole purpose of recording them in the General Ledger system. In such a scenario, funds are transferred to the dummy account, and the integration code posts the information to the General Ledger system. Finally, the funds are cleaned up by making an account adjustment with this method.

The method accepts the following parameters.

- The public ID string of an account
- The adjustment amount expressed as a negative value of the type `MonetaryAmount`. The amount must be less than zero or the method throws a `SoapException`. If the specified amount is greater than the funds stored in the account's Unapplied fund, the amount is adjusted to equal the Unapplied amount.
- The public ID of a policy period (optional). If specified, the transaction processes the funds from the policy's designated Unapplied fund. If the designated Unapplied fund does not exist, the method throws a `SoapException`. If the parameter is null, the transaction processes the funds from the account's default Unapplied fund. In either case, if the applicable Unapplied fund is empty or has a negative balance, the method throws a `SoapException`.

The method returns the actual amount processed as type `MonetaryAmount`.

Producer Unapplied Methods

To trigger various actions regarding unapplied payments from an external system, call the producer unapplied methods on the `PaymentAPI` web service. The producer unapplied fund is a T-account that represents incoming payment targeted for a producer but not yet assigned to a specific invoice.

See also

- The producer unapplied methods use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Making a Payment to Producer Unapplied

To make a payment to a producer unapplied account, call the `PaymentAPI` web service method `payToProducerUnapplied`. This method takes the money details as a `PaymentReceiptRecord` DTO.

Note: This method uses a Gosu class DTO without the suffix DTO.

The method returns the unapplied amount of the producer as a `String` object for display, not as a `BigDecimal` or `MonetaryAmount` DTO object.

See also

- See “Payment Receipt Record DTOs” on page 107.

Get Producer Unapplied

To get the balance of a producer unapplied T-account, call the `PaymentAPI` web service method `getProducerUnapplied`. The one method argument is the public ID (`String`) of a producer.

The method returns the producer unapplied amount as a `MonetaryAmount` DTO object.

Write Off Producer Unapplied

To create a producer write-off to the producer's unapplied fund, call the `PaymentAPI` web service method `writeoffProducerUnapplied`. The method arguments are:

- Public ID of a producer
- Write-off amount, as a `MonetaryAmount` DTO object

If you pass a positive value for the write-off, it is a regular write-off, which reduces a positive balance in the producer unapplied account. If you pass a negative value for the write-off, it is a negative write-off.

The method returns the producer write-off expense balance as a `MonetaryAmount` DTO object.

Make an Agency Payment

To make an agency payment from an external system, call the `PaymentAPI` web service method `makeAgencyBillPayment` or `makeAgencyBillPaymentToPolicyPeriod`.

See also

- The agency payment methods use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Agency Payment by Distribution Item List

To make an agency payment with a specified list of distributions, call `makeAgencyBillPayment`.

Note: This method uses Gosu class DTOs without the suffix DTO.

This method distributes the agency payment according to the distributions you provide as an array of `DistributionItemRecord` DTO objects. A `DistributionItemRecord` DTO is a Gosu class instance that represents one `DistItem` entity instance to create for a given invoice item. The details include the gross and commission amounts to allocate for that invoice item. The properties in `DistributionItemRecord` are as follows:

- `InvoiceItemID` – the invoice item public ID, as a `String`
- `GrossAmount` – the gross amount, as a `MonetaryAmount`
- `CommissionAmount` – the commission amount, as a `MonetaryAmount`

The method arguments are:

- A `PaymentReceiptRecord` DTO object.
- The array of `DistributionItemRecord` DTO objects, discussed earlier
- A boolean flag to determine whether to write off the difference between the payment amount and the net distribution amount. If the value is `true`, BillingCenter attempts the write-off. If `false`, the difference goes into producer unapplied.

The method returns the public ID (`String`) of the `AgencyBillMoneyRcvd` entity instance that is the new payment.

See also

- “Payment Receipt Record DTOs” on page 107

Agency Payment to a Policy Period

To make an agency payment to a policy period, call `makeAgencyBillPaymentToPolicyPeriod`.

Note: This method uses Gosu class DTOs without the suffix DTO.

This method makes an agency payment and distributes to all unsettled `InvoiceItem` objects that belong to the specified `PolicyPeriod`. BillingCenter distributes the money by using the registered implementation of the `IAgencyCycleDist` plugin interface. To distribute the money, BillingCenter calls the plugin implementations' `distributeGrossAndCommission` method.

- `CommissionAmount` – the commission amount, as a `MonetaryAmount`

The method arguments are:

- A `PaymentReceiptRecord` DTO object.
- A policy period public ID (`String`)
- The net amount to distribute, as a `MonetaryAmount` DTO.
- A `boolean` flag to determine whether to write off the difference between the payment amount and the net distribution amount. If the value is `true`, BillingCenter attempts the write-off. If `false`, the difference goes into producer unapplied.

BillingCenter creates executes a producer write-off if the net amount of the payment does not match the amount distributed. If the amount to write off exceeds the producer's write-off threshold, BillingCenter creates an activity to notify the account representative. However, the payment still goes through without the write-off.

The method returns the public ID (`String`) of the new payment (a `AgencyBillMoneyRcvd` entity instance).

See also

- “Payment Receipt Record DTOs” on page 107

Payment Instrument Web Service APIs (PaymentInstrumentAPI)

BillingCenter publishes the `PaymentInstrumentAPI` web service to help BillingCenter and PolicyCenter store information about payment instruments such as credits in a secure way. The `PaymentInstrumentAPI` web service lets you add payment instruments to accounts and to producers.

In an integration with PolicyCenter or another policy administration system, BillingCenter is the system of record for payment instruments. However, BillingCenter stores only tokens to represent payment instruments in its database. An external system serves as a secure repository for actual payment information, such as account numbers, expiration dates, and other personally identifiable information.

See also

- The payment instrument APIs use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Payment Instrument Record DTOs

All of the payment instrument APIs use a DTO called `PaymentInstrumentRecord`, which represents information in the `PaymentInstrumentRecord` entity type. Both objects describes a specific payment instrument for an account or a producer.

Note: The payment instrument APIs use Gosu class DTOs without the suffix DTO.

Create a Payment Instrument

The `PaymentInstrumentAPI` web service provides two methods to create payment instruments:

- `createPaymentInstrumentOnAccount` – Pass an account number as a String and a `PaymentInstrumentRecord`. Returns an updated `PaymentInstrumentRecord` DTO.
- `createPaymentInstrumentOnProducer` – Pass a producer number as a String and a `PaymentInstrumentRecord`. Returns an updated `PaymentInstrumentRecord` DTO.

Get Payment Instruments

The `PaymentInstrumentAPI` web service provides two methods to get payment instruments:

- `getPaymentInstrumentsForAccount` – Pass an account number as a String. Returns an array of `PaymentInstrumentRecord` objects;
- `getPaymentInstrumentsForProducer` – Pass a producer number as a String. Returns an array of `PaymentInstrumentRecord` objects.

Billing Summary Web Service APIs (BillingSummaryAPI)

BillingCenter publishes the `BillingSummaryAPI` web service for external systems to get billing summaries for accounts and policies. This web service is intended primarily for InsuranceSuite integration with PolicyCenter.

Note: The billing summary methods use Gosu class DTOs without the suffix DTO.

The following table summarizes when and how to use each billing summary method.

| To retrieve | Call this method | Description |
|--|---|---|
| account-level billing summary | <code>retrieveAccountBillingSummary</code> | Takes an account number and returns an account billing summary designed for PolicyCenter integration. The result type is an array of <code>BCPAccountBillingSummary</code> DTO objects. See the DTO Gosu class in Studio for a list of properties. |
| policy-level billing summary for an account | <code>retrievePeriodsBilledToAccount</code> | Takes an account number and a term number (an int). Returns a billing summary for that policy suitable for display in the PolicyCenter user interface. An account billing summary contains information like current and past amounts due. The method return type is an array of <code>PolicyBillingSummary</code> DTO objects. That DTO contains money values as instances of the <code>MonetaryAmount</code> type. |
| open policy periods that the given account pays but does not own | <code>retrievePeriodsBilledToAccount</code> | Takes an account number and returns a displayable policy period as a <code>DisplayablePolicyPeriod</code> DTO object. See the DTO Gosu class in Studio for a list of properties. |
| open policy periods owned by the given account | <code>retrievePeriodsForAccount</code> | Takes an account number and returns a displayable policy period as a <code>DisplayablePolicyPeriod</code> DTO object. See the DTO Gosu class in Studio for a list of properties. |
| invoices of a specified account | <code>retrieveInvoicesForAccount</code> | Takes an account number and returns invoices of the specified account as an array of <code>PCInvoiceInfo</code> DTO objects. See the DTO Gosu class in Studio for a list of properties. |

See also

- For important information on DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Trouble Ticket Web Service APIs (TroubleTicketAPI)

The TroubleTicketAPI web service that BillingCenter publishes has the following methods to let external systems manipulate trouble tickets:

- “Create Disaster Trouble Tickets” on page 114
- “Create a Trouble Ticket” on page 114
- “Place or Release Holds on Accounts, Policies, or Producers” on page 114

See also

- The trouble ticket APIs use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Create Disaster Trouble Tickets

To create disaster trouble tickets from an external system, call one of several methods of the TroubleTicketAPI web service:

- To create disaster trouble tickets on both accounts and policies based on a list of postal codes, call the method `createDisasterTroubleTicketsOnAccountsAndPoliciesWithPostalCodes`.
- To create disaster trouble tickets on policies (not also on accounts) based on a list of postal codes, call the method `createDisasterTroubleTicketsOnPoliciesWithPostalCodes`.

Both methods take a list of postal codes as `String` objects. Both methods return nothing.

Create a Trouble Ticket

To create a trouble ticket from an external system, call the TroubleTicketAPI web service method `createTroubleTicket`.

Note: This method uses Gosu class DTOs with the suffix `DTO`.

The one method argument is a `TroubleTicketDTO` object, which is a DTO that represents data in the `TroubleTicket` entity type. Refer to the *Data Dictionary* for the `TroubleTicket` entity properties. For the exact syntax of the corresponding properties in the DTO, refer to the `TroubleTicketDTO` Gosu class in Studio.

The method returns the public ID (`String`) of the new trouble ticket entity instance.

Place or Release Holds on Accounts, Policies, or Producers

You can place or release a *hold* on an account, policy, or producer from an external system. Placing a hold blocks some automated processes on that object until further action is taken. For more information about holds, see “Trouble Ticket Holds” on page 236 in the *Application Guide*.

Note: These methods use Gosu class DTOs without the suffix `DTO`.

Types of Holds

You can place different types of holds on accounts, policies, or producers. Whenever you place or release holds, you must specify an array of hold types.

The types are defined in the `HoldType` typelist. The `HoldType` typelist has values such as `CommissionPayments`, `CommissionPolicyEarn`, `Delinquency`, `InvoiceSending`, `PaymentDistribution`, or `Disbursements`.

Place a Hold

To place a hold, use the methods on the `TroubleTicketAPI` web service that have method names that begin with `putHoldOn`. These methods take the following arguments:

- A public ID of the object on which to place a hold
- An array of hold types, each of which is a `HoldType` typekey. See “Types of Holds” on page 114.
- A trouble ticket DTO of type `TroubleTicketDTO`. This DTO corresponds to the `TroubleTicket` entity type. Refer to the *Data Dictionary* for the full list of properties in the entity. For the exact syntax of properties in the DTO, see the `TroubleTicketDTO` class in Studio.

The following table summarizes which method to use.

| To put a hold on | Call this method |
|------------------|------------------------------------|
| account | <code>putHoldOnAccount</code> |
| policy | <code>putHoldOnPolicy</code> |
| policy period | <code>putHoldOnPolicyPeriod</code> |
| producer | <code>putHoldOnProducer</code> |

If the object already has a hold, these methods add hold types to the existing hold. If no hold exists, BillingCenter creates a new trouble ticket, and then links the hold to the trouble ticket.

Release a Hold

To release a hold, use the methods on the `TroubleTicketAPI` web service that have method names that begin with `releaseHoldOn`. These methods take the following arguments:

- A public ID of the object from which to release a hold
- An array of hold types that you want to release, each of which is a `HoldType` typekey. See “Types of Holds” on page 114.

The following table summarizes which method to use.

| To release a hold on | Call this method |
|----------------------|--|
| account | <code>releaseHoldOnAccount</code> |
| policy | <code>releaseHoldOnPolicy</code> |
| policy period | <code>releaseHoldOnPolicyPeriod</code> |
| producer | <code>releaseHoldOnProducer</code> |

After releasing the hold types, if there are zero active hold types on the policy period, BillingCenter marks the policy period as having no holds and closes the trouble ticket.

Other Billing Web Service APIs (BCAPI)

The BCAPI web service that BillingCenter publishes has the following methods to let external systems add billing collateral and other billing related information to objects in BillingCenter:

- “Create a Collateral Requirement” on page 116
- “Add a Note” on page 116
- “Add a Document” on page 117
- “Create an Activity” on page 117
- “Create a Trouble Ticket” on page 114

- “Start a Delinquency” on page 119
- “Trigger a Delinquency” on page 120
- “Change Billing Methods” on page 121
- “Create or Cancel a Premium Report Due Date” on page 122
- “Encrypt Data in Staging Tables” on page 122

See also

- Many of the BCAPI methods use data transfer objects (DTOs) to represent entity type data. For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Create a Collateral Requirement

An insurance company may require that a customer provide collateral for a loan or other instrument. BillingCenter tracks collateral requirements and can monitor that the requirements are satisfied.

The BCAPI web service method `createCollateralRequirement` enables an external system to create a new collateral requirement. The method accepts a `CollateralRequirementDTO` argument that specifies the properties of the new requirement. If the requirement is successfully created, the method returns its `PublicID` string.

The `CollateralRequirementDTO` argument cannot be `null`. Also, the following fields in the object must be set.

- `CollateralPublicID`
- `RequirementName`
- `RequirementType`. If the `RequirementType` is `CASH`, the `Segregated` field must be set.
- `EffectiveDate`. The `EffectiveDate` must be on or after the current date.
- The `ExpirationDate` can be `null`. If the `ExpirationDate` is not `null`, it must be after both the current date and the `EffectiveDate`.
- The `Required` field can be `null`. If the `Required` field is not `null` then the `Required.IsZero` field must be `false`.
- Either the `PolicyPublicID` or the `PolicyPeriodPublicID` must be set, but not both.

For field details, refer to the *Gosu API Reference*.

Changes to a customer’s collateral can be monitored by listening for a `CollateralChanged` messaging event. In the event handler, check the `Account.Collateral.Compliance` property which contains a typecode from the `CollateralStatus` typelist. For information about events and messaging, see “Messaging and Events” on page 303.

Add a Note

You can add notes to BillingCenter objects from external systems using methods on the BCAPI web service.

Note: These methods use Gosu class DTOs with the suffix `DTO`.

The following table lists the methods available to add notes for each object type.

| To add a note to this object | Use this method on the BCAPI web service |
|------------------------------|--|
| account | <code>addNoteToAccount</code> |
| policy period | <code>addNoteToPolicyPeriod</code> |
| producer | <code>addNoteToProducer</code> |
| trouble ticket | <code>addNoteToTroubleTicket</code> |

These methods all take two arguments:

- a populated NoteDTO DTO object, which represents data in the Note entity type.
- a public ID of the object on which to attach a new note. Depending on which method you call, this is the public ID of either an account, a policy period, a producer, or a trouble ticket.

All methods return the public ID (a String) of the new Note entity instance.

Add a Document

You can add documents to BillingCenter objects from external systems using methods on the BCAPI web service.

Note: These methods use Gosu class DTOs with the suffix DTO.

The following table lists the methods available to add notes for each object type.

| To add a document to this object | Use this method on the BCAPI web service |
|----------------------------------|--|
| account | addDocumentToAccount |
| policy period | addDocumentToPolicyPeriod |
| producer | addDocumentToProducer |

These methods all take two arguments:

- DocumentDTO DTO object, which represents data in the Document entity type.
- Public ID of the object on which to attach a new note. Depending on which method you call, this is the public ID of either an account, a policy period, or a producer.

All methods return the public ID (a String) of the new Document entity instance.

Create an Activity

You can add activities to BillingCenter objects from external systems using methods on the BCAPI web service. The BCAPI web service has several related methods called `createSharedActivity`, `createActivity`, and `createSharedActivityAssignedToUser`.

Note: These methods use Gosu class DTOs with the suffix DTO.

See also

- For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84

Create a Shared Activity

A BillingCenter *shared activity* is an activity with collective ownership, which means it has no specific owner. Shared activities are useful for billing systems because typical billing teams are small and as a group share many small tasks. Shared activities are similar to activity queues but without the overhead or capabilities of BillingCenter assignment. For example, a group might share tasks such as:

- Marking policies for closure after final audit or final retro adjustment
- Calling the insured for payment promises

To create a shared activity, call the BCAPI web service method `createSharedActivity`. The method takes as arguments:

- a public ID of an activity pattern for the new activity
- a subject
- a description
- a target date

- an escalation date
- a Boolean specifying whether the activity is mandatory
- a `TroubleTicketDTO` object, which is a DTO that represents properties on the `TroubleTicket` entity type

The method returns the public ID of the new `SharedActivity` entity instance.

Create an Activity for Automatic Assignment

To create a standard activity and run automatic assignment, call the `BCAPI` web service method `createActivity`. The method takes as arguments the same items as `createSharedActivity`. The method returns the public ID of the new `Activity` entity instance.

Create an Activity for Manual Assignment

To create a standard activity and assign to a specific user, call the `BCAPI` web service method `createActivityAssignedToUser`. The method takes the same arguments as `createSharedActivity` with two additional required arguments:

- Public ID of a User
- Public ID of a Group, which must contain the User

The method returns a the public ID of the new `Activity` entity instance.

Get a Billing Object Summary

You can get snapshots of some billing objects from an external system using several related methods on the `BCAPI` web service. The methods are `getAccountInfo`, `getProducerInfo`, and `getPolicyPeriodInfo`.

Note: These methods use Gosu class DTOs with the suffix `DTO`.

Some DTO properties are high-level summaries that require implicit calculations. For example, the `AccountInfoDTO.TotalBilled` property contains the total billed amount on the account, including all policies associated with this account.

WARNING These web service methods trigger very resource-intensive code on the server. On a production server, use extreme caution because of implications for server performance. If you require frequent or complex reporting, use a separate reporting system rather than these web service methods.

The returned information is a momentary snapshot of changing data. Do not rely on the information to be unchanged in subsequent API calls.

Refer to the DTO implementation classes for the list of all DTO properties. However, refer to the *Data Dictionary* for `Account`, `PolicyPeriod`, and `Producer` entity types for details of each property.

The following table lists the available BCAPI web service methods that get these objects.

| To get a summary of this object | Use this method on the BCAPI web service | Returns this DTO type | Description |
|---------------------------------|--|-----------------------|--|
| account | getAccountInfo | AccountInfoDTO | <p>The DTO includes information such as:</p> <ul style="list-style-type: none"> AmountBilled – Amount billed AmountDue – Amount due AmountPaid – Amount paid AmountUnbilled – Amount unbilled Delinquent – Whether the account is delinquent HasOpenPolicies – Whether the account has open policies LastInvoiceAmount – Last invoice payment NextInvoiceAmount – Next invoice due <p>Refer to the DTO class for more details and more properties.</p> |
| policy period | getPolicyPeriodInfo | PolicyPeriodInfoDTO | <p>The DTO includes properties such as the effective date, the expiration date, and a reference to the Policy entity linked to this policy period.</p> <p>Properties include:</p> <ul style="list-style-type: none"> EffectiveDate – The date this policy period takes effect ExpirationDate – The date this policy period expires Policy – The Policy entity associated with this policy period Account – The Account entity associated with this policy period <p>Refer to the DTO class for more details and more properties.</p> |
| producer | getProducerInfo | ProducerInfoDTO | <p>The ProducerInfo entity includes properties such as:</p> <ul style="list-style-type: none"> InfoDate – The date and time that this snapshot was taken PrimaryAddress – The address of the producer's primary contact Producer – The Producer entity itself, from which you can extract any additional properties totalGrossPremium – The total gross premium for this producer totalOutstanding – The total outstanding for this producer totalPastDue – The total past due for this producer <p>Refer to the DTO class for more details and more properties.</p> |

Start a Delinquency

To start a delinquency on an account from an external system, call the BCAPI web service method `startDelinquencyOnAccount`. To start delinquency on a policy period, call the BCAPI web service method `startDelinquencyOnPolicyPeriod`.

These methods do not verify whether an active delinquency process exists on the account or policy period. You must be certain the account or policy period does not have an active delinquency process before you call either method.

The Delinquency Process DTO

Both delinquency methods return a `DelinquencyProcessDTO` object, which represents properties on the `Delinquency` entity type. See the *Data Dictionary* for more details about `Delinquency`. For the full list of properties on the DTO, see the `DelinquencyProcessDTO` Gosu class in Studio.

Note: These methods use Gosu class DTOs with the suffix `DTO`.

See also

- For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84

Start a Delinquency for an Account

To start a delinquency on an account, call the `startDelinquencyOnAccount` method. Its arguments are an account public ID and a reason (`DelinquencyReason` typecode).

Start a Delinquency on a Policy

To start a delinquency on a policy, call the `startDelinquencyOnPolicyPeriod` method. Its arguments are an policy period public ID and a reason (`DelinquencyReason` typecode).

Trigger a Delinquency

You can trigger a delinquency workflow from an external system by using methods on the BCAPI web service. Triggering a delinquency means to invoke the trigger in the workflow (if available) for the active delinquency processes associated with a specified account or policy period.

See also

- “Delinquency Plans” on page 127 in the *Application Guide*

Trigger Delinquency on an Account

To trigger a delinquency workflow on an account, call the BCAPI web service method `triggerDelinquencyWorkflowOnAccount`. The method takes an account public ID and a workflow key.

A workflow key is a typecode from the `WorkflowTriggerKey` typelist. A workflow key identifies changes to the delinquency such as:

- `ExitDelinquency` – exit delinquency
- `CanceledInPAS` – canceled in a policy administration system
- `Cancel` – cancel delinquency

The method invokes the trigger in the workflow (if available) for the active delinquency processes for the account. BillingCenter invokes delinquency processes of all `PolicyPeriod` objects of the `Account`.

If all workflow triggers are available and invoked, the method returns `true`. If one or more workflow trigger is unavailable, the method returns `false`.

Trigger Delinquency on a Policy

To trigger a delinquency workflow on a policy period, call the BCAPI web service method `triggerDelinquencyWorkflowOnPolicyPeriod`. The method takes a policy period public ID and a workflow key. For more about workflow keys, see “Trigger Delinquency on an Account” on page 120.

The method triggers the workflow (if available) for the single active delinquency process associated with the policy period.

The method returns `true` if the trigger invoked successfully. Otherwise, returns `false` to indicate the trigger was unavailable.

Change Billing Methods

You can change the billing method on a policy period from external systems using BCAPI web service methods. There are three related methods `changeBillingMethodToAgencyBill`, `changeBillingMethodToDirectBill`, and `changeBillingMethodToListBill`. These methods are web service equivalents to tools that change billing methods in the BillingCenter user interface. The methods all return nothing.

Change to Agency Bill

To change the billing method of a policy period to agency bill, call the BCAPI web service method `changeBillingMethodToAgencyBill`. The method takes a policy period specified by public ID. The policy period that you specify must have a producer primary and the producer must have an agency bill plan already. Ensure these things are true before calling this API method.

The billing method change takes effect at the beginning of the policy period. To implement this change, BillingCenter does the following:

- adds or deletes commissions
- adds or deletes invoice items
- updates the payer

In the base configuration, BillingCenter does not reverse payments for invoice items when the items are moved.

The method returns nothing.

Change to Direct Bill

To change a policy period's billing method to direct bill, call the BCAPI web service method `changeBillingMethodToDirectBill`. This method tries to make the original policy period appear like it was originally direct bill. BillingCenter processes reallocating commissions, moves invoice items, and modifies anything else necessary for the billing plan change. This method takes as arguments:

- a policy period specified by public ID
- a Boolean value that indicates whether to create an invoice for today. If this parameter is `true`, the new invoice contains all past invoice items for this policy period. If this parameter is `false`, BillingCenter does not create an invoice.

The billing method change takes effect at the beginning of the policy period. To implement this change, BillingCenter does the following:

- adds or deletes commissions
- adds or deletes invoice items
- updates the payer

In the base configuration, BillingCenter reverses payments for invoice items being moved if the item has been billed or is due.

The method returns nothing.

Change to List Bill

To change a policy period's billing method to list bill, call the BCAPI web service method `changeBillingMethodToListBill`.

This method takes as required non-empty arguments:

- the public ID of a `PolicyPeriod` to change

- the public ID of a list bill Account to become the payer
- the public ID of a PaymentPlan
- the public ID of an InvoiceStream

Additionally, there is an Boolean argument that specifies which invoices to move. To move all invoices to the new payer, pass `false` or `null`. To move only the planned invoices to the new payer, pass `true`.

The billing method change takes effect at the beginning of the policy period. To implement this change, BillingCenter does the following:

- adds or deletes commissions
- adds or deletes invoice items
- updates the payer

In the base configuration, BillingCenter does not reverse payments for invoice items when the items are moved.

The method returns nothing.

Create or Cancel a Premium Report Due Date

You can create or cancel a premium report due date from an external system by using methods on the BCAPI web service.

Note: These methods use Gosu class DTOs with the suffix `DTO`.

See also

- For important information about DTOs generally, see “BillingCenter Data Transfer Objects (DTOs)” on page 84

Create a Premium Report Due Date

To create a premium report due date, call the BCAPI web service method `createPremiumReportDueDate`. The only argument is a `PremiumReportDueDateDTO` object. This object is a DTO that represents data from the billing instruction entity type `PremiumReportDueDate`.

For this method, you must set the following `PremiumReportDueDateDTO` properties to non-null values: `DueDate`, `PeriodStartDate`, and `PremiumReportDDPolicyPeriodPublicID`.

The method returns the public ID of the new billing instruction.

Cancel a Premium Report Due Date

To cancel the premium report due date, call the BCAPI web service method `cancelPremiumReportDueDate`. The arguments are:

- Policy period public ID
- Start date (a `Date` object)
- End date (a `Date` object)

If BillingCenter found and canceled the `PremiumReportDueDate` successfully, the method returns `true`. Otherwise, the method returns `false`.

Encrypt Data in Staging Tables

To encrypt data in staging tables, BillingCenter provides two methods that appear both in the `TableImportAPI` and `BCAPI` web services:

- `encryptDataOnStagingTables` – Encrypt data in staging tables.

- `encryptDataOnStagingTablesAsBatchProcess` – Encrypt data in staging tables asynchronously in a batch process.

These methods have the same purpose and operation in both web services. One exception is that `encryptDataOnStagingTables` does not return a value in the BCAPi variant.

See also

- “Encryption Features for Staging Tables” on page 259
- “Importing from Database Staging Tables” on page 367.

General Web Services

This topic describes general-purpose web services, such as mapping typecodes general system tools.

Note: This topic relies on you to understand what web services are and how to call them from remote systems. Before reading this topic, read “What are Web Services?” on page 27.

This topic includes:

- “Mapping Typecodes to External System Codes” on page 125
- “Importing Administrative Data” on page 127
- “Maintenance Tools Web Service” on page 128
- “System Tools Web Services” on page 129
- “Workflow Web Services” on page 130
- “Profiling Web Services” on page 131

Mapping Typecodes to External System Codes

If possible, configure the BillingCenter typelists to include typecode values that match those already used in external systems. If you can do that, you do not need to map codes between systems.

However, in many installations this is infeasible and you must map between internal codes and external system codes. For example, you might have multiple external legacy systems and they do not match each other, so BillingCenter can only match one system at maximum.

BillingCenter provides a built-in utility to support simple typecode mappings. The first step in using the utility is to define the mappings. The mappings go into the `BillingCenter/modules/configuration/config/typelists/mapping/typecodemapping.xml` file. The following is a simple example:

```
<?xml version="1.0"?>
<typecodemapping>
  <namespacelist>
    <namespace name="ns1" />
    <namespace name="ns2" />
  </namespacelist>

  <typelist name="LossType">
```

```

<mapping typecode="PR" namespace="ns1" alias="Prop" />
<mapping typecode="PR" namespace="ns2" alias="CPL" />
</typelist>
</typecodemapping>

```

The first section of the mapping file lists the set of *namespaces*. These namespaces correspond to the different external systems you need to map. For example, if the mappings are different between two external systems, then each has its own namespace.

The rest of the mapping file contains sections for each typelist that requires mapping. Within the typelist, add elements for mapping entries. Each mapping entry contains:

- **Typecode code** – The BillingCenter typecode to map.
- **Namespace** – The namespace is the name of the external system in the XML file and used by any tool that translates type codes. You may wish to define your namespace strings with your company name and a system name. For example, for the company name ABC for a check printing service, you might use "ABC:checkprint".
- **Typecode alias** – The *alias* is the value for this typecode in the external system.

There can be multiple mapping entries for the same typecode and namespace. This supports situations in which multiple external codes map to the same BillingCenter code during data import.

Using Web Services to Translate Typecodes

After you define the mappings, you can translate between internal and external codes using `TypeListToolsAPI` web service methods:

- `getAliasByInternalCode` – Find the alias, if any, for a given typelist, code, and namespace. If there is no alias for that code and namespace, returns `null`. The `null` return value indicates that the internal and external codes are the same.
- `getTypeKeyByAlias` – Find the internal typecode, if any, for a given typelist, alias, and namespace. This returns a data object that contains the typecode's code, name, and description properties. If no mapping is found, there are two possible meanings. It might indicate that the external and internal codes are the same. However, it might indicate that the mapping is missing from your mapping code. Use the `getTypeListValues` method to verify that the external code is a valid internal code in this situation.
- `getAliasesByInternalCode` – During exports, gets an array of `String` values that represent external aliases to internal typecodes given a typelist, a namespace, and an internal code.
- `getTypeKeysByAlias` – During imports, gets an array of `TypeKeyData` objects given a typelist, a namespace, and an alias.
- `getTypeListValues` – Given the name of a typelist, returns an array of all the typekey instances contained within that typelist.

From Gosu and Java code that runs on the BillingCenter server, always use the `TypecodeMapperUtil` utility class, not the `TypeListToolsAPI` web service. See “Using Gosu or Java to Translate Typecodes” on page 126.

Using Gosu or Java to Translate Typecodes

Typecode translation may occur very frequently in Gosu or Java plugin code, or in Gosu templates used for data extraction by external systems. From Gosu and Java code that runs on the BillingCenter server, always use the `TypecodeMapperUtil` utility class, not the `TypeListToolsAPI` web service.

From Gosu or Java, translate a typecode using `TypecodeMapperUtil`. Call its static method `getTypecodeMapper` to return a mapper object that has a `getInternalCodeByAlias` method.

From Gosu, the code looks like:

```

var mapper = gw.api.util.TypecodeMapperUtil.getTypecodeMapper()

var mycode = mapper.getInternalCodeByAlias( "Contact", "ABC:system1", "ATTORNEY" )

```

See also

- For details of other plugin utilities, see “Useful Java Plugin APIs” on page 143.

Importing Administrative Data

BillingCenter provides tools for importing and exporting data in XML using the import tools API (`ImportToolsAPI`) web service. The easiest way to export data suitable for import is to use the built-in user interface in the application.

For related topics, see:

- “Importing Administrative Data Using the `import_tools` Command” on page 100 in the *System Administration Guide*
- “Importing and Exporting Administrative Data from BillingCenter” on page 101 in the *System Administration Guide*

To prepare data for XML import, use the generated *XML Schema Definition* (XSD) files that define the XML data formats:

```
BillingCenter/build/xsd/bc_import.xsd  
BillingCenter/build/xsd/bc_entities.xsd  
BillingCenter/build/xsd/bc_typelists.xsd
```

Regenerate these files by calling the `regen-xsd` command using the `gwbc` command line tool. See “Importing and Exporting Administrative Data from BillingCenter” on page 101 in the *System Administration Guide*.

The `ImportToolsAPI` web service method `importXmlData` imports administrative data from an XML file. Only use this for administrative database tables. Any other use is unsafe. This API does not perform integrity checks (as done during staging-table import) or data validation on imported data. Administrative tables include `User` and `Group` and their related entities.

IMPORTANT The `ImportToolsAPI` web service method `importXmlData` is only supported for administrative data due to the lack of validation for this type of import.

The main XML import routine is `importXmlData` and it takes a single `String` argument containing XML data:

```
importToolsAPI.importXmlData(myXMLData);
```

CSV Import and Conversion

There are several other methods in this interface related to importing and converting to and from comma-separated value data (CSV data). For example, import data from simple sample data files and convert them to a format suitable for XML import, or import them directly. See the Javadoc in the implementation file for more information about these CSV-related methods:

- `importCsvData` – import CSV data
- `csvToXml` – convert CSV data to XML data
- `xm1ToCsv` – convert XML data to CSV data

Advanced Import or Export

If you must import data in other formats or export administrative data programmatically, write a new web service to export administrative information one record at a time. For both import and export, if you write your own web service, be careful never to pass too much data across the network in any API call. If you send too much data, memory errors occur. Do not try to import or export all administrative data in a dataset at once.

Maintenance Tools Web Service

The maintenance tools (`MaintenanceToolsAPI`) web service provides a set of tools available only if the system is at the `maintenance run` level or higher.

Running Batch Processes Using Web Services

One of the methods in the `MaintenanceToolsAPI` web service is `startBatchProcess`, which runs a batch process or a writer for a work queue. The API notifies the caller that the request is received. The caller must poll the server later to see if the process failed or completed successfully. For server clusters, batch process runs only on the batch server. However, you can make the API request to any of the servers in the cluster. If the receiving server is not the batch server, the request automatically forwards to the batch server.

For example, run a batch process and get the process ID of the batch process:

```
processID = maintenanceTools.startBatchProcess("memorymonitor");
```

Terminate a batch process by process name or ID, for example:

```
maintenanceTools.terminateBatchProcessByName("memorymonitor");
maintenanceTools.terminateBatchProcessByID(processID);
```

Check the status of a batch process, for example:

```
maintenanceTools.batchProcessStatusByName("memorymonitor");
maintenanceTools.batchProcessStatusByID(processID);
```

For work queues, the status methods returns the status only of the writer thread. The status methods do not check the work queue table for remaining work items. The status of a writer reports as completed after the writer finishes adding work items for a batch to the work queue. Meanwhile, many work items in the batch may remain unprocessed.

For the batch process methods of the maintenance tools API, the batch processes apply only to BillingCenter, not additional Guidewire applications that you integrated with it.

Whenever you use the `MaintenanceToolsAPI` web service to run a batch process provided in the base configuration, identify the batch process by its code as listed in the documentation. To run a custom batch process, identify the process by the `BatchProcessType` typecode that you added for it. The typecode must include the `APIRunnable` category to start your custom batch process with the `MaintenanceToolsAPI` web service.

See also

- For the list of codes for batch processes in the base configuration, see “List of Work Queues and Batch Processes” on page 120 in the *System Administration Guide*
- For alternatives to the `MaintenanceToolsAPI` web service, see “Running Work Queue Writers and Batch Processes” on page 111 in the *System Administration Guides*.

Manipulating Work Queues Using Web Services

A *work queue* represents a pool of work items that can be processed in a distributed way across multiple threads or even multiple servers. Several web service APIs query or modify the existing work queue configuration. For example, APIs can get the number of the workers threads configured for this server (`instances`) and the configured delay between processing each work item (`throttleInterval`).

Wake up all workers for the specified work queue across the entire cluster:

```
maintenanceTools.notifyQueueWorkers("ActivityEsc");
```

Get the work queue names for this instance of BillingCenter:

```
stringArray = maintenanceTools.getWorkQueueNames();
```

Get the number of instances and throttle interval for a work queue:

```
WorkQueueConfig wqConfig = maintenanceTools.getWorkQueueConfig("ActivityEsc");
numInstances = wqConfig.getInstances();
```

Set the number of instances and throttle interval for a work queue:

```
WorkQueueConfig wqConfig = new WorkQueueConfig();
wqConfig.setInstances(1);
wqConfig.setThrottleInterval(999);
WorkQueueConfig wqConfig = maintenanceTools.setWorkQueueConfig("ActivityEsc", wqConfig);
```

Worker instances that are running stop after they complete their current work item. Then, the server creates and starts new worker instances as specified by the configuration object that you pass to the method.

The changes made using the batch process web service API are temporary. If the server starts (or restarts) at a later time, the server rereads the values from `work-queue.xml` to define how to create and start workers.

For these APIs, the term *product* and *cluster* apply to the current Guidewire product only as determined by the SOAP API server requested.

Stopping Startable Plugins Using Web Services

A *startable plugin* is a special type of code that runs without human intervention in the application server as a background process, beginning at server startup. You can stop startable plugins and start them again by using methods on the `MaintenanceToolsAPI` web service.

To stop a startable plugin, call the `stopPlugin` method. To start a startable plugin again, call the `startPlugin` method. These methods take the plugin name in `String` format as their only argument. Plugin names are defined in the Plugin Registry in Studio.

In general, startable plugins run only the batch server, but you can develop startable plugins that run distributed on all servers. Before you use the `MaintenanceToolsAPI` web service to start or stop a distributed startable plugin, be certain you understand the implications for state management across all servers.

See also

- “Startable Plugins” on page 273
- “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*
- “Configuring Startable Plugins to Run on All Servers” on page 277

System Tools Web Services

The system tools API (`SystemToolsAPI`) interface provides a set of tools that are always available, even if the server is set to `dbmaintenance` run level. For servers in clusters, system tools API methods execute **only** on the server that receives the request. For the complete set of methods in the system tools API, refer to the `Gosu` implementation class in Studio.

Getting and Setting the Run Level

The most important usage of the `SystemToolsAPI` interface is to set the system run level:

```
systemTools.setRunLevel(SystemRunlevel.GW_MAINTENANCE);
```

Or, to get the system run level:

```
runlevelString = systemTools.getRunLevel().getValue();
```

Sometimes you may want a more lightweight way of determining the run level of the server from another computer on the network than to use SOAP APIs. You might want to informally use your web browser during development to check the run level.

To check the run level, simply call the ping URL on a BillingCenter server:

```
http://server:port/bc/ping
```

For example:

```
http://BillingCenter.mycompany.com:8080/bc/ping
```

Assuming that the server is running, this returns an extremely short result as an HTML document containing a text encoding of the server run level in a special format.

If you are just checking whether the server is up, you do not care about the return result from this ping URL. Typically in such cases, if it returns a result at all it proves your server is running. In contrast, if there is an HTTP error or browser error, the server is not running to respond to the ping request.

If you want the actual run level, check the contents of the HTTP result. However, the value is not simply a standard text encoding of the public run level enumeration. It represents the ASCII character with decimal value of an integer that represents the internal system run level (30, 40, 50).

The following table lists the correlation between the run level and the value return by the ping URL:

| Run level | Ping URL result character |
|---------------------------|---|
| <i>Server not running</i> | <i>No response to the HTTP request</i> |
| DB_MAINTENANCE | ASCII character 30, which is the Record Separator character |
| MAINTENANCE | ASCII character 40, which is the character "(" |
| MULTIUSER | ASCII character 50, which is the character "2" |
| GW_STARTING | ASCII character 0. A null character result might not be returnable for some combinations of HTTP servers and clients. |

Getting Server and Schema Versions

You can use the `SystemToolsAPI` interface to get the current server and schema versions.

The following example code in Java demonstrates how to get this information:

```
versionInfo = systemTools.getVersion();
appVersion = versionInfo.getAppVersion();
schemaVersion = versionInfo.getSchemaVersion();
configVersion = versionInfo.getConfigVersion();
configVersionModified = versionInfo.getConfigVersionModified();
```

Workflow Web Services

The web service API interface `WorkflowAPI` allows you to control BillingCenter workflows from external client API code, including BillingCenter plugins that use the web service APIs. In addition to being called by remote systems, the built-in `workflow_tools` command-line tools use these methods internally.

Workflow Basics

You can invoke a workflow trigger remotely from an external system using the `invokeTrigger` method. To check whether you can invoke that trigger, call the `isTriggerAvailable` method, described later in the section.

Be aware that any time the application detects a workflow error, the workflow sets itself to the state `TC_ERROR`. If this happens, you can remotely resume the workflow using these APIs.

Refer to the following table for workflow actions you can request from remote systems:

| Action | WorkflowAPI method | Description |
|--------------------------------------|--------------------|---|
| Invoke a workflow trigger | invokeTrigger | Invokes a trigger key on the current step of the specified workflow, causing the workflow to advance to the next step. This method takes a workflow public ID and a String value that represents the workflow trigger key from the WorkflowTriggerKey typelist. To check whether you can call this workflow trigger, use the isTriggerAvailable method in this interface (see later in this table). This method returns nothing. |
| Check whether a trigger is available | isTriggerAvailable | Check if a trigger is available in the workflow. If a trigger is available, it means that it is acceptable to pass the trigger ID to the invokeTrigger method in this web services interface. This method takes a workflow public ID and a String value that represents the workflow trigger key from the WorkflowTriggerKey typelist. It returns true or false. |
| Resume a single workflow | resumeWorkflow | Restarts one workflow specified by its public ID. This method sets the state of the workflow to TC_ACTIVE. This method returns nothing. |
| Resume all workflows | resumeAllWorkflows | Restarts all workflows that are in the error state. It is important to understand that this only affects workflows currently in the error state TC_ERROR or TC_SUSPENDED. The workflow engine subsequently attempts to advance these workflows to their next steps and set their state to TC_ACTIVE. For each one, if an error occurs again, the application logs the error sets the workflow state back to TC_ERROR. This method takes no arguments and returns nothing. |
| Suspend a workflow | suspend | Sets the state of the workflow to TC_SUSPENDED. If you must restart this workflow later, use the resumeWorkflow method or the resumeAllWorkflows method. |
| Complete a workflow | complete | Sets the state of a workflow (specified by its public ID) to TC_COMPLETED. This method returns nothing. |

Profiling Web Services

From remote systems you can enable or disable the BillingCenter profiler system using the ProfilerAPI web service. The methods use these method arguments:

- A boolean value that indicates whether to enable (`true`) or disable (`false`) the profiler for a particular component of the system, which varies by method.
- A process type (a typecode in the BatchProcessType typelist)
- A boolean value that controls whether to use high resolution clock for timing (`true`) or not (`false`). This only has an affect on the Windows operating system.
- A boolean value that controls whether to enable stack traces (`true`) or not (`false`). This feature is performance intensive, so think carefully before enabling this. This parameter is ignored if the first argument (to enable profiling) is false.
- A boolean value that controls whether to enable query optimizer tracing (`true`) or not (`false`). This feature is performance intensive, so think carefully before enabling this. This parameter is ignored if the first argument (to enable profiling) is false.
- A boolean value that controls whether to allow *extended* query tracing. This feature is performance intensive, so think carefully before enabling this. This parameter is ignored if the first argument (to enable profiling) is false.
- An integer (`int`) value that threshold for how long (in milliseconds) a database operation can take before generating a report using dbms counters. Set the value 0 to disable dbms counters.

Batch Processes

To enable the profiler for batch processes of a specific type, call the `setEnableProfilerForBatchProcess` method.

Batch Processes and Work Queues

To enable the profiler for batch processes and work queues, call the `setEnableProfilerForBatchProcessAndWorkQueue` method.

Messaging Destinations

To enable the profiler for messaging destinations, call the `setEnableProfilerForMessageDestination` method.

Startable Plugins

To enable the profiler for startable plugins, call the `setEnableProfilerForStartablePlugin` method.

Subsequent Web Sessions

To enable the profiler for subsequent web sessions, call the `setEnableProfilerForSubsequentWebSessions` method.

Web Services

To enable the profiler for web services, call the `setEnableProfilerForWebService` method.

Work Queues

To enable the profiler for work queues, call the `setEnableProfilerForWorkQueue` method.

part III

Plugins

Plugin Overview

BillingCenter plugins are software modules that BillingCenter calls to perform an action or calculate a result. BillingCenter defines a set of plugin interfaces. You can write your own implementations of plugins in Gosu or Java.

This topic includes:

- “Overview of BillingCenter Plugins” on page 136
- “Error Handling in Plugins” on page 141
- “Temporarily Disabling a Plugin” on page 141
- “Example Gosu Plugin” on page 141
- “Special Notes For Java Plugins” on page 142
- “Getting Plugin Parameters from the Plugins Registry Editor” on page 143
- “Writing Plugin Templates For Plugins That Take Template Data” on page 144
- “Plugin Registry APIs” on page 145
- “Plugin Thread Safety” on page 147
- “Reading System Properties in Plugins” on page 151
- “Do Not Call Local Web Services From Plugins” on page 152
- “Creating Unique Numbers in a Sequence” on page 152
- “Restarting and Testing Tips for Plugin Developers” on page 153
- “Summary of All BillingCenter Plugins” on page 153

See also

- To help with writing Java plugins, see “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.
- To help understand plugin interfaces, see “Interfaces” on page 219 in the *Gosu Reference Guide*.
- For information about messaging plugins, see “Messaging and Events” on page 303.
- For information about authentication plugins, see “Authentication Integration” on page 207.
- For information about document and form plugins, see “Document Management” on page 217.
- For information about other plugins, see “Other Plugin Interfaces” on page 265.

- For a feature-level perspective of the Billing and Policy integration, see “Policy Administration System Integration” on page 399 in the *Application Guide*.
- In the PolicyCenter (not BillingCenter) documentation, see the “Billing Integration” topic of the *PolicyCenter Integration Guide*.

Overview of BillingCenter Plugins

BillingCenter plugins are classes that BillingCenter invokes to perform an action or calculate a result at a specific time in its business logic. BillingCenter defines plugin interfaces for various purposes:

- Perform calculations
- Provide configuration points for your own business logic at clearly defined places in application operation, such as validation or assignment
- Generate new data for other application logic, such as generating a new claim number
- Define how the application interacts with other Guidewire InsuranceSuite applications for important actions relating to claims, policies, billing, and contacts.
- Define how the application interacts with other third-party external systems such as a document management system, third-party claim systems, third-party policy systems, or third-party billing systems.
- Define how BillingCenter sends messages to external systems.

You can implement a plugin in the programming languages Gosu or Java. In many cases, it is easiest to implement a plugin with a Gosu class. If you use Java, you must use a separate IDE other than Studio, and you must regenerate Java API libraries after any data model changes.

If you implement a plugin in Java, optionally you can write your code as an *OSGi* bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java API libraries. Guidewire recommends OSGi for all new Java plugin development. For extended documentation on Java and OSGi usage, see “Java and OSGi Support” on page 451.

From a technical perspective, BillingCenter defines a plugin as an *interface*, which is a set of functions (also known as methods) that are necessary for a specific task. Each plugin interface is a strict contract of interaction and expectation between the application and the plugin implementation. Some other set of code that implements the interface must perform the task and return any appropriate result values. For conceptual information about interfaces, see “Interfaces” on page 219 in the *Gosu Reference Guide*.

Conceptually, there are two main steps to implement a plugin:

- 1. Write a class (in Gosu or Java) that implements a plugin interface** – See “Implementing Plugin Interfaces” on page 137.
- 2. Register your plugin implementation class** – See “Registering a Plugin Implementation Class” on page 139

For most plugin interfaces, you can only register a single plugin implementation for that interface. However, some plugin interfaces support multiple implementations for the same interface, such as messaging plugins and startable plugins. For the maximum supported implementations for each interface, see the table in “Summary of All BillingCenter Plugins” on page 153.

The plugin itself might do most of the work or it might consult with other external systems. Many plugins typically run while users wait for responses from the application user interface. Guidewire strongly recommends that you carefully consider response time, including network response time, as you write your plugin implementations.

Implementing Plugin Interfaces

Choose a Plugin Implementation Type

There are several ways to implement a BillingCenter plugin interface:

- **Gosu plugin** – A Gosu class. Because you write and debug your code directly in BillingCenter Studio, in many cases it is easiest to implement a plugin interface as a Gosu class. The Gosu language has powerful features including type inference, easy access to web service and XML, and language enhancements such as Gosu blocks and collection enhancements.
- **Java plugin** – A Java class. You must use an IDE other than Studio. If you write your plugin in Java, you must regularly regenerate the Java API libraries after changes to data model configuration. You can use any Java IDE. You can choose to use the included application called IntelliJ IDEA with OSGi Editor for your Java plugin development even if you do not choose to use OSGi.
- **OSGi plugin** – A Java class encapsulated in an *OSGi* bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java API libraries. To simplify OSGi configuration, BillingCenter includes an application called IntelliJ IDEA with OSGi Editor. For more information about OSGi, see “Overview of Java and OSGi Support” on page 451.

IMPORTANT Guidewire recommends OSGi for all new Java plugin development.

The following table compares the types of plugin implementations you can create:

| Features of each plugin implementation type | Gosu plugin | Java plugin (no OSGi) | OSGi plugin (Java with OSGi) |
|---|-------------|-----------------------|------------------------------|
| Choice of development environment | | | |
| You can use BillingCenter Studio to write and debug code | ● | | |
| You can use the included application IntelliJ IDEA with OSGi editor to write code | | ● | ● |
| Usability | | | |
| Native access to Gosu blocks, collection enhancements, Gosu classes | ● | | |
| Entity and typecode APIs are the same as for Rules code and PCF code | ● | | |
| Requires regenerating Java API libraries after data model changes | | ● | ● |
| Third-party Java libraries | | | |
| Your plugin code can use third-party Java libraries | ● | ● | ● |
| You can embed third-party Java libraries within a OSGi bundle to reduce conflicts with other plugin code or BillingCenter itself. | | | ● |
| Dependencies on specific third-party packages and classes are explicit in manifest files and validated at startup time. | | | ● |

Writing Your Plugin Implementation Class

First you must decide which plugin implementation type you want to use and launch the appropriate IDE based on whether you are using Gosu or Java. To learn how the plugin type affects what IDE you must use, see “Implementing Plugin Interfaces” on page 137.

In your IDE, create the class within your own package hierarchy such as `mycompany.plugins`. Do not create your own classes in the `gw.*` or `com.guidewire.*` package hierarchies.

In Gosu and Java, create a class with a declaration containing the `implements` keyword, such as:

```
public class MyContactSystemClass implements IContactSystemPlugin {
```

If your class does not properly implement the plugin interface, your IDE shows compilation errors and can offer to add methods to your class that the interface requires. You must fix any issues before you code compiles.

Implement all public methods of the interface. Create as many other private methods and related classes as you need to provide internal logic for your code. However, BillingCenter only calls the public methods in your main implementation class.

IMPORTANT If you use OSGi, you must perform additional configuration. See “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 469.

For Gosu Plugins, the Method Signatures May Be Different than Java

In BillingCenter Studio or in your Java IDE, the compiler can detect that a class that implements an interface does not yet implement all the methods in the interface. The editor provides a tool that creates new stub versions of any unimplemented methods.

Note that the Gosu versions of the methods look different from the Java versions in many cases. The most common compilation issue is that if an interface’s method looks like properties, you must implement the interface as a Gosu property.

If the interface contains a method starting with the substring `get` or `is` and takes no parameters, define the method using property syntax. Do not simply implement it as simple method with the name as defined in the interface. For example, if plugin `ExamplePlugin` declared a method `getMyVar()`, your Gosu plugin implementation of this interface must not include a `getMyVar` method. Instead, it must look similar to the following:

```
package mycompany.plugins  
  
class MyClass implements gw.plugin.IExamplePlugin {  
  
    property get MyVar() : String {  
        ...  
    }  
}
```

Similarly, methods that begin with `set` and take exactly one argument become property setters.

See “Java get/set/is Methods Convert to Gosu Properties” on page 125 in the *Gosu Reference Guide*.

Plugin Templates (Only for Some Plugin Interfaces)

A small number of plugin interface methods provide method arguments that specify data as `String` values that contain data extracted from potentially large object graphs. This `String` data is called *template data*. Template data is the output of a Gosu template called a *plugin template*. A Gosu template is a short Gosu program that generates some output. In this case, the extracted data is an intermediary data format between rather than passing references to the original entity data objects directly to the plugin implementation.

If you see a plugin method with an argument called `templateData`, configure an appropriate plugin template that generates the information that your plugin implementation needs. See “Writing Plugin Templates For Plugins That Take Template Data” on page 144.

Built-in Plugin Implementation Classes

For some plugin interfaces, BillingCenter provides a built-in plugin implementation that you can use instead of writing your own version. Some plugin implementation classes are pre-registered in the default configuration.

Some plugin implementations are for demonstration only. Check the documentation for each plugin interface or contact Customer Support if you are not sure whether an included plugin implementation is supported for production.

BillingCenter includes plugin implementations to connect with other Guidewire applications in the Guidewire InsuranceSuite. The plugin implementations for InsuranceSuite integration are located in packages that include the intended target application and version number. This helps ensure smooth migration and backwards compatibility among Guidewire applications. Carefully confirm package names for any plugin implementations you want to use. The package may include the Guidewire application two-digit abbreviation, followed by the application version number with no periods. For example, the shortened version of BillingCenter 8.0.1 is bc801. Be sure to choose the plugin implementation class to match the version of the *other* external application, not the current application.

Registering a Plugin Implementation Class

In most cases, the first step to implementing a plugin interface is to create your implementation class. See “[Implementing Plugin Interfaces](#)” on page 137.

In other cases, you might be using a built-in plugin implementation class. See “[Built-in Plugin Implementation Classes](#)” on page 138.

In either case, you must also *register* the plugin implementation class so the application knows about it. The registry configures which implementation class is responsible for which plugin interface. If you correctly register your plugin implementation, BillingCenter calls the plugin at the appropriate times in the application logic. The plugin implementation performs some action or computation and in some cases returns results back to BillingCenter.

To use the Plugins registry, you must know all of the following:

- The plugin interface name. You must choose a supported BillingCenter plugin interface as defined in “[Summary of All BillingCenter Plugins](#)” on page 153. You cannot create your own plugin interfaces.
- The fully-qualified class name of your concrete plugin implementation class.
- Any initialization parameters, also called *plugin parameters*. See “[Plugin Parameters](#)” on page 140.

In nearly all cases, you must only have one active enabled implementation for each plugin interface. However, there are exceptions, such as messaging plugins, encryption plugins, and startable plugins. The table of all plugin interfaces has a column that specifies whether the plugin interface supports one or many implementations. See “[Summary of All BillingCenter Plugins](#)” on page 153. If the plugin interface only supports one implementation, be sure to remove any existing registry entries before adding new ones.

As you register plugins in Studio, the interface prompts you for a *plugin name*. If the interface accepts only one implementation, the plugin name is arbitrary. However, it is the best practice to set the plugin name to match the plugin interface name and omit the package. For example, enter the name `IContactSystemPlugin`.

If the interface accepts more than one implementation, the plugin name may be important. For example, for messaging plugins, enter the plugin name when you configure the messaging destination in the separate Messaging editor in Studio. See “[Messaging Editor](#)” on page 131 in the *Configuration Guide*. For encryption plugins, if you ever change your encryption algorithm, the plugin name is the unique identifier for each encryption algorithm.

To register a plugin, in Studio in the Project window, navigate to `configuration → config → Plugins → registry`. Right-click on `registry`, and choose `New → Plugin`. For more instructions about options in the Plugins Registry editor, see “[Using the Plugins Registry Editor](#)” on page 109 in the *Configuration Guide*.

In the dialog box that appears, it will ask for the interface. If the plugin name you used is the interface name (for example, `IContactSystemPlugin`), it is best to enter the interface name to be explicit. However, you can optionally leave the interface field blank.

IMPORTANT If the interface field is blank, BillingCenter assumes the interface name matches the plugin name. You might notice that some of the plugin implementations that are pre-registered in the default configuration have that field blank for this reason.

Plugin Parameters

In the Plugins Registry editor, you can add a list of parameters as name-value pairs. The plugin implementation can use these values. For example, you might pass a server name, a port number, or other configuration information to your plugin implementation code using a parameter. Using a plugin parameter in many cases is an alternative to using hard-coded values in implementation code.

To use plugin parameters, a plugin implementation must implement the `InitializablePlugin` interface in addition to the main plugin interface. If BillingCenter detects that your plugin implementation implements `InitializablePlugin`, BillingCenter calls your plugin's `setParameters` method. That method must have exactly one argument, which is a `java.util.Map` object. In the map, the parameters names are keys in the map. See “Getting Plugin Parameters from the Plugins Registry Editor” on page 143.

By default, all plugin parameters have the same name-value pairs for all values of the servers and environment system variables. However, the Plugins registry allows optional configuration by server, by environment, or both. See “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*. For example, you could set a server name differently depending on whether you are running a development or production configuration.

For Java Plugins (Without OSGi), Define a Plugin Directory

For Java plugin implementations that do not use OSGi, the Plugins Registry editor has a field for a *plugin directory*. A *plugin directory* is where you put non-OSGI Java classes and library files. In this field, enter the name of a subdirectory in the `BillingCenter/modules/configuration/plugins` directory.

To reduce the chance of conflicts in Java classes and libraries between plugin implementations, define a unique name for a plugin directory for each plugin implementation. For details, see “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*. If you do not specify a plugin directory, the default is `shared`.

IMPORTANT OSGi plugin implementations automatically isolate code for your plugin with any necessary third-party libraries in one OSGi bundle. Therefore, for OSGi plugin implementations, you do not configure a plugin directory in the Plugins Registry editor in Studio. To deploy OSGi bundles and third-party libraries in OSGi plugins, see “Java and OSGi Support” on page 451

For details of where to deploy your Java files for non-OSGi use, see “Deploying Non-OSGi Java Classes and JARs” on page 468.

Deploying Java Files (Including Java Code Called From Gosu Plugins)

How to deploy any Java files or third-party Java libraries varies based on your plugin type:

- If your Gosu plugin implements the plugin interface but accesses third-party Java classes or libraries, you must put these files in the right places in the configuration environment. See “Deploying Non-OSGi Java Classes and JARs” on page 468. It is important to note that the plugin directory setting discussed in that section has the value `Gosu` for code called from Gosu.
- For Java plugins that do not use OSGi, first ensure you define a plugin directory. See “For Java Plugins (Without OSGi), Define a Plugin Directory” on page 140. For details of where to put your Java files, see “Deploying Non-OSGi Java Classes and JARs” on page 468.
- For OSGi plugins (Java classes deployed as OSGi bundles), deployment of your plugin files and third party libraries is very different from deploying non-OSGi Java code. See “Java and OSGi Support” on page 451 and “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 469.

IMPORTANT BillingCenter supports OSGi bundles only to implement a BillingCenter plugin interface and any of your related third-party libraries. It is unsupported to install OSGi bundles for any other purpose.

Additional Information By Plugin Type

The additional information by plugin type, see the following sections.

| Plugin type | For more information |
|-----------------------|--|
| Gosu | <ul style="list-style-type: none">“Example Gosu Plugin” on page 141 |
| Java (no OSGi) | <ul style="list-style-type: none">“Special Notes For Java Plugins” on page 142“Useful Java Plugin APIs” on page 143“Java and OSGi Support” on page 451“Calling Java from Gosu” on page 123 in the <i>Gosu Reference Guide</i> |
| OSGi (Java with OSGi) | <ul style="list-style-type: none">“Java and OSGi Support” on page 451“Accessing Entity and Typecode Data in Java” on page 455“OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 469 |

Error Handling in Plugins

Where possible, BillingCenter tries to respond appropriately to errors during a plugin call and performs a default action in some cases.

However, in some cases, there is no meaningful default behavior, so the action that triggered the plugin fails and displays an error message.

There is not one standard way of handling errors in plugins. It is highly dependent on the plugin interface and the context. Check the method signatures to see what exceptions are expected. In some cases, it might be appropriate to return an empty set or error value if the response return value supports it. Contact the Guidewire Customer Technical Support group if you have questions about specific plugin interfaces and methods.

The BillingCenter server catches any runtime exception and rolls back any related database transaction, and then displays an error in the application user interface.

Temporarily Disabling a Plugin

By default, BillingCenter calls Java plugins in the Plugins registry at the appropriate time in the application logic. To disable a plugin in the registry temporarily or permanently, navigate to the Plugins Registry editor in Studio for your plugin implementation. To disable the plugin implementation, deselect the **Enabled** checkbox. To enable the plugin again, select the **Enabled** checkbox.

Example Gosu Plugin

The most important thing to remember about implementing a plugin interface in Gosu is that Gosu versions of the interface methods look different from the Java versions in many cases. The most common problem is that if an interface’s method looks like properties, you must implement the interface as a Gosu property. See “Writing Your Plugin Implementation Class” on page 137.

This topic shows a basic Gosu plugin implementation that uses parameters. For more information about plugin parameters, see “Getting Plugin Parameters from the Plugins Registry Editor” on page 143.

The following Gosu example is a simple messaging plugin that uses parameters:

```
uses java.util.Map;
uses java.plugin;

class MyTransport implements MessageTransport, InitializablePlugin {
    private var _servername : String
```

```

// note the empty constructor. If you do provide an empty constructor, the application
// calls it as the plugin instantiates, which is before application calls setParameters
construct() {
}

override function setParameters(parameters: Map<String, String>) {
    // access values in the MAP to get parameters defined in Plugins registry in Studio
    _servername = parameters["ServerName"] as String
}

override function suspend() {}

override function shutdown() {}

override function setDestinationID(id:int) {}

override function resume() {}

override function send(message:entity.Message, transformedPayload:String) {
    print("MESSAGE SEND ===== ${message.Payload} --> ${transformedPayload}")
    message.reportAck()
}
}

```

If Your Gosu Plugin Needs Java Classes and Library Files

If your Gosu class implements the plugin interface but needs to access Java classes or libraries, you must put these files in the right places in the configuration environment. See “Deploying Non-OSGi Java Classes and JARs” on page 468. Note that the plugin directory setting discussed in that section can have the value `Gosu` for code called from Gosu. Alternatively, you can put your classes in the plugin directory called `shared`.

Special Notes For Java Plugins

If you write your plugin in Java, you must regularly regenerate the Java API libraries to compile against them. For a comparison of Gosu and Java for plugin development, see “Overview of BillingCenter Plugins” on page 136.

It is important to understand some of the special considerations for writing Java code to use within BillingCenter. For example:

- The way you access and use Guidewire business data entity instances is different between Gosu and Java. They are in different packages. Also, the way you create new entity instances is different in Java compared to Gosu.
- For plugin interface methods, remember that what Gosu exposes as properties (the `PropertyName` property) appear in Java as getter and setter methods (for example, the `getMyPropertyName` method).

For important information about writing Java in BillingCenter, see “Java and OSGi Support” on page 451. Note that deployment of OSGi plugins is very different from non-OSGi plugin deployment.

Which Libraries to Compile Java Code Against

After changes to the data model, it might be necessary to regenerate Java API libraries. See “Regenerating Integration Libraries and WSDL” on page 20.

BillingCenter creates the library JAR files at the path:

`BillingCenter/java-api/lib`

For important information about deploying Java classes and libraries, see “Java and OSGi Support” on page 451 and “Deploying Non-OSGi Java Classes and JARs” on page 468.

For more information about working with Guidewire entities from your Java code, see “Accessing Entity and Typecode Data in Java” on page 455.

Where To Put Java Class and Library Files Needed By Java Plugins

For Java plugins without OSGi, see “Deploying Non-OSGi Java Classes and JARs” on page 468.

For OSGi plugins (Java with OSGi), see “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 469.

Useful Java Plugin APIs

Entity Data from Java

For important information about working with entity data, see “Accessing Entity and Typecode Data in Java” on page 455.

Getting Current User from a Java Plugin

Sometimes it is useful to determine which user triggered a user interface action that triggered a Java plugin method call. Similarly, it is sometimes useful to determine which application user called a SOAP API triggered a Java plugin method call. In both cases, the Java plugin can use the `CurrentUserUtil` utility class.

To get the current user from a Java plugin, use the following code:

```
myCurrentUser = CurrentUserUtil.getCurrentUser().getUser();
```

Translating Typecodes

Typical BillingCenter implementations need integration code that interfaces with external systems with different typecodes values than in BillingCenter. BillingCenter provides a typecode translation system that you can configure with name/value pairs.

Configure the values using an XML file. To modify the typelist conversion configuration file or to convert typecodes from Gosu or Java, see “Mapping Typecodes to External System Codes” on page 125.

Because typecode translation may occur very frequently in Java plugin code, Java plugins can use a utility class called `TypecodeMapperUtil`.

From external systems, the web service `ITypelistAPI` translates typecodes. It has similar methods to the Java API, such as `getInternalCodeByAlias`.

For example, to translate a typecode with `TypecodeMapperUtil`, use code such as:

```
TypeKey tk = TypecodeMapperUtil.getInternalCodeByAlias("Contact", "ABC:system1", "ATTORNEY");
```

Getting Plugin Parameters from the Plugins Registry Editor

In the Studio Plugins Registry editor, you can add one or more optional parameters to pass to your plugin during initialization. For example, you could use the editor to pass server names, port numbers, timeout values, or other settings to your plugin code. The parameters are pairs of `String` values, also known as name/value pairs. BillingCenter treats all plugin parameters as text values, even if they represent numbers or other objects.

To use the plugin parameters in your plugin implementation, your plugin must implement the `InitializablePlugin` interface in addition to the main plugin interface.

If you do this, BillingCenter calls your plugin’s `setParameters` method. That method must have exactly one argument, which is a `java.util.Map` object. In the map, the parameters names are keys in the map, and they map to the values from Studio.

The following Gosu example demonstrates how to define an actual plugin that uses parameters:

```
uses java.util.Map;
uses java.plugin;
```

```

class MyTransport implements MessageTransport, InitializablePlugin {
    private var _servername : String

    // note the empty constructor. If you do provide an empty constructor, the application
    // calls it as the plugin instantiates, which is before application calls setParameters
    construct() {
    }

    override function setParameters(parameters: Map<String, String>) {
        // access values in the MAP to get parameters defined in Plugins registry in Studio
        _servername = parameters["MyServerName"] as String
    }

    // NEXT, define all your other methods required by the MAIN interface you are implementing...
    override function suspend() {}

    override function shutdown() {}

    override function setDestinationID(id:int) {}

    override function resume() {}

    override function send(message:entity.Message, transformedPayload:String) {
        print("MESSAGE SEND ===== ${message.Payload} --> ${transformedPayload}")
        message.reportAck()
    }
}

```

Getting the Local File Path of the Root Directory

For Gosu plugins and Java plugins, you can access the plugin root directory path by getting a special built-in property from the Map. For the key name for root directory, use the name in the static variable `InitializablePlugin.ROOT_DIR`. This parameter is unavailable from OSGi plugins.

Getting the Local File Path of the Temp Directory

For Gosu plugins and Java plugins, you can access the plugin temporary directory path by getting a special built-in property from the Map. For the key name to get the root directory, use the name in the static variable `InitializablePlugin.TEMP_DIR`. This parameter is unavailable from OSGi plugins.

Writing Plugin Templates For Plugins That Take Template Data

Some plugin interface methods have parameters that directly specify their data as simple objects such as `String` objects. Some plugin methods have parameters of structured data such as a `java.util.Map` objects. Some methods take entities such as `Account`. Some objects might link to other objects, resulting in a potentially large object graph.

However, some plugin interface methods take a single `String` that it must parse to access important parameters. This text data is *template data* specified as plugin method parameters called `templateData`. Template data is the output of running a Gosu template called a *plugin template*. Plugin templates always have the suffix `.gsm`.

IMPORTANT For plugin templates, the file suffix must be `.gsm`. Do not use `.gst`, which is the normal Gosu template extension (see “Template Overview” on page 359 in the *Gosu Reference Guide*).

For example, the approval adapter plugin (`IApprovalAdapter`) uses template data as a parameter in its methods.

BillingCenter passes this potentially-large `String` as a parameter to the plugin for the subset of plugins that use template data. This approach lets BillingCenter pass the plugin all necessary properties in a large data graph but with minimal data transfer.

Then, a plugin method can simply parse the text to access the properties. For plugins written in Java, it is easy to use the standard Java class called `Properties`. It can parse a `String` in that format into name/value pairs from which you can extract information using code such as: `propertiesObject.getProperty(fieldname)`.

For example, this Java code takes the `templateData` parameter encoded in the simple format described earlier, and then extracts the value of the `AddressID` property from it:

```
// Create a Java Properties object
Properties accountProperties = new Properties();

try {
    // extract the template data string and load it to the Properties object
    accountProperties.load(new ByteArrayInputStream(templateData.Bytes));
} catch (java.io.IOException IOE) {
    System.out.println("MyPluginName: bad template data");
}

// Extract properties from the Properties object
String myAddressID = accountProperties.getProperty("AddressID");
```

This Gosu code does a similar thing for a Gosu plugin as a private method within the `Gosu` class:

```
private function loadPropertiesFromTemplateData(templateData : String) : Properties
{
    var props = new Properties();
    try{
        props.load(new ByteArrayInputStream(templateData.getBytes()));
        _logger.info("The properties are : " + props);
    }
    catch (e) {
        e.printStackTrace();
        return null;
    }
    return props;
}
```

You can design any text-based data format you want to pass to the plugin in the `templateData` string. If your data is not very structured, Guidewire recommends the simple `fieldname=value` format demonstrated earlier. In some cases, it may be convenient to generate XML formatted data, which permits hierarchical structure despite being a text format. This is especially useful for communicating to external systems that require XML-formatted data. Whatever text-based format you choose to use, you can modify the associated plugin template to generate the desired XML format.

For each plugin method call that takes a `templateData` parameter, BillingCenter has a Gosu template file in `BillingCenter/modules/configuration/config/templates/plugins`. BillingCenter selects the correct plugin using a naming convention:

```
{interface name}_{entity name}.gsm
```

After the Gosu engine generates a response using the designated Gosu template, the resulting `String` passes to the plugin as the `templateData` parameter to the plugin method. Again, this is only for plugin interface methods that take a `templateData` parameter.

Plugin Registry APIs

Getting References to Plugins from Gosu

To ask the application for the currently-implemented instance of a plugin, call the `Plugins.get(INTERFACENAME)` static method and pass the plugin interface name as an argument. It returns a reference to the plugin implementation. The return result is properly statically typed so you can directly call methods on the result.

For example:

```
uses gw.plugin.Plugins

// Gosu uses type inference to know the type of the variable is IContactSystemPlugin
var plugin = Plugins.get( IContactSystemPlugin )
```

```

try{
    plugin.retrieveContact("abc:123" )
}catch(e){
    e.printStackTrace()
    throw new DisplayableException(e.Message)
}

```

Alternatively, you can request a plugin by the plugin name in the Plugins registry. This is important if there is more than one plugin implementation of the interface. For example, this is common in messaging plugins. To do this, use an alternative method signature of the get method that takes a `String` for the plugin name as defined in the Studio plugin configuration.

For example, if your plugin was called `MyPluginName` and the interface name is `IStartablePlugin`:

```

uses gw.plugin.Plugins

// you must downcast to the plugin interface
var contactSystem = Plugins.get( "MyContactPluginRegistryName" ) as IContactSystemPlugin

try{
    contactSystem.retrieveContact("abc:123")
}catch(e){
    e.printStackTrace()
    throw new DisplayableException(e.Message)
}

```

Check Is Enabled

From Gosu you can use the `Plugins` class to determine if a plugin is enabled in Studio. Call the `isEnabled` method on the `Plugins` class and pass either of the following as a `String` value:

- the interface name type (with no package)
- the plugin implementation name in the Plugins registry

For example:

```

uses gw.plugin.Plugins

var contactSystemEnabled = Plugins.isEnabled( IContactSystemPlugin )

```

Getting References to Java Plugins from Java

From Java you can get references to other installed plugins either by class or by the string representation of the class. Do this using methods on the `PluginRegistry` class within the `guidewire.pl.plugin` package. Get the plugin by class by calling the `getPlugin(Class)` method. Get the plugin by name calling the `getPluginByName(String)` method.

This is useful if you want to access one plugin from another plugin. For example, a messaging-related plugin that you write might need a reference to another messaging-related plugin to communicate or share common code.

Also, this allows you to access plugin interfaces that provide services to plugins or other Java code. For example, the `IScriptHost` plugin can evaluate Gosu expressions. To use it, get a reference to the currently-installed `IScriptHost` plugin. Next, call its methods. Call the `putSymbol` method to make a Gosu context symbol such account to evaluate to a specific `Account` object reference. Call the `evaluate` method to evaluate a `String` containing Gosu code.

IMPORTANT Do not use this API from Gosu. From Gosu, call the `Plugins.get(INTERFACENAME)` static method and pass the plugin interface name or plugin name as an argument. See earlier in this topic for details.

Plugin Thread Safety

If you register a Java plugin or a Gosu plugin, exactly one instance of that plugin exists in the Java virtual machine on that server, generally speaking. For example, if you register a document production plugin, exactly one instance of that plugin instantiates on each server.

The rules are different for messaging plugins in BillingCenter server clusters. Messaging plugins instantiate only on the batch server. The other non-batch servers have zero instances of message request, message transport, and message reply plugins. See “[Messaging Flow Details](#)” on page 309. Messaging plugins must be especially careful about thread safety because messaging supports a large number of simultaneous threads, configured in Studio.

However, one server instance of the Java plugin or Gosu plugin must service multiple user sessions. Because multiple user sessions use multiple process threads, follow these rules to avoid thread problems:

- Your plugin must support multiple simultaneous calls to the same plugin method from different threads. You must ensure multiple calls to the plugin never access the same shared data. Alternatively, protect access to shared resources so that two threads do not access it simultaneously.
- Your plugin must support multiple simultaneous calls to the plugin in general. For example, BillingCenter might call two different plugin methods at the same time. You must ensure multiple method calls to the plugin never access the same shared data. Alternatively, protect access to shared resources so that two threads do not access it simultaneously.
- Your plugin implementation must support multiple user sessions. Generally speaking, do not assume shared data or temporary storage is unique to one user request (one HTTP request of a single user).

Collectively, these requirements describe thread safety. You must ensure your implementation is thread safe.

IMPORTANT For important information about concurrency, see “[Concurrency](#)” on page 381 in the *Gosu Reference Guide*.

The most important way to avoid thread safety problems in plugin implementations is to avoid variables stored once per class, referred to as *static variables*. Static variables are a feature of both the Java language and the Gosu language. Static variables let a class store a value once per class, initialized only once. In contrast, object *instance variables* exist once per instance of the class.

Static variables can be extremely dangerous in a multi-threaded environment. Using static variables in a plugin can cause serious problems in a production deployment without taking great care to avoid problems. Be aware that such problems, if they occur, are extremely difficult to diagnose and debug. Timing in a multi-user multi-threaded environment is difficult, if not impossible, to control in a testing environment.

Because plugins could be called from multiple threads, there is sometimes no obvious place to store temporary data that stores state information. Where possible and appropriate, replace static variables with other mechanisms, such as setting properties on the relevant data passed as parameters. For example, in some cases perhaps use a data model extension property on an Account or other relevant entity (including custom entities) to store state-specific data for the plugin. Be aware that storing data in an entity shares the data across servers in a BillingCenter cluster (see “[Design Plugin Implementations to Support Server Clusters](#)” on page 151). Additionally, even standard instance variables (not just static variables) can be dangerous because there is only one instance of the plugin.

If you are experienced with multi-threaded programming and you are certain that static variables are necessary, you must ensure that you *synchronize* access to static variables. Synchronization refers to a feature of Java (but not natively in Gosu) that locks access between threads to shared resources such as static variables.

WARNING Avoid static variables in plugins if at all possible. BillingCenter may call plugins from multiple process threads and in some cases this could be dangerous and unreliable. Additionally, this type of problem is extremely difficult to diagnose and debug.

For important information about concurrency, see “Concurrency” on page 381 in the *Gosu Reference Guide*.

For more information about concurrency and related APIs in Java, see:

<http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>

The following sections list some common approaches for thread safety with static variable in Java:

- “Using Java Concurrent Data Types, Even from Gosu” on page 148
- “Using Synchronized Methods (Java Only)” on page 149
- “Using Java Synchronized Blocks of Code (Java only)” on page 150

For thread safety issues in Gosu, see “Gosu Static Variables and Gosu Thread Safety” on page 148.

Additionally, note some similar issues related to multi-server (cluster) plugin design in “Design Plugin Implementations to Support Server Clusters” on page 151.

Gosu Static Variables and Gosu Thread Safety

The challenges of static variables and thread safety applies to Gosu classes, not just Java. This affects Gosu in plugin code and also for Gosu classes triggered from rules sets. The most important thing to know is that static variables present special challenges to ensure your code is thread safe.

The typical way to create a Gosu class with static variable is with code like:

```
class MyClass {  
    static var _property1 : String;  
}
```

You must be as careful in Gosu with static variables and synchronizing data in them to be thread safe as in Java. Use the Java concurrent data types describes in this section to ensure safe access.

WARNING Thread safety APIs that use blocking can affect performance negatively. For highest performance, use such APIs wisely and test your code under heavy loads that test the concurrency.

For important information about concurrency, see “Concurrency” on page 381 in the *Gosu Reference Guide*.

Using Java Concurrent Data Types, Even from Gosu

The simplest way to synchronizing access to a static variable in Java is to store data as an instance of a Java classes defined in the package `java.util.concurrent`. The objects in that package automatically implement synchronization of their data, and no additional code or syntax is necessary to keep all access to this data thread-safe. For example, to store a mapping between keys and values, instead of using a standard Java `HashMap` object, instead use `java.util.concurrent.ConcurrentHashMap`.

These tools protect the integrity of the keys and values in the map. However, you must ensure that if multiple threads or user sessions use the plugin, the business logic still does something appropriate with shared data. You must test the logic under multi-user and multi-thread situations.

WARNING All thread safety APIs that use blocking can affect performance negatively. For high performance, use such APIs carefully and test all code under heavy loads that test the concurrency.

For important information about concurrency, see “Concurrency” on page 381 in the *Gosu Reference Guide*.

Using Synchronized Methods (Java Only)

Java provides a feature called synchronization that protects shared access to static variables. It lets you tag some or all methods so that no more than one of these methods can be run at once. Then, you can add code safely to these methods that get or set the object’s static class variables, and such access are thread safe.

If an object is visible to more than one thread, and one thread is running a synchronized method, the object is locked. If an object is locked, other threads cannot run a synchronized method of that object until the lock releases. If a second thread starts a synchronized method before the original thread finishes running a synchronized method on the same object, the second thread waits until the first thread finishes. This is known as *blocking* or *suspending execution* until the original thread is done with the object.

Mark one or more methods with this special status by applying the `synchronized` keyword in the method definition. This example shows a simple class with two synchronized methods that use a static class variable:

```
public class SyncExample {  
    private static int contents;  
  
    public int get() {  
        return contents;  
    }  
  
    // Define a synchronized method. Only one thread can run a syncced method at one time for this object  
    public synchronized void put1(int value) {  
        contents = value;  
        // do some other action here perhaps...  
    }  
  
    // Define a synchronized method. Only one thread can run a syncced method at one time for this object  
    public synchronized void put2(int value) {  
        contents = value;  
        // do some other action here perhaps...  
    }  
}
```

Synchronization protects invocations of all synchronized methods on the object: it is not possible for invocations of two different synchronized methods on the same object to interleave. For the earlier example, the Java virtual machine does all of the following:

- Prevents two threads simultaneously running `put1` at the same time
- Prevents `put1` from running while `put2` is still running
- Prevents `put2` from running while `put1` is still running.

This approach protects integrity of access to the shared data. However, you must still ensure that if multiple threads or user sessions use the plugin, your code does something appropriate with this shared data. Always test your business logic under multi-user and multi-thread situations.

BillingCenter calls the plugin method initialization method `setParameters` exactly once, hence only by one thread, so that method is automatically safe. The `setParameters` method is a special method that BillingCenter calls during plugin initialization. This method takes a Map with initialization parameters that you specify in the Plugins registry in Studio. For more information about plugin parameters, see “Special Notes For Java Plugins” on page 142.

On a related note, Java class constructors cannot be synchronized; using the Java keyword `synchronized` with a constructor generates a syntax error. Synchronizing constructors does not make sense because only the thread that creates an object has access to during the time Java is constructing it.

For important information about concurrency, see “Concurrency” on page 381 in the *Gosu Reference Guide*.

Using Java Synchronized Blocks of Code (Java only)

Java code can also synchronize access to shared resources by defining a block of statements that can only be run by one thread at a time. If a second thread starts that block of code, it waits until the first thread is done before continuing. Compared to the method locking approach described earlier in this section, synchronizing a block of statements allows much smaller granularity for locking.

To synchronize a block of statements, use the `synchronized` keyword and pass it a Java object or class identifier. In the context of protecting access to static variables, always pass the class identifier `ClassName.class` for the class hosting the static variables.

For example, this demonstrates statement-level or block-level synchronization:

```
class MyPluginClass implements IMyPluginInterface {  
    private static byte[] myLock = new byte[0];  
  
    public void MyMethod(Address f){  
        // SYNCHRONIZE ACCESS TO SHARED DATA!  
        synchronized(MyPluginClass.class){  
            // Code to lock is here....  
        }  
    }  
}
```

This finer granularity of locking reduces the frequency that one thread is waiting for another to complete some action. Depending on the type of code and real-world use cases, this finer granularity could improve performance greatly over using synchronized methods. This is particularly the case if there are many threads. However, you might be able to refactor your code to convert blocks of synchronized statements into separate synchronized methods. See “Using Synchronized Methods (Java Only)” on page 149.

Both approaches protect integrity of access to the shared data. However, you must plan to handle multiple threads or user sessions to use your plugin, and do safely access any shared data. Also, test your business logic under realistic heavy loads for multi-user and multi-thread situations.

WARNING Thread safety APIs that use blocking can affect performance negatively. For highest performance, use such APIs wisely and test your code under heavy loads that test the concurrency.

For important information about concurrency, see “Concurrency” on page 381 in the *Gosu Reference Guide*.

Avoid Singletons Due to Thread-Safety Issues

The thread safety problems discussed in the previous section apply to any Java object that has only a single instance (also referred to as a *singleton*) implemented using static variables. Because static variable access in multi-threaded code is complex, Guidewire strongly discourages using singleton Java classes. You must synchronize access to all data singleton instances just as for other static variables as described earlier in this section. This restriction is important for all Gosu Java that BillingCenter runs.

This is an example of creating a singleton using a class static variable:

```
public class MySingleton {  
    private static MySingleton _instance =  
        new MySingleton();  
  
    private MySingleton() {  
        // construct object . . .  
    }  
  
    public static MySingleton getInstance() {  
        return _instance;  
    }  
}
```

For more information about singletons in Java, see:

<http://java.sun.com/developer/technicalArticles/Programming/singletons>

If you absolutely must use a singleton, you must synchronize updates to class static variables as discussed at the beginning of “Plugin Thread Safety” on page 147.

WARNING Avoid creating singletons, which are classes that enforce only a single instance of the class. If you really must use singletons, you must use the synchronization techniques discussed in “Plugin Thread Safety” on page 147 to be thread safe.

For important other information about concurrency, see “Concurrency” on page 381 in the *Gosu Reference Guide*.

Design Plugin Implementations to Support Server Clusters

Generally speaking, if your plugin deploys in a BillingCenter server cluster, there are instances of the plugin deployed on every server in the cluster. Consequently, design your plugin code (and any associated integration code) to support concurrent instances. If the Gosu code calls out to Java for any network connections, that code must support concurrent connections.

Note: There is an exception for this cluster rule: messaging plugins exist only for the single server designated the batch server. See “Messaging Flow Details” on page 309.

Because there may be multiple instances of the plugin, you must ensure that you update a database from Java code carefully. Your code must be thread safe, handle errors fully, and operate logically for database transactions in interactions with external systems. For example, if several updates to a database must be treated as one action or several pieces of data must be modified as one atomic action, design your code accordingly.

The thread safety synchronization techniques in “Plugin Thread Safety” on page 147 are insufficient to synchronize data shared across multiple servers in a cluster. Each server has its own Java virtual machine, so it has its own data space. Write your plugins to know about the other server’s plugins but not to rely on anything other than the database to communicate among each other across servers.

You must implement your own approach to ensure access to shared resources safely even if accessed simultaneously by multiple threads and on multiple servers.

For important information about concurrency, see “Concurrency” on page 381 in the *Gosu Reference Guide*.

Reading System Properties in Plugins

You might want to test plugins in multiple deployment environments without recompiling plugins. For example, perhaps if a plugin runs on a test server, then the plugin queries a test database. If it runs on a production server, then the plugin queries a production database.

Or, you might want a plugin that can be run on multiple machines within a cluster with each machine knowing its identity. You might want to implement unique behavior within the cluster. Alternatively, add this information to log files on the local machine and in the external system also.

For these cases, plugins can use environment (`env`) and server ID (`serverid`) deployment properties to deploy a single plugin with different behaviors in multiple contexts or across clustered servers. Define these system properties in the server configuration file or as command-line parameters for the command that launches your web server container. In general, use the default system property settings. If you want to customize them, use the Plugins Registry editor in Studio.

Gosu plugins can query system properties using the methods `getEnv`, `getServerId` and `isBatchServer`, all on the `ServerUtil` class. For example the following Gosu expression evaluate to the current value of the `env` system property, the server ID, and whether this server is the batch server:

```
var envValue = gw.api.system.server.ServerUtil.getEnv("gw.bc.env")
var serverID = gw.api.system.server.ServerUtil.getServerId()
var isBatchServer = gw.api.system.server.ServerUtil.isBatchServer()
```

Java plugins can query system properties using code such as the following example, which gets the `env` property:

```
java.lang.System.getProperty("gw.bc.env");
```

See also

- “Clustering Application Servers” on page 73 in the *System Administration Guide*.
- For more information about `env` and `serverid` settings, see “Specifying Environment Properties in the `<registry>` Element” on page 15 in the *System Administration Guide*.
- “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*.

Do Not Call Local Web Services From Plugins

Do not call locally-hosted web service APIs (SOAP APIs) from within a plugin or the rules engine in production systems. If the web service hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data for your plugin implementation. Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance.

If you have questions about how to refactor your code to avoid local loopback API calls over web services, contact Guidewire Customer Support.

Creating Unique Numbers in a Sequence

Typical BillingCenter implementations need to reliably create unique numbers in a sequence for some types of objects. For example, to enforce a series of unique IDs, such as public ID values, within a sequence. You can generate new numbers in a sequence using sequence generator APIs.

These methods take two parameters:

- An initial value for the sequence, if it does not yet exist.
- A `String` with up to 256 characters that uniquely identifies the sequence. This is the sequence key (`sequenceKey`).

For example, suppose you want to get a new number from a sequence called `WidgetNumber`.

In Gosu, use code like the following:

```
nextNum = gw.api.system.database.SequenceUtil.next(10, "WidgetNumber")
```

If this is the first time any code has requested a number in this sequence, the value is 1. If other code calls this method again, the return value is two, three, or some larger number depending on how many times any code requested numbers for this sequence key.

Similarly, in Java, use the code:

```
nextNum = gw.api.system.database.SequenceUtil.next(10, "WidgetNumber");
```

Restarting and Testing Tips for Plugin Developers

If you frequently modify your plugin code, you might need to frequently redeploy BillingCenter to test your plugins. If it is a non-production server, you may not want to shut down the entire web application container and restart it. For development (non-product) use only, reload only the BillingCenter application rather than the web application container. If your web application container supports this, replace your plugin class files and reload the application.

For example, Apache Tomcat provides a web application called Manager that provides this functionality. For documentation on this Apache Tomcat Manager, refer to:

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/manager-howto.html>

Summary of All BillingCenter Plugins

The following table summarizes the plugin interfaces that BillingCenter defines. For a table that summarizes the plugin interfaces that ContactManager defines, see “ContactManager Plugins” on page 251 in the *Contact Management Guide*.

| BillingCenter Plugin Interface | Description | Maximum enabled implementations |
|------------------------------------|---|---------------------------------|
| Contact-related plugins | | |
| IContactSystemPlugin | Integrates with a centralized contact system. See “Integrating with a Contact Management System” on page 395. | 1 |
| OfficialIdToTaxIdMappingPlugin | Determines whether BillingCenter treats an official ID type as a tax ID for contacts. See “Official IDs Mapped to Tax IDs Plugin” on page 268. | 1 |
| Other BillingCenter plugins | | |
| IAccount | Customizes the distribution limit for a given account. See “Account Distribution Limit Plugin” on page 187. | 1 |
| IAccountEvaluationCalculator | Customizes BillingCenter account evaluation. See “Account Evaluation Calculation Plugin” on page 160. | 1 |
| IAccountInfo | Customizes logic associated with a non-persistent AccountInfo entity before returning it to an external system using the web BCAPI service method getAccountInfo. See “Account Information Customization Plugin” on page 187. | 1 |
| IAgencyCycleDist | Customizes application logic related to making Agency Payments. See “Agency Cycle Distribution Pre-fill Customization Plugin” on page 184. | 1 |
| IAgencyDistributionDisposition | Provides hooks for the customization of how distribution exceptions are dealt with during execution of an agency distribution. See “Agency Distribution Disposition Plugin” on page 189. | 1 |
| IBillingInstruction | Customizes how to create invoice items from a charge and how to handle a billing instruction. See “Billing Instruction Execution Customization Plugin” on page 167. | 1 |
| ICollateral | Customizes how to deal with collateral cash requirements. See “Collateral Cash Plugin” on page 188. | 1 |
| ICommission | Customizes the selection of a commission rates and subplans given a policy period. See “Commission Plugin” on page 162. | 1 |

| BillingCenter Plugin Interface | Description | Maximum enabled implementations |
|--------------------------------|---|---------------------------------|
| ICommissionOverride | For agency bill, determines whether to change the commission amount of an invoice item or leave it untouched and create a commission adjustment invoice item. See “Commission Plugins” on page 162. | 1 |
| IDateSequence | Customizes how BillingCenter creates date sequences. See “Date Sequence Plugin” on page 183. | 1 |
| IDelinquencyProcessExtensions | Customizes BillingCenter delinquency processing, specifically determining in what conditions to pay charges and how to push forward held events. See “Delinquency Processing Customization Plugin” on page 165. | 1 |
| IDirectBillPayment | Customizes how BillingCenter handles direct bill payments. See “Direct Bill Payment Plugin” on page 192. | 1 |
| IEventHandler | <p>Provides miscellaneous hooks into BillingCenter flow:</p> <ul style="list-style-type: none"> • determining whether commissions are payable • determining whether the delinquency process starts • determining in which cases to close policies • custom processing during policy transfer to new producers • validation credit card numbers • validating manual refunds • determining whether to place holds on accounts • determining whether to pay disbursements. | 1 |
| IFundsTracking | The name of this plugin includes “event”. However, “event” in this context is a general term unrelated to the messaging event system. See “BillingCenter Application Event Customization Plugin” on page 170. | 1 |
| IIIncentiveCalculator | Customizes BillingCenter evaluation of <i>commission incentives</i> . Incentives are fully customizable and can cover cases that the normal commission processing cannot handle. See “Incentive Calculation Plugin” on page 162. | 1 |
| IIInvoice | Customizes default invoice-related behaviors, such as calculating an installment fee or determining whether to carry an invoice forward. See “Invoice Plugin” on page 176. | 1 |
| IIInvoiceAssembler | Customizes how BillingCenter generates invoice items and installment event dates. See “Invoice Assembler Plugin” on page 177. | 1 |
| IIInvoiceItem | Customizes exception information for invoice items that change. See “Invoice Item Exception Information Plugin” on page 182. | 1 |
| IIInvoiceStream | Customizes how BillingCenter chooses the right invoice stream, based on the account, the charge, and the payment plan. See “Invoice Stream Plugin” on page 180. | 1 |
| INumberGenerator | Customizes BillingCenter number generation, such as generating new account numbers. See “Numbers and Sequences Plugin” on page 166. | 1 |
| IPaymentPlan | Customizes how BillingCenter generates invoice items and installment event dates. See “Payment Plan Plugin” on page 174. | 1 |

| BillingCenter Plugin Interface | Description | Maximum enabled implementations |
|--|---|---------------------------------|
| IPolicyPeriod | Provides a hook to add information to the non-persistent entity PolicyPeriodInfo (or add special logic) before it returning it to web service clients. This effectively customizes the behavior of the BCAPI web service interface method getPolicyPeriodInfo. See “Policy Period Information Customization Plugin” on page 168. | 1 |
| IPolicySystemPlugin | Most BillingCenter integration points with a policy system start in the policy system, not BillingCenter. For example, if a user adds a new policy or changes a policy in the policy administration system, it sends the billing implications of the change to BillingCenter. However, there are a few times in the application flow that BillingCenter needs to initiate a request to the policy system. For these cases, BillingCenter calls out to the currently registered implementation of the policy system plugin (IPolicySystemPlugin). For example, if an insured fails to pay, BillingCenter tells the policy system to start policy cancellation. See “Policy System Plugin” on page 189. | 1 |
| IPremiumReport | Customizes logic related to the premium report billing instruction. See “Premium Report Customization Plugin” on page 185. | 1 |
| IProducerInfo | Customizes logic related to the non-persistent entity ProducerInfo. See “Producer Information Customization Plugin” on page 187. | 1 |
| ISuspensePayment | Customizes how BillingCenter handles an incoming suspense payments. A suspense payment is a payment for which either the account or policy number is missing, or is a renewal offer payment with an unknown renewal offer number. See “Direct Bill Payment Plugin” on page 192 for the plugin, or “Policy System Plugin” on page 189 for more information about renewal offers. | 1 |
| ISystemParameters | Customizes BillingCenter system parameters such as getting the threshold for making automatic producer payments and getting dynamic values for certain date-based calculations. See “BillingCenter Parameter Calculation Plugin” on page 167. | 1 |
| Authentication plugins (see “Authentication Integration” on page 207) | | |
| AuthenticationServicePlugin | The authentication service plugin authorizes a user from a remote authentication source, such as a corporate LDAP or other single-source sign-on system. | 1 |
| AuthenticationSource | A marker interface representing an authentication source for user interface login. The implementation of this interface must provide data to the authentication service plugin that you register in BillingCenter. All classes that implement this interface must be serializable. Any object contained with those objects must be serializable as well. | 1 |
| AuthenticationSourceCreatorPlugin | For WS-I web services authentication, see the row in this table for WebservicesAuthenticationPlugin. | 1 |
| AuthenticationSourceCreatorPlugin | Creates an authentication source (an AuthenticationSource) for user interface login. This takes an HTTP protocol request (from an HttpRequest object). The authorization source must work with your registered implementation of the AuthenticationServicePlugin plugin interface. | 1 |

| BillingCenter Plugin Interface | Description | Maximum enabled implementations |
|--|---|---------------------------------|
| DBAuthenticationPlugin | Allows you to store the database username and password in a way other than plain text in the config.xml file. For example, retrieve it from an external system, decrypt the password, or read a file from the file system. The resulting username and password substitutes into the database configuration for each instance of that \${username} or \${password} in the database parameters. | 1 |
| WebservicesAuthenticationPlugin | For WS-I web services only, configures custom authentication logic. This plugin interface is documented with other WS-I information. See "Web Services Authentication Plugin" on page 50. | 1 |
| Document content and metadata plugins (see "Document Management" on page 217) | | |
| IDocumentContentSource | Provides access to a remote repository of documents, for example to retrieve a document, store a document, or remove a document from a remote repository. BillingCenter implements this with a default version, but for maximum data integrity and feature set, implement this plugin and link it to a commercial document management system. | 1 |
| IDocumentDataSource | Stores metadata associated with a document, typically to store it in a remote document management system; see IDocumentContentSource mentioned earlier for a related plugin. By default, BillingCenter stores the metadata locally in the BillingCenter database if you do not define it. For maximum data integrity and feature set, implement this plugin and link it to a commercial document management system. | 1 |
| Document production plugins (see "Document Production" on page 231) | | |
| IDocumentProduction | Generates documents from a template. For example, from a Gosu template or a Microsoft Word template. This plugin can create documents synchronously and/or asynchronously. | 1 (see note) |
| | Note: Only register one implementation of this plugin. However, there are multiple other classes that implement this interface and handle one document type. The registered implementation of this plugin interface dispatches requests to the other classes that implement this same interface. See "Document Production" on page 231. | |
| IDocumentTemplateSource | Provides access to a repository of document templates that can generate forms and letters, or other merged documents. An implementation may simply store templates in a local repository. A more sophisticated implementation might interface with a remote document management system. | 1 |
| IEmailTemplateSource | Gets email templates. By default, BillingCenter stores the email templates on the server but you can customize this behavior by implementing this plugin. | 1 |
| IDocumentTemplateSerializer | <i>Use the built-in version of this plugin using the "Plugins registry" but generally it is best not implement your own version.</i> This plugin serializes and deserializes document template descriptors. Typically, descriptors persist as XML, as such implementations of this class understand the format of document template descriptors and can read and write them as XML. | 1 |

| BillingCenter Plugin Interface | Description | Maximum enabled implementations |
|---|--|---------------------------------|
| Messaging plugins (see “Messaging and Events” on page 303) | | |
| MessageTransport | <p>This is the main messaging plugin interface. This plugin sends a message to an external/remote system using any transport protocol. This could involve submitting to a messaging queue, calling a remote API call, saving to special files in the file system, sending e-mails, or anything else you require. Optionally, this plugin can also acknowledge the message if it is capable of sending synchronously (that is, as part of a synchronous send request).</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems. To distinguish them, as you create the plugin in Studio, Studio prompts you for a name for the plugin. That is called the <i>plugin name</i>. Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio.</p> | Multiple |
| MessageRequest | <p>Optional pre-processing of messages, and optional post-send-processing (separate from post-acknowledgement processing).</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems. To distinguish them, as you create the plugin in Studio, Studio prompts you for a name for the plugin. That is called the <i>plugin name</i>. Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio.</p> | Multiple |
| MessageReply | <p>Handles asynchronous acknowledgements of a message. After submitting an acknowledgement to optionally handles other post-processing afterward such as property updates. If you can send the message synchronously, do not implement this plugin. Instead, implement only the transport plugin and acknowledge each message immediately after it sends the message.</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems. To distinguish them, as you create the plugin in Studio, Studio prompts you for a name for the plugin. That is called the <i>plugin name</i>. Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio.</p> | Multiple |
| Other plugins | | |
| InboundIntegrationStartablePlugin | High performance inbound integrations, with support for multi-threaded processing of work items. See “Multi-threaded Inbound Integration” on page 281 | Multiple |
| ITestingClock | <p>Used for testing complex behavior over a long span of time, such as multiple billing cycles or timeouts that are multiple days or weeks later. This plugin is <i>for development (non-production) use only</i>. It programmatically changes the system time to accelerate the perceived passing of time within BillingCenter.</p> <p>WARNING The testing clock plugin is for application development only. You must never use it on a production server. See “Testing Clock Plugin (Only For Non-Production Servers)” on page 266.</p> | 1 |
| IAddressAutocompletePlugin | Configures how address automatic completion and fill-in operate. See “Automatic Address Completion and Fill-in Plugin” on page 265. | 1 |
| IPhoneNormalizerPlugin | Normalizes phone numbers that users enter through the application and that enter the database through data import. See “Phone Number Normalizer Plugin” on page 266. | 1 |

| BillingCenter Plugin Interface | Description | Maximum enabled implementations |
|--|--|---------------------------------|
| IEncryptionPlugin | Encodes or decodes a String based on an algorithm you provide to hide important data, such as bank account numbers or private personal data. BillingCenter does not provide any encryption algorithm in the product. BillingCenter simply calls this plugin implementation, which is responsible for encoding an unencrypted String or reversing that process. The built-in implementation of this plugin does nothing. See “Encryption Integration” on page 253. | Multiple |
| | The IEncryption plugin interface supports multiple implementations to support changes to encryption algorithms. See “Changing Your Encryption Algorithm Later” on page 258. | |
| IBaseURLBuilder | Generates a base URL to use for web application pages affiliated with this application, given the HTTP servlet request URI (<code>HttpServletRequest</code>). See “Defining Base URLs for Fully-Qualified Domain Names” on page 270. | 1 |
| ManagementPlugin | The external management interface for BillingCenter, which allows you to implement management systems such as JMX, SNMP, and so on. See “Management Integration” on page 261. | 1 |
| IScriptHost | <i>You can use the built-in version of this plugin using the Plugins registry APIs but do not attempt to implement your own version.</i> Get the current implementation using the Plugins registry and use it to evaluate Gosu code to provide context-sensitive data to other services. For example, property data paths defined with Gosu expressions. For example, this service could evaluate a String that has the Gosu expression <code>"policy.myFieldName"</code> at run time. | 1 |
| IProcessesPlugin | Instantiates custom batch processing classes so they can be run on a schedule or on demand. See “Implementing the Processes Plugin” on page 429. | 1 |
| IStartablePlugin | Creates new plugins that immediately instantiate and run on server startup. You can register multiple startable plugin implementations for different tasks. See “Startable Plugins Overview” on page 273. | Multiple |
| IPreupdateHandler | Implements your preupdate handling in plugin code rather than in the built-in rules engine. See “Preupdate Handler Plugin” on page 268. | 1 |
| IWorkItemPriorityPlugin | Calculates the processing priority of a work item. See “Work Item Priority Plugin” on page 268. | 1 |
| IActivityEscalationPlugin | Overrides the behavior of activity escalation instead of simply calling rule sets. See “Exception and Escalation Plugins” on page 271. | 1 |
| IGroupExceptionPlugin | Overrides the behavior of group exceptions instead of simply calling rule sets. See “Exception and Escalation Plugins” on page 271. | 1 |
| IUserExceptionPlugin | Overrides the behavior of user exceptions instead of simply calling rule sets. See “Exception and Escalation Plugins” on page 271. | 1 |
| ClusterBroadcastTransportFactory ClusterFastBroadcastTransportFactory ClusterUnicastTransportFactory | <i>For internal use only.</i> A plugin implementation is registered in the Plugins registry in the default configuration for these plugin interfaces. However, they are unsupported for customer use. Do not change settings in the Plugins registry for these interfaces. Do not use or implement these plugin interfaces. | n/a |

Billing Plugins

This topic describes plugin interfaces that BillingCenter calls to perform tasks or calculate values related to billing.

This topic includes:

- “Account Evaluation Calculation Plugin” on page 160
- “Incentive Calculation Plugin” on page 162
- “Commission Plugins” on page 162
- “Delinquency Processing Customization Plugin” on page 165
- “Numbers and Sequences Plugin” on page 166
- “BillingCenter Parameter Calculation Plugin” on page 167
- “Billing Instruction Execution Customization Plugin” on page 167
- “Policy Period Information Customization Plugin” on page 168
- “BillingCenter Application Event Customization Plugin” on page 170
- “ChargeInitializer Plugin” on page 172
- “Payment Plan Plugin” on page 174
- “Invoice Plugin” on page 176
- “Invoice Assembler Plugin” on page 177
- “Invoice Stream Plugin” on page 180
- “Invoice Item Exception Information Plugin” on page 182
- “Date Sequence Plugin” on page 183
- “Agency Cycle Distribution Pre-fill Customization Plugin” on page 184
- “Premium Report Customization Plugin” on page 185
- “Account Information Customization Plugin” on page 187
- “Producer Information Customization Plugin” on page 187
- “Account Distribution Limit Plugin” on page 187
- “Collateral Cash Plugin” on page 188

- “Agency Distribution Disposition Plugin” on page 189
- “Policy System Plugin” on page 189
- “Direct Bill Payment Plugin” on page 192
- “Suspense Payment Plugin” on page 197
- “Funds Tracking Plugin” on page 198

See also

- For general information about plugins, see “Plugin Overview” on page 135.
- For a list of all BillingCenter plugin interfaces, see “Summary of All BillingCenter Plugins” on page 153.
- For detailed information about the integration of PolicyCenter and BillingCenter, see “Billing Integration” in the *PolicyCenter Integration Guide*.

Account Evaluation Calculation Plugin

You can customize BillingCenter account evaluation by implementing the `IAccountEvaluationCalculator` plugin. For general information about account evaluation, see “Account Evaluation” on page 265 in the *Application Guide*.

You must implement several interface methods to perform the following tasks.

To compute an evaluation rating for an account, implement the `calculateEvaluation` method. It takes information about how many negative events occurred in a specified time period. It takes the number of delinquencies, insufficient funds occurrences, and cancellations in that time period. It returns results in an `AccountEvaluation` entity.

To decide how to count the number of delinquencies, implement the `countUniqueDelinquenciesAndCancellationsOnly` method. It returns `true` or `false`.

If the method returns `true`, only unique instances of delinquency count if calculating the number of delinquencies and the number of policy cancellations metrics for account evaluation. For example, suppose you had an account with four policies. If one payment causes all four policies to be delinquent (or cancelled), if this method returns `true` the problem calculates only as one delinquency (or cancellation).

If the method returns `false`, BillingCenter uses only the raw counts of delinquencies and policy cancellations. In that earlier example, that would count as four delinquencies (or cancellations) rather than one.

To customize how many days to search for negative events, implement the `getMetricsQueryTimePeriod` method. As part of account evaluation, the system queries the number of negative events recently. BillingCenter must query for delinquencies, cancellations, and pejorative payment reversals on the account in recent days. This method returns that number of days before today to use for the query.

Refer to the Javadoc for `IAccountEvaluationCalculator` for method signature details.

The following example implements this plugin interface as a Java plugin:

```
public class AccountEvaluationCalculator implements IAccountEvaluationCalculator {  
    // Time period in days over which system runs queries for  
    // various pieces of information you use in  
    // determining the account evaluation rating (for example: how many delinquencies  
    // on the account in the last 365 days)  
    private static final int NumberOfDaysForSearch = 365;  
  
    // Thresholds for number of delinquencies on the account  
    private static final int MaxDelinquenciesForExcellent = 1;  
    private static final int MaxDelinquenciesForGood = 2;  
    private static final int MaxDelinquenciesForAcceptable = 4;  
    private static final int MaxDelinquenciesForMarginal = 6;  
  
    // Thresholds for number of occurrences of pejorative payment reversals on the account  
    private static final int MaxPaymentReversalsForExcellent = 0;
```

```
private static final int MaxPaymentReversalsForGood = 0;
private static final int MaxPaymentReversalsForAcceptable = 1;
private static final int MaxPaymentReversalsForMarginal = 2;

// Thresholds for number of policy cancellations on the account
private static final int MaxPolicyCancellationsForExcellent = 0;
private static final int MaxPolicyCancellationsForGood = 1;
private static final int MaxPolicyCancellationsForAcceptable = 2;
private static final int MaxPolicyCancellationsForMarginal = 3;

// give an account an evaluation of New Account if it is less than a year old
private static final long MaxAgeOfNewAccountInMilliseconds = 365L * 24L * 60L * 60L * 1000L;

public AccountEvaluation calculateEvaluation(Account account,
                                             Date currentTime,
                                             int numberDelinquenciesInTimePeriod,
                                             int numberPaymentReversalsInQueryPeriod,
                                             int numberPolicyCancellationsInTimePeriod) {

    if (currentTime.getTime() - account.getCreateTime().getTime() <= MaxAgeOfNewAccountInMilliseconds)
    {
        return AccountEvaluation.NEWACCOUNT;
    }
    else if (numberDelinquenciesInTimePeriod <= MaxDelinquenciesForExcellent &&
              numberPaymentReversalsInQueryPeriod <= MaxPaymentReversalsForExcellent &&
              numberPolicyCancellationsInTimePeriod <= MaxPolicyCancellationsForExcellent) {
        return AccountEvaluation.EXCELLENT;
    }
    else if (numberDelinquenciesInTimePeriod <= MaxDelinquenciesForGood &&
              numberPaymentReversalsInQueryPeriod <= MaxPaymentReversalsForGood &&
              numberPolicyCancellationsInTimePeriod <= MaxPolicyCancellationsForGood) {
        return AccountEvaluation.GOOD;
    }
    else if (numberDelinquenciesInTimePeriod <= MaxDelinquenciesForAcceptable &&
              numberPaymentReversalsInQueryPeriod <= MaxPaymentReversalsForAcceptable &&
              numberPolicyCancellationsInTimePeriod <= MaxPolicyCancellationsForAcceptable) {
        return AccountEvaluation.ACCEPTABLE;
    }
    else if (numberDelinquenciesInTimePeriod <= MaxDelinquenciesForMarginal &&
              numberPaymentReversalsInQueryPeriod <= MaxPaymentReversalsForMarginal &&
              numberPolicyCancellationsInTimePeriod <= MaxPolicyCancellationsForMarginal) {
        return AccountEvaluation.MARGINAL;
    }
    else {
        return AccountEvaluation.POOR;
    }
}

public int getMetricsQueryTimePeriod() {
    return NumberOfDaysForSearch;
}

public Boolean countUniqueDelinquenciesAndCancellationsOnly() {
    return true;
}
}
```

This example can be modified or customized as needed to meet business requirements.

If you wish to different rating values other than the built-in ones (Excellent, Good, and so on), you can extend them in the `AccountEvaluation` typelist defined in the `t1_bc_account.xml` file. Its reference implementation includes these values:

```
<typelist name="AccountEvaluation" final="true" desc="Evaluation rating for an account">
    <typecode code="newaccount" name="New Account" desc="New Account"/>
    <typecode code="excellent" name="Excellent" desc="Excellent"/>
    <typecode code="good" name="Good" desc="Good"/>
    <typecode code="acceptable" name="Acceptable" desc="Acceptable"/>
    <typecode code="marginal" name="Marginal" desc="Marginal"/>
    <typecode code="poor" name="Poor" desc="Poor"/>
</typelist>
```

Incentive Calculation Plugin

You can customize how BillingCenter evaluates commission incentives by implementing the `IIncentiveCalculator` plugin. You can fully define how incentives work and how to cover cases that normal commission processing cannot handle.

Policy-based incentives use many different parameters. They are designed to be for a single policy only. Some value on the policy itself might be an input to offer some sort of tiered commission rates, such as higher rates for larger policies.

The incentive definition in the `PolicyBasedIncentive` entity is an example only and does not contain significant information in the reference implementation. Add any necessary information by extending the data model as appropriate. The `IIncentiveCalculator` plugin can access your data model extension properties. For more information, see the “Working with the Data Dictionary” on page 141 in the *Configuration Guide*.

The `calculatePolicyBasedIncentiveBonus` interface method performs the incentive calculations. If you create a BillingCenter incentive plan and a producer uses the incentive, the policy-based incentive batch process calls this method to calculate incentive values for the producer.

BillingCenter calculates the bonus earned by the given producer for the given policy-based incentive by calling the `calculatePolicyBasedIncentiveBonus` interface method. Your producer is specified by a `PolicyCommission` entity, and the incentive information is provided in a `PolicyBasedIncentive` entity. The method returns the result as a `BigDecimal` number that represents the dollar amount of any bonus.

The following Gosu example implements an incentive calculator plugin:

```
class MyIncentiveCalculatorTestGosuPlugin {
    private var _logger : LoggerCategory;
    function MyIncentiveCalculatorTestGosuPlugin() {
        _logger = LoggerCategory.PLUGIN;
        _logger.info("**** MyIncentiveCalculatorTestGosuPlugin. ****");
    }

    public function calculatePolicyBasedIncentiveBonus(incentive : PolicyBasedIncentive,
        comm : PolicyCommission) : BigDecimal {
        var bonus : BigDecimal;
        if (incentive typeis PremiumIncentive) {
            bonus = (incentive as PremiumIncentive).calculateBonus(comm);
        } else {
            throw "Unhandled incentive type: " + (typeof incentive);
        }
        return bonus;
    }
}
```

Commission Plugins

There are two BillingCenter commission plugins:

- “Commission Plugin” on page 162
- “Commission Override Plugin” on page 165

Commission Plugin

The Commission plugin enables configuration of the selected commission rate and subplan. A subplan defines a set of restrictions and commission parameters.

Determine Whether a Charge Earns a Commission

The `isCommissionable` method determines whether a specified charge earns a commission.

```
function isCommissionable( commissionSubPlan : CommissionSubPlan,
    charge : Charge, role : PolicyRole ) : boolean
```

The `commissionSubPlan` argument specifies the commission subplan to consult when determining whether to grant a commission. If the argument is `null`, the subplan associated with the policy is consulted.

The `charge` argument specifies the relevant charge. The `role` argument specifies the producer code role for the policy, such as Primary or Secondary.

If the specified charge earns a commission, the method returns `true`.

Getting Potential Commission Subplans

The plugin method `selectSubPlan` returns an array of potential commission subplan entities based on a policy period and a commission plan. The method signature is:

```
selectSubPlan(policyPeriod : PolicyPeriod, commissionPlan : CommissionPlan) : CommissionSubPlan[]
```

BillingCenter chooses the applicable subplan according to the following rules:

- If the method returns a single subplan from the array, the returned subplan is used.
- If the method returns more than one subplan from the array, BillingCenter chooses the first returned subplan that applies to the policy period.
- If a subplan is not selected after applying the above rules, an `InvalidCommissionPlanStateException` is thrown.

The base configuration implementation of this method returns no items as an empty array.

Getting the Commission Rate for a Charge

The `getCommissionRate` method returns the commission rate for a specified policy charge.

```
getCommissionRate( policyCommission : PolicyCommission, charge : Charge ) : BigDecimal
```

BillingCenter calls the method in the following contexts:

- To calculate the commission reserve for a new charge
- To retrieve the commission rate applied to an existing charge

The base configuration implementation of the method checks for any charge or policy override of the commission rate. If an override exists, the overriding commission rate is returned. If there are no overrides, the method returns the commission rate specified in the policy's commission subplan.

Distributing Commission Write-offs

The charge commission entity (`ChargeCommission`) has several method signatures for the method called `writeoffCommission` to write off some or all of the commission. If any code calls this method, BillingCenter must distribute the write-off amount to individual item commissions (`ItemCommission` entities) associated with that charge commission.

To determine how to distribute these items, BillingCenter calls the `distributeCommissionWriteoffAcrossItemCommissions` plugin method.

The method takes two arguments:

- Item commissions, as the type `List<ItemCommission>`
- The write-off amount, as the type `MonetaryAmount`

The method returns a map that maps an item commissions to the write-off amount for that item. Using generics notation, the return type is `Map<ItemCommission, MonetaryAmount>`.

The base configuration method sorts the item commissions by event date and adds as much write-off as possible for each commission amount that has not yet been written off. After the `distributeCommissionWriteoffAcrossItemCommissions` method completes and returns the mapping of item commissions to the write-off amount, BillingCenter validates the mapping. It checks the following:

- Verifies the sum of the allocations are correct.

- Runs validation on each of the `ItemCommission` entities.

Commission Rate Overrides During Producer Transfer

If BillingCenter is transferring a producer, BillingCenter calls the plugin method `getCommissionRateOverrideDuringProducerTransferForNewPolicyCommission`. Your method must determine the value of the commission rate override for the newly-created policy producer code.

The method's only argument is the original policy producer code before the transfer.

Your plugin method must return the desired commission rate override percentage as a `BigDecimal` value. This becomes the `CommissionRateOverridePercentage` property in the newly-created Policy Producer Code.

To return no commission rate override, return the value `null`.

The default implementation of this plugin returns the `CmsnPlanOverridePercentage` property in the original policy producer code.

Custom Item Commission Allocations

The commission payable batch process may encounter a policy commission whose earning criteria is "Custom". If so, BillingCenter calls the commission plugin's `getCustomItemCommissionAllocations` method to get the commission's allocations.

Its only parameter is a set of item commission entities that are available on the policy commission currently being made payable. The type of the parameter is `Set<ItemCommission>` using Gosu generics syntax.

It must return a map of amounts to be made payable for the corresponding invoice item. You must map each invoice item to a `BigDecimal` value representing the allocation for that invoice item. BillingCenter will make earning transactions on each invoice item for the amount that you specify. The return type using Gosu generics syntax is `Map<InvoiceItem, BigDecimal>`.

In the default plugin implementation, the behavior for the custom setting merely allocates all unpaid commission.

Determine Whether to Write Off Commission When Removing Producer Codes

When BillingCenter transfers an item from a policy commission (`PolicyCommission`) to a `null` producer code, BillingCenter calls the commission plugin `shouldWriteoffCommissionWhenProducerCodeRemoved` method. In other words, when removing producer codes, BillingCenter needs to know how to handle the commission for that item. The return value of this method determines how BillingCenter handles it:

- To write off the remaining reserve for any item commissions (not just override it), return `true`.
- To override the commission rate on the old `PolicyCommission` to zero percent, return `false`. This takes back any payable or paid commission as well as all reserve.

Its one parameter is the item commission (`ItemCommission`) with the removed producer code.

In the default plugin implementation, this method always returns `true`.

Determine Whether to Write Off Commission When Writing Off a Charge

When BillingCenter writes off a charge, the commission associated with the charge can also be written off. The `shouldWriteOffActiveCommissionForChargeWrittenOff` plugin method is called to instruct BillingCenter how to handle the commission.

To write off the commission associated with a written off charge, return `true`. To leave the commission unchanged, return `false`. The base configuration method always returns `false`.

Commission Override Plugin

BillingCenter calls the commission override plugin (`ICommissionOverride`) when the user overrides the commission rate in agency bill or direct bill. This plugin determines whether to change the commission amount of an invoice item or leave it untouched and create a commission adjustment invoice item.

There is one method you must implement, called `shouldCreateCommissionAdjustmentInvoiceItem`. It takes an invoice item, an original commission rate, and the new commission rate. The method takes commission rates as `BigDecimal` values.

If your method returns `true`, BillingCenter leaves the invoice item unchanged and BillingCenter creates an adjustment invoice item. If you return `false`, BillingCenter modifies the invoice item and potential change the invoice paid status.

BillingCenter includes a default implementation of this plugin that returns true if the item is unplanned or is a fully paid planned item.

Write your own implementation of this plugin to customize this logic.

Delinquency Processing Customization Plugin

You can customize BillingCenter delinquency processing by implementing the delinquency processing plugin `IDelinquencyProcessExtensions` plugin. This plugin performs two tasks:

- Exiting delinquency if charges are paid
- Pushing forward held events

BillingCenter includes a built-in implementation that represents the base configuration behavior. In Studio refer to the Gosu class:

```
gw.plugin.delinquency.impl.DelinquencyProcessExtensions
```

This plugin implementation calls similarly-named methods on the appropriate objects. For example, the `onCompliance` plugin method calls `collateral.onCompliance()`. You can add code before or after this code, or modify the code in other ways.

If you want to do something before or after the built-in behavior, base your plugin on the pattern of that Gosu class. To customize the behavior further, refer to the following subsections for details of each process.

Exiting Delinquency If Appropriate Charges Are Paid

BillingCenter calls the `IDelinquencyProcessExtensions` plugin method `onChargesPaid` to determine whether to end delinquency based on custom criteria.

If you do not register an implementation of the delinquency process extensions plugin, BillingCenter calls `DelinquencyProcess.onChargesPaid(chargesPaid)`. The default implementation triggers the delinquency process workflow to stop if the delinquent amount is below the threshold for exiting delinquency.

BillingCenter default behavior in the `DelinquencyProcess` method `onChargesPaid` is implemented by an enhancement method. You can edit this enhancement in Studio. Refer to the Gosu enhancement `libraries.DelinquencyProcessExt`. The reference implementation of this method gets the target due amount. If the current delinquency amount is below `DelinquencyPlan.ExitDelinquencyThreshold` and the delinquency reason is `TC_PASTDUE`, the method triggers the process to stop.

However, you might want to define the criteria differently. For example, you can make the amount due threshold to be five percent of the total policy premium due, rather than a fixed amount of money. This plugin lets you dynamically evaluate whether to end delinquency.

To support custom behavior if charges are paid, the plugin `onChargesPaid` method takes:

- a `DelinquencyProcess` entity instance
- a list of `Charge` entity instances (`List<Charge>`)

Your implementation of this method must stop the delinquency process in the right circumstances or the delinquency will never stop through other means.

The method returns nothing.

See also

- For additional information about the `DelinquencyProcess` entity, see “Rule Set Categories” on page 33 in the *Rules Guide*.

Pushing Forward Held Events

Pushing held events forward advances the date of delinquency process events if a delinquency process is held. In the reference implementation, BillingCenter uses the following rule. If no plugin is defined, BillingCenter pushes forward all uncompleted events that have a target date defined in terms of number of days after inception.

In the reference implementation, this means that the value for the number of days after inception is advanced by the number of days for which the hold was applied. To customize this behavior, this plugin’s `pushForwardHeldEvents` method takes a `DelinquencyProcess` entity and a date. The date represents the calendar data that the event was held. The plugin method can do whatever it wants in this case. It returns nothing.

As described earlier for the `onChargesPaid` entity method, the default behavior for pushing forward held events uses an entity extension method that you can edit.

The default method looks similar to the following:

```
/***
 * Example method for pushing forward events if a delinquency process is held.
 * This implementation advances all events by the number of days for which the delinquency process
 * was on hold. It does not advance events set with an exact target date (instead of days after
 * inception).
 */
function pushForwardHeldEvents(heldSince : java.util.Date) {
    var today = DateUtil.currentDate();

    var numDaysHeld = DateUtil.daysBetween(heldSince, today);

    for (event in this.OrderedEvents as DelinquencyProcessEvent[]) {
        if (not event.Completed and event.ExactTargetDate == null) {
            event.DaysAfterInception = event.DaysAfterInception + numDaysHeld;
        }
    }
}
```

If you want to do something different entirely from this approach, your plugin method must do whatever is appropriate instead of calling `delinquencyProcess.pushForwardHeldEvents(heldSince)`.

Numbers and Sequences Plugin

You can customize BillingCenter number generation by implementing the `INumberGenerator` plugin interface.

In many cases, you can implement the methods of the `INumberGenerator` interface with the sequence generator API. This API is available in both Gosu and Java with slightly different syntax. In other cases, more complex calculations may be necessary, or you may need to call an external legacy system to calculate one or more of the generated numbers.

The default implementation of the `INumberGenerator` plugin is for demonstration purposes only. Carefully consider how you want to generate your numbers and implement your own version of the `INumberGenerator` plugin interface.

Your plugin must implement the following methods:

- `generateAccountNumber` – Generate an account number string for an `Account`
- `generateCheckNumber` – Generates a check number string for a `Check`
- `generateDisbursementNumber` – Generates a disbursement number for a `Refund`
- `generateInvoiceNumber` – Generate an invoice number for an `Invoice`
- `generateStatementNumber` – Generates a statement number for a `ProducerStatement`
- `generateTransactionNumber` – Generates a transaction number from a `Transaction`
- `generateTroubleTicketNumber` – Generates a trouble ticket number from a `TroubleTicket`

BillingCenter includes a feature called *field validators*. Field validators assure that user input of a certain type of number exactly matches the correct format. The following example shows an account number field validator.

```
<ValidatorDef name="AccountNumber" value="[0-9]{3}-[0-9]{5}"  
    description="Validator.AccountNumber"  
    input-mask="###-#####"/>
```

Whenever you implement new number generator code for the first time, remember to configure the corresponding number field validator to match your new number format.

WARNING After first implementing a number generator or changing number formats, update field validators. If you do not update field validators, data entry of numbers may fail.

See also

- “Creating Unique Numbers in a Sequence” on page 152
- “Configure Account Numbers for Multicurrency Accounts in BillingCenter” on page 406
- “Field Validation” on page 239 in the *Configuration Guide*

BillingCenter Parameter Calculation Plugin

You can customize BillingCenter system parameters by implementing the `ISystemParameters` plugin.

The plugin must implement the following methods:

- `getProducerAutoPaymentThreshold` – Return the threshold for making automatic producer payments or return `null` to not suppress low producer payments. If returning non-`null`, the return value represents the minimum automatic producer payment to make. If an automatic producer payment would be made for less than this amount, the payment is canceled until the next scheduled payment date.
- `getPejorativePaymentReversalReasons` – Gets the set of payment reversal reasons that are *pejorative*. Pejorative is an insurance term for situations that result in a fee and is considered a negative for account evaluation. This method must return an array of typekeys, which in the reference implementation includes only a returned check (typekey value `RETURNEDCHECK`). You can extend this list for other values by extending the `PaymentReversalReason` typelist.

Billing Instruction Execution Customization Plugin

Implement the `IBillingInstruction` plugin interface to add charges to a billing instruction. Refer to the Gosu code of the default implementation `gw.plugin.billinginstruction.impl.BillingInstruction` for an example implementation.

Add Additional Charges

The method `addAdditionalCharges` takes a billing instruction as a parameter. Your implementation of this method lets you create additional charges on the billing instruction. In the default implementation of `IBillingInstruction` plugin, this method does nothing. However, you could add additional charges. You probably want to check the billing instruction type before adding charges, with code such as:

```
if (billingInstruction typeis AccountGeneral)
```

BillingCenter treats any new charges identically to charges originally on the billing instruction. New charges have an effective date identical to the effective date of the billing instruction. BillingCenter automatically adds charge transactions and invoice items as appropriate.

For example, the following is a Gosu implementation of the plugin that adds one charge:

```
public override function addAdditionalCharges(billingInstruction : BillingInstruction) {  
  
    if (billingInstruction typeis AccountGeneral) {  
        var lateFee = new Charge()  
        lateFee.ChargePattern = ChargePatternKey.ACOUNTLATEFEE;  
        lateFee.Amount = 100  
        billingInstruction.addToCharges(lateFee)  
  
    } else if (billingInstruction typeis Issuance) {  
        var premium = new Charge()  
        premium.ChargePattern = ChargePatternKey.PREMIUM;  
        premium.Amount = 100  
        billingInstruction.addToCharges(premium)  
    }  
}
```

Policy Period Information Customization Plugin

IMPORTANT For detailed information about the integration of PolicyCenter and BillingCenter, see “Billing Integration” in the *PolicyCenter Integration Guide*.

General Hooks for Customizing Policy Period Info

An external system can request a summary of a policy period using the `BCAPI.getPolicyPeriodInfo(...)` SOAP API. For more information about this API, see “Get a Billing Object Summary” on page 118. That API returns an account information summary of type `PolicyPeriodInfo`. Before the policy period summary is sent to the external system, you can set properties on it or trigger custom logic. To add such customization, implement the `IPolicyPeriod` plugin.

For example, use this to populate data model extension properties that you add to the `PolicyPeriodInfo` non-persistent entity. Suppose you want a separate web portal for customers to be able to query BillingCenter for a policy period status. Suppose you want to display whether the policy period’s account is past due for over 90 days. Instead of calculating account past due dates in the external integration code, add a data model extension Boolean field that means past due for over 90 days. Use this plugin to populate the data model extension property.

The `IPolicyPeriod` plugin has one main method called `updatePolicyPeriodInfo`. You can do whatever is appropriate with the one argument, which is an `PolicyPeriodInfo` entity instance. There is no return result from the method, as there are no specific requirements or tests required by this method. BillingCenter calls this method before `getPolicyPeriodInfo` returns the value to a SOAP API client calling `BCAPI.getPolicyPeriodInfo(...)`. Add whatever application logic you deem appropriate.

Applying a Full Pay Discount

The plugin has another method you must implement, called `applyFullPayDiscount`. This method provides a hook for you to implement the full pay discount on a policy period. BillingCenter calls this method before `getPolicyPeriodInfo` returns the value to a SOAP API client. Add whatever application logic you deem appropriate.

Cancellation Handling

During cancellation, BillingCenter calls the `IPolicyPeriod` plugin to let you customize how BillingCenter handles invoice items.

BillingCenter calls the `getCustomPayersForCancellationCollapsedInvoiceItems` method during cancellation. This method must return an `InvoiceItemPayers` object. This object specifies a mapping between `InvoiceItem` and `InvoicePayer` to use when collapsing invoice items during policy cancellation. This allows customization of which payer an invoice item is assigned to. For example, when invoice items collapse during policy cancellation, use this class. It allows you to specify whether to assign those invoice items to the account that owns the policy period, or to some other payer account.

The design and usage of an `InvoiceItemPayers` object uses a builder-style syntax to add (or replace) mappings. First call the `assign` method and pass a list of invoice items (the type `List<InvoiceItem>`). Next, on the result of that method, call the `to` method and pass an invoice payer object (the type `InvoicePayer`).

Your implementation of the `getCustomPayersForCancellationCollapsedInvoiceItems` method gets one parameter that contains the default mapping between `InvoiceItem` and `InvoicePayer`. To add additional mappings, use the syntax:

```
myInvoiceItemPayers.assign({myInvoiceItem}).to(myInvoicePayer)
```

Note: The usage syntax of an `InvoiceItemPayers` object is similar to another class called `InvoiceItemPlacements`. The `InvoiceItemPlacements` class encapsulates mapping of invoice items to invoices. It uses the builder style syntax: `placements.place(theItem).on(myInvoice)`. For more about that object, see “Date Sequence Plugin” on page 183.

Instances of this class contain the following methods:

- `getInvoiceItemsFor(InvoicePayer payer)` – Get a list of invoice items for a specific invoice payer.
- `payerFor(InvoiceItem invoiceItem)` – Get the payer for a specific invoice item.
- `getInvoiceItems()` – Return the full set (`java.util.Set`) of all invoice items in this object. The return type is `Set<InvoiceItem>`.
- `getPayers()` – Return the full set (`java.util.Set`) of all invoice items in this object. The return type is `Set<InvoicePayer>`

You can optionally choose to create a new instance of `InvoiceItemPayers` and return it instead of modifying the passed-in version and returning it.

During cancellation, BillingCenter provides a hook to update the set of future invoice items to collapse for policy cancellation. BillingCenter calls this method just prior to placing the collapsed invoice items. Its method signature is:

```
void updateCancellationCollapsedInvoiceItemsBeforePlacement(List<InvoiceItem> invoiceItems,  
Date cancellationCollapseDate)
```

The first parameter is a list of invoice items from the future that BillingCenter is collapsing due to a policy cancellation. After returning from this method, BillingCenter immediately calls the `InvoiceAssembler` plugin to place these items.

The second parameter is the cancellation collapse date. This is the later (further in future) of the following two dates: the cancellation effective date or the current date

The primary use of this method is to set the event date (`EventDate`) of each invoice item to an appropriate value based on the cancellation effective date. The `InvoiceAssembler` plugin typically would use this event date to place the invoice item as appropriate.

Billing Method Changes

This plugin interface also provides a hook to tell BillingCenter which items to move if the billing method changes on a policy. This plugin method, called `getItemsAffectedByBillingMethodChange`, takes a single parameter which is the `PolicyPeriod` that is changing billing method.

If `policy.isAgencyBill()` returns `true`, the policy is in the process of changing from agency bill to direct bill. If `policy.isAgencyBill()` returns `false`, the policy is in the process of changing from direct bill to agency bill.

This method must return a list of invoice items whose billing method must change as a result of the policy period changing billing methods. The return type is `List<InvoiceItem>`.

BillingCenter Application Event Customization Plugin

You can customize BillingCenter processing of certain types of objects by implementing the `IEventHandler` plugin. BillingCenter calls the methods of this plugin whenever important events occur in the life cycle of BillingCenter entities.

In the context of the `IEventHandler` plugin, the term *event* means *an opportunity to trigger customer code*. This plugin has nothing to do with the BillingCenter messaging event system, the Event Fired rule set, or messaging destinations. Similarly, this plugin has nothing to do with history events.

Most of the `IEventHandler` plugin methods return a Boolean value, which determines an important setting or behavior. You can calculate the return values dynamically and even call out to external systems if necessary. For performance reasons however, call out to external systems only if strictly necessary.

For any of the `IEventHandler` plugin methods, if the answer is always the same and does not rely on any calculation, you must still include this method. Write the method as a single line of code that always returns `true` or always return `false`.

Determining Whether to Start the Delinquency Process

BillingCenter calls the `canStartDelinquencyProcess` method to start delinquency. It takes a policy period object and returns true or false. If BillingCenter calls this method, the policy (technically a `PolicyPeriod` entity) in the method parameters is already past due but an active delinquency process has not yet started. This method confirms whether to start the delinquency process for a past-due policy.

Separately, BillingCenter calls the `beforeStartDelinquencyProcessing` method before starting a delinquency process for a past-due policy. The method takes an array of policy periods as an argument. The policy period is past due but does not have an active delinquency process yet.

Controlling Whether to Close Policies

BillingCenter calls the `beforePolicyPeriodClosed` method immediately before a policy period closes. This method returns `true` to close the policy period. Otherwise, it returns `false` or `null` to prevent the policy period from closing.

The method signature looks like:

```
Boolean beforePolicyPeriodClosed(PolicyPeriod policyPeriod)
```

Custom Processing During Policy Transfer to New Producers

BillingCenter calls the `policyPeriodTransferredToNewProducer` method immediately before transferring a policy period to a new producer. There is no return value. The method takes a policy period and two producers, specified by producer code.

The method signature looks like:

```
void policyPeriodTransferredToNewProducer(PolicyPeriod policyPeriod,  
                                         ProducerCode oldProducerCode, ProducerCode newProducerCode)
```

Validating Credit Card Numbers

BillingCenter calls the `validateCreditCardNumber` method whenever credit card information changes on an Account or a Policy. The method returns `true` if the information is valid. Otherwise, it returns `false`.

The method signature looks like:

```
Boolean validateCreditCardNumber(CreditCardIssuer issuer, String number, String cvv, Date expiration)
```

Validating Manual Refunds

BillingCenter calls the `validateManualRefundAmount` method whenever someone attempts a manual refund. The method must return `true` if the amount is less than or equal to the allowable refund amount. Otherwise, it returns `false`.

The method signature looks like:

```
Boolean validateManualRefundAmount(BigDecimal amount, BigDecimal suspended,  
                                 BigDecimal pendingRefund, Refund refund)
```

The default behavior is to return `true` if `amount` is less than or equal to the value:

```
(suspended - pendingRefund)
```

If you want a similar behavior as the default behavior, this method would look like the following, shown as a Gosu plugin method:

```
public function validateManualRefundAmount(amount : BigDecimal, suspended : BigDecimal,  
                                         pendingRefund : BigDecimal, refund : Refund) : Boolean {  
    return amount <= (suspended - pendingRefund)  
}
```

Determining Whether To Place Holds On Accounts

If an approver rejects a disbursement, BillingCenter calls the `rejectDisbursementBehavior` method to determine how to handle the disbursement on the `RefundDetail` screen. The method returns `true` to allow the approver to put a hold on the account by showing a button that places a hold on the policy. If method returns `false`, BillingCenter hides the button on that screen.

The return value affects the user interface code only. The return result does not define whether to put a policy on hold automatically. If customers want a hold to happen automatically, customers could implement this method and place a hold manually on the policy.

The method signature looks like:

```
Boolean rejectDisbursementBehavior(Refund refundToReject)
```

Determining Whether to Pay Disbursements

If an approver rejects a disbursement, BillingCenter calls the `payDisbursementUponDueDate` method to determine whether to pay it. In your implementation of this method, return `true` to pay the disbursement if the date reaches the due date regardless of the disbursement's approval status. Otherwise, return `false`.

The method signature looks like:

```
Boolean payDisbursementUponDueDate(Refund refundToPay)
```

ChargeInitializer Plugin

The ChargeInitializer plugin configures the final step of the charge invoicing process for new charges. It enables modification of invoice item settings, such as an item's monetary amount and billing date, and also supports assigning invoice items to particular invoices.

Note: The ChargeInitializer plugin configures brand new charges. To modify an executed charge, use the ChargeInstallmentChanger class. For further information, see “Modifying an Existing Charge” on page 427 in the *Configuration Guide*.

Using the ChargeInitializer Plugin

When processing a new charge, the creation of invoice items and the placement of the items on invoices is configurable by implementing the ChargeInitializer plugin. The default implementation is located in the Gosu class `gw.plugin.charge.ChargeInitializer`. The class's entry point for processing a new charge is the `customizeChargeInitializer` method.

This section describes the `customizeChargeInitializer` method and provides code samples that demonstrate some common operations used to configure the plugin.

The `customizeChargeInitializer` Method

The `customizeChargeInitializer` method of the ChargeInitializer plugin is called once for each planned charge in a billing instruction. It is the only method you must implement to customize charges and invoice items before their associated billing instruction is executed.

The method accepts a single parameter of the type `gw.api.domain.charge.ChargeInitializer`. The parameter contains a List of planned invoice items in the form of Entry objects. These Entry objects can be modified in the method's implementation.

Note: Do not confuse the `gw.plugin.charge.ChargeInitializer` class used to implement the ChargeInitializer plugin with the `gw.api.domain.charge.ChargeInitializer` class passed as a parameter to the `customizeChargeInitializer` method. The following table differentiates the two classes.

| | |
|---|---|
| <code>gw.plugin.charge.ChargeInitializer</code> | Implements the ChargeInitializer plugin. This class provides access to an instance of the <code>gw.api.domain.charge.ChargeInitializer</code> class. |
| <code>gw.api.domain.charge.ChargeInitializer</code> | Parameter to the <code>customizeChargeInitializer</code> method. An instance of this class contains a List of planned invoice items in the form of Entry objects. |

Create a New Invoice Item

Use the `ChargeInitializer.addEntry` method to create a new invoice item Entry object and add the object to the collection of invoice items to create for the charge. When the ChargeInitializer is subsequently executed, each Entry in the collection becomes an invoice item on an invoice in the appropriate invoice stream.

```
// Objective: If the down payment is 20% or more and the payment plan is on a monthly invoice stream,
// bill half the down payment on the first invoice and half on a later invoice.
var isDirectBillIssuance = !initializer.AgencyBill and initializer.BillingInstruction.typeis Issuance
var paymentPlan = initializer.PolicyPeriod.PaymentPlan
if ( paymentPlan.DownPaymentPercent > 20 and paymentPlan.Periodicity == TC_MONTHLY ) {
    // Find down payment entry and amount
    var firstDownPaymentItem =
        initializer.Entries.firstWhere( \ entry -> entry.InvoiceItemType == InvoiceItemType.TC_DEPOSIT )
    var totalDownPaymentAmount = firstDownPaymentItem.Amount

    // Divide down payment in half
    firstDownPaymentItem.Amount = totalDownPaymentAmount / 2.0
    var secondDownPaymentAmount = totalDownPaymentAmount - firstDownPaymentItem.Amount

    // Look for the next invoice
}
```

```

var nextInvoiceEntry =
    initializer.Entries.firstWhere( \ entry -> entry.Invoice != firstDownPaymentItem.Invoice )

// Add second down payment item entry for other half of down payment
var secondHalfDownPaymentEntry =
    initializer.addEntry(secondDownPaymentAmount,
        InvoiceItemType.TC_DEPOSIT,
        firstDownPaymentItem.EventDate)

// Move the second half of the down payment to the second invoice
secondHalfDownPaymentEntry.setInvoice(nextInvoiceEntry.Invoice)
}

```

Bill an Item Immediately

Use the `ChargeInitializer.Entry.billToday` method to bill an Entry immediately. When BillingCenter creates invoice items for the ChargeInitializer, the entry will be placed on the invoice with today's date.

```

// Bill back dated direct bill down payments immediately (today)
var isBackDatedPolicy = DateUtil.compareIgnoreTime(initializer.BillingInstruction.EffectiveDate,
    DateUtil.currentDate()) < 0

var isDirectBillIssuance = !initializer.AgencyBill and initializer.BillingInstruction.typeis Issuance
if (isBackDatedPolicy and isDirectBillIssuance) {
    for (entry in initializer.Entries) {
        var isDeposit = entry.InvoiceItemType == InvoiceItemType.TC_DEPOSIT
        if (isDeposit) {
            entry.billToday()
        }
    }
}

```

Place an Item on an Invoice

To place an Entry on an invoice, use the `ChargeInitializer.Entry.setInvoice` method.

The following code sample looks for Entry objects that are one-time charges. Whenever one is found and its event date is in the future, the code creates a new invoice and places the Entry on it.

```

/** Place an invoice item on an invoice */
var today = DateUtil.currentDate()

/* Iterate through all Entry objects */
for (entry in initializer.Entries) {
    /* Is this Entry a one-time charge? Is its event date in the future? */
    var isOneTime = (entry.InvoiceItemType == InvoiceItemType.TC_ONETIME)
    if (isOneTime and entry.EventDate.after(today)) {
        /* Yes, create a new invoice to hold the Entry */
        var newInvoice = initializer.InvoiceStream.createAndAddInvoice(entry.EventDate)

        /* Place the Entry on the new invoice */
        entry.setInvoice(newInvoice)
    }
}

```

Customize a Charge Based on the Billing Instruction Type

To customize charge invoicing based on the type of billing instruction received, perform the steps depicted in the following code sample.

```

/* Type of billing instruction that you want to perform custom invoicing on.
 * Sample code uses type PolicyChange, but other types can also be used.
 */
PolicyChange billingInstruction      // Pre-initialized to appropriate value

/* Prepare for customizing the items.
 *   1. Slices the charge into Entry objects. Entry objects are subsequently converted to invoice
 *      items when the billing instruction is executed. These Entry objects can be added to,
 *      modified, or removed.
 *   2. Calls the Charge Initializer plugin's customizeChargeInitializer() method.
 */
billingInstruction.prepareForManualModification()

/* Retrieve the charge initializer */
ChargeInitializer initializer = billingInstruction.getChargeInitializers().get( 0 )

/* Retrieve all the initializer's Entry objects */

```

```

List<? extends ChargeInitializer.Entry>entries = initializer.getEntries()

/* Add a new Entry to the initializer */
Entry newEntry = initializer.addEntry( amount, invoiceItemType, eventDate )

/* Getter and setter functions exist for the amount, invoiceItemType, invoice, eventDate, and
 * any custom configuration fields. Other functions specify whether the invoiceItem will be placed
 * on an invoice with today's date.
*/
/* Modify an existing Entry */
entries[ 0 ].setAmount( $100 )

/* Remove an Entry */
entries[ 1 ].remove()

/* Properties on the future Charge object can be set on and retrieved from the ChargeInitializer
 * object. The property values will be carried over to the Charge when the Billing Instruction is
 * executed.
*/
initializer.setExtensionProperty( Charge#ChargeGroup, "One-time" )
String chargeGroup = initializer.getExtensionProperty( Charge#ChargeGroup )

/* Properties on the future InvoiceItem object can be set on and retrieved from the Entry object.
 * The property values will be carried over to the InvoiceItem when the Entry is converted.
*/
entries[ 0 ].setExtensionProperty( InvoiceItem#Comment, "Special fee" )
String invoiceItemComment = entries[ 0 ].getExtensionProperty( InvoiceItem#Comment )

/* When customization is finished, the billing instruction is executed. This converts the Entry
 * objects to InvoiceItem objects. InvoiceItem properties set on the Entry object (such as the
 * Comment property set above) are carried over to the InvoiceItem. Also, the ChargeGroup property
 * set on the ChargeInitializer.Charge object above is carried over to the created Charge.
*/
billingInstruction.executeCustomizedCharges()

```

Payment Plan Plugin

The Payment Plan plugin enables customization of the following items:

- Installment event dates for a new charge
- Installment event dates for an existing charge that were altered because of a change to a payment plan
- Invoice items that were replaced because of a change to a payment plan

BillingCenter provides a base configuration implementation of the Payment Plan plugin in the Gosu class `gw.plugin.invoice.impl.PaymentPlan`. The class implements the `IPaymentPlan` interface. The base configuration's implementation of the interface methods can be modified to customize the plugin's operations.

Method: `createFullSetOfInstallmentEventDates`

The `createFullSetOfInstallmentEventDates` method is called during the processing of a new charge. The method enables the customization of the event dates for the charge.

Parameters and Return Value

The `createFullSetOfInstallmentEventDates` method receives a `ChargeInitializer` object and a list of installment event dates generated for the charge. The plugin method can modify the event dates to achieve a desired customization.

Properties in the `ChargeInitializer` object can be retrieved, but they cannot be set. Also, at the time the plugin method is invoked, the object is still progressing along its path of creation. Accordingly, all of its properties have not yet been initialized. The object essentially contains the information received from the originating billing instruction but not the complete set of properties generated by BillingCenter.

The method returns the potentially customized list of event dates. In the base configuration, the method returns the list of event dates unmodified.

When the Method is Called

There are two operations that lead to the invocation of the method.

Preparing a Charge for Customization

Before performing customizations on a new charge, the charge must be prepared. A charge is prepared by calling the `prepareForManualModification` method of the `BillingInstruction` class. During charge preparation, the plugin method `createFullSetOfInstallmentEventDates` is invoked. Charge preparation occurs at an early stage of the charge processing operation.

Sample code to demonstrate charge processing when customizing a charge is shown below.

```
billingInstruction = ... // Previously assigned billing instruction object of appropriate type  
  
// New charges are handled by a ChargeInitializer object  
var chargeInitializer = billingInstruction.buildCharge( amount, chargePattern )  
  
// Prepare the charge for modification.  
// The Payment Plan plugin method createFullSetOfInstallmentEventDates() is invoked during the  
// processing of this method.  
billingInstruction.prepareForManualModification()  
  
// ... Customization of event dates occurs here ...  
  
// Execute the customized charge.  
billingInstruction.executeCustomizedCharges()
```

Executing a Standard Charge

Charges that follow the standard charge invoicing process can customize the charge's event dates when the charge is executed. A charge is executed at the end of the charge processing operation.

Sample code to demonstrate standard charge processing is shown below.

```
billingInstruction = ... // Previously assigned billing instruction object of appropriate type  
  
// New charges are handled by a ChargeInitializer object  
var chargeInitializer = billingInstruction.buildCharge( amount, chargePattern )  
  
// Execute the charge.  
// The Payment Plan plugin method createFullSetOfInstallmentEventDates() is invoked during the  
// processing of this method.  
billingInstruction.execute()
```

Method: `recreatePlannedInstallmentEventDatesForPaymentPlanChange`

Whenever a policy period's payment plan is changed, the `recreatePlannedInstallmentEventDatesForPaymentPlanChange` method is called for each *non-one-time* charge. The method is not invoked for a one-time charge. The method enables customization of the event dates that were generated based on the modified payment plan.

Parameters and Return Value

The `recreatePlannedInstallmentEventDatesForPaymentPlanChange` method accepts a `Charge` object for the relevant charge and a list of event dates that have been generated based on the modified payment plan. The event dates can be for *any* installment, not just planned items as the method's name implies.

The method returns the potentially customized list of event dates. In the base configuration, the method returns the list of event dates unmodified.

Method: recreateInvoiceItemsForPaymentPlanChange

Whenever a policy period's payment plan is changed, the `recreateInvoiceItemsForPaymentPlanChange` method is called for each charge. The method enables customization of invoice item settings. Invoice items are modified by setting the properties of their intermediate `Entry` objects. For more information, see “Customize a Charge Based on the Billing Instruction Type” on page 173.

Parameters and Return Value

The `recreateInvoiceItemsForPaymentPlanChange` method accepts a `ChargeInstallmentChanger` object for the relevant charge. The list of `Entry` objects for the charge is stored within that object.

The method has no return value. Configuration of the charge's invoice items is performed on the `Entry` objects stored in the `ChargeInstallmentChanger` parameter.

The plugin method in the base configuration is an empty placeholder method.

Plugin Methods Invoked Following a Payment Plan Change

When a payment plan is changed, the charge handling process invokes different plugin methods depending on the type of charge.

Charges that are *not* one-time charges call the `recreatePlannedInstallmentEventDatesForPaymentPlanChange` and `recreateInvoiceItemsForPaymentPlanChange` methods, in that order.

One-time charges invoke only the `recreateInvoiceItemsForPaymentPlanChange` method.

Invoice Plugin

Use the invoice plugin (`IInvoice`) to change the default information about fees on an invoice or to change some default behaviors of invoices. To open the plugin registry, in the Project window in Studio, navigate to `configuration` → `config` → `Plugins` → `registry`, and then open `IInvoice.gwp`. The default implementation is the Gosu class `gw.plugin.invoice.impl.Invoice`.

There are several getter methods that you can implement to override fee behaviors:

- `getCollateralOutstandingAmount`
- `getCollateralRemainingBalance`
- `getCollateralUnappliedAmount`
- `getInstallmentFeeAmount`
- `getInvoiceFeeAmount`
- `getLateFeeAmount`
- `getOutstandingAmount`
- `getPaidStatus`
- `getPaymentReversalFeeAmount`
- `getReinstatementFeeAmount`
- `getRemainingBalance`
- `getUnappliedAmount`

There is also one setter method:

- `setPolicyPeriodInvoiceSnapshot`

There are some other methods that you can implement to override default behaviors. All the following methods return Boolean values:

- `shouldHoldDirectBillInvoiceItem` – Return `true` to hold the direct bill invoice item. Otherwise, return `false`. This runs during the Invoice batch process.

- `shouldCarryForwardInvoice` – Return `true` to carry the invoice forward. Otherwise, return `false`. This runs internally every time `Invoice.sendInvoice` runs.
- `shouldCreatePaymentRequest` – Override the condition on which BillingCenter sends a payment request. The default is for BillingCenter to send a payment request when the payment method is non-responsive.
- `shouldAccountUnappliedFundsPayChargesOnPolicies` – Return `true` to use account unapplied to pay charges on invoice billed.
- `shouldMakeInvoiceItemsDueBeforePaying` – Return a list of invoice items that must be made billed and due before completing the execution of a payment.

Invoice Assembler Plugin

The placement of invoice items on invoices is determined by settings defined in various BillingCenter components, including the policy period's payment plan and the account's billing plan. After applying the settings of these components, BillingCenter generates sets of invoice items and invoices and places each invoice item on an appropriate invoice. At that point, the Invoice Assembler plugin is called, which enables customization of invoice item placement on invoices.

Method: `getCustomInvoiceItemPlacements`

The Invoice Assembler plugin implements the `IInvoiceAssembler` interface in the `gw.plugin.invoice.impl.InvoiceAssembler` class. The interface defines a single method called `getCustomInvoiceItemPlacements`, which accepts the following arguments:

- `defaultPlacements` – An `InvoiceItemPlacements` object that contains sets of invoice items and invoices. Each invoice item has been placed on an invoice based on the BillingCenter settings that determine item placement.
- `context` – Many BillingCenter operations call the Invoice Assembler plugin. The `context` parameter identifies the operation that caused the current plugin call.

The method returns an `InvoiceItemPlacements` object containing potentially modified invoice item placements. The base configuration method returns the `defaultPlacements` argument unchanged.

The `defaultPlacements` Parameter

The `defaultPlacements` parameter is a Java object of type `InvoiceItemPlacements`. It contains the following fields:

- `InvoiceItems` – A Java Set of `InvoiceItem` objects that are being placed onto invoices as a result of the BillingCenter operation that called the plugin.
- `AllInvoiceItemsToBePlaced` – Identical to `InvoiceItems`. Use of `InvoiceItems` is recommended.
- `Invoices` – A Java Set of `Invoice` objects that contain the placed invoice items.

The `defaultPlacements` parameter provides the following methods:

- `invoiceFor(InvoiceItem)` – Returns the `Invoice` object containing the specified `InvoiceItem`.
- `getInvoiceItemsFor(Invoice)` – Returns a Java Set of `InvoiceItem` objects included on the specified `Invoice`. The returned set is a subset of the objects in the `InvoiceItems` set, which includes only the invoice items being placed for the BillingCenter operation that called the plugin.
- `place(InvoiceItem)` – Returns a `Placer` object for the specified `InvoiceItem`. The `Placer` object provides the following method:
 - `on(Invoice)` – Places the `InvoiceItem` associated with the `Placer` object onto the specified `Invoice`.

The context Parameter

Many operations cause BillingCenter to call the Invoice Assembler plugin. For example, the plugin is called when processing a new or modified charge, reversing a charge, or handling a change to a payment plan. The `context` parameter identifies the operation that BillingCenter is performing when the plugin is called.

The parameter value is a typecode from the `InvoiceAssemblerContext` typelist. Each typecode is described below, along with the public class method that calls the plugin, and the operation being performed when the plugin is called.

| <code>InvoiceAssemblerContext Typecode</code> | <code>Calling Methods and the Operations Being Performed</code> |
|---|--|
| <code>TC_AGENCYPLANCHANGE</code> | Not used. |
| <code>TC_API</code> | <p><code>ChargeInstallmentChanger.execute</code>: Adding non-reversed invoice items to an existing charge. The <code>context</code> value is set to <code>TC_API</code> when the object is created by the <code>ChargeInstallmentChanger(Charge)</code> constructor. Calls the plugin once per invoice that non-reversed items were added to and passes all non-reversed invoice items for the invoice.</p> <p>If the <code>ChargeInstallmentChanger</code> object includes any reversed invoice items, the plugin is called with the <code>TC_OFFSET</code> context for each Reversal item. See the <code>TC_OFFSET</code> typecode.</p> <p><code>InvoiceItemImpl.setInvoice(Invoice)</code>: Placing Reversal invoice items on an invoice. Calls the plugin once for each Reversal invoice item. Does not call the plugin if the <code>Invoice</code> parameter is <code>null</code>. The base configuration does not call the plugin through this public method.</p> <p><code>InvoiceUtil.moveInvoiceItems</code>: Moving invoice items to another invoice. Calls the plugin once for each invoice receiving moved items.</p> |
| <code>TC_ASSIGNMENTOFITEMS</code> | <p><code>AccountImpl.becomePayerOfInvoiceItems</code>: Assigning invoice items to a new Account payer. Calls plugin once to place all items in the new invoice stream. The new invoice stream is specified as a parameter or, if <code>null</code>, defaults to the invoice stream associated with the <code>AccountImpl</code> object.</p> <p><code>ProducerImpl.becomePayerOfInvoiceItems</code>: Assigning invoice items to a new Producer payer. Calls plugin once to place all items in the new invoice stream. The new stream is specified as a parameter or, if <code>null</code> or unspecified, defaults to the invoice stream associated with the <code>ProducerImpl</code> object. Note: Two instances of this method exist with different signatures. Both instances call the plugin.</p> |
| <code>TC_BILLINGINSTRUCTION</code> | <p><code>ChargeInitializerImpl.executePreparedEntries</code>: Processing a new charge. Calls plugin once for each invoice that contains invoice items for the charge. Passes a list of all invoice items on the invoice.</p> <p><code>InvoiceImpl.addFees</code>: Adding an invoice fee for a charge. Calls plugin with the fee invoice item.</p> |
| <code>TC_CARRIEDFORWARD</code> | <p><code>AccountInvoiceImpl.doInvoiceProcessing</code>: Carrying invoice items forward to the next invoice. Calls plugin once and passes all invoice items being carried forward.</p> <p><code>InvoiceItemImpl.pushForward</code>: Moving held invoice items to the next invoice. Calls plugin once for each held invoice item.</p> |
| <code>TC_COLLAPSEFUTUREINVOICES</code> | <code>PolicyPeriodImpl.collapseFutureInvoices</code> : Collapsing future invoices due to a cancellation. Calls plugin once for each payer with collapsing invoice items. Items are placed on the invoice stream of the payer. |

| InvoiceAssemblerContext Typecode | Calling Methods and the Operations Being Performed |
|----------------------------------|--|
| TC_COMMISSIONREMAINDER | <p>ChargeCommissionImpl.writeoffCommissionOnRemainderFor: Writing off a commission remainder. Calls plugin with the single invoice item and places the item on the invoice stream of the charge's default payer. Note: The code path to the plugin includes the public method ChargeImpl.createCommissionRemainderItem.</p> |
| | <p>CommissionDistributor.distributeCommission: Creating a commission adjustment for an agency bill policy period. Calls plugin with the single invoice item and places it on the invoice stream of the charge's default payer.</p> |
| | <p>InvoiceItemImpl.overrideCommissionRate: Creating a commission adjustment for an overridden commission rate for an agency bill policy period. Calls plugin with the single invoice item.</p> |
| TC_INVOICEITEMEVENTDATECHANGE | <p>InvoiceItemImpl.setEventDateAndInvoice: Changing the event date for an existing invoice item. Calls plugin with the single invoice item and places it on the invoice stream of the default payer. Note: The base configuration does not call this public method.</p> |
| TC_INVOICINGOVERRIDABLECHANGE | <p>ChargeImpl.updateWith, ChargeInitializerImpl.updateWith, PolicyPeriodImpl.updateWith: Updating the invoice stream or payer for the object. Calls the plugin once passing the affected invoice items. When processing a policy period, if the policy period has not yet been issued, the plugin is not called. Note: The code path to the plugin includes the public method InvoicingOverriderInternal.update.</p> |
| TC_MAKEITEMDUE | <p>BaseDistExecutor.executeDistItems: Creating new due invoices. Calls plugin once for each invoice stream and passes all invoice items and the new invoice to place them on. The new invoice is subsequently made due.</p> |
| TC_OFFSET | <p>ChargeInstallmentChanger.execute: Reversing existing invoice items that are not caused by a payment plan change. Calls plugin once for each Reversal invoice item created. Note: The code path to the plugin includes the public method InvoiceItemImpl.createAndPlaceOffset.</p> |
| | <p>ChargeImpl.afterReversalCreated: Reversing an entire charge. Every invoice item for the charge is reversed, creating Reversal items. Calls plugin once per payer and includes all the Reversal invoice items for the payer.</p> |
| TC_PAYERBILLINGSETTINGSCHANGE | <p>InvoiceStreamUpdater.execute: Updating invoice streams because of a change to the payer's invoicing settings. Moving invoice items from obsolete invoices to invoices in the appropriate stream. Calls plugin once with all invoice items.</p> |
| TC_PAYMENTPLANCHANGE | <p>ChargeInstallmentChanger.placeInvoiceItems: Adding replacement invoice items caused by a change to the payment plan. The context parameter is set to TC_PAYMENTPLANCHANGE when the object is created by the ChargeInstallmentChanger(Charge, PaymentPlan, boolean) constructor. Calls plugin once for each invoice per charge that receives added invoice items. For example, if ChargeA adds invoice items to the January and February invoices, and ChargeB adds items to the same invoices, the plugin is called four times in the order:</p> <ul style="list-style-type: none"> • January invoice with ChargeA invoice items • February invoice with ChargeA items • January invoice with ChargeB items • February invoice with ChargeB items |
| | <p>InvoiceItemImpl.createAndPlaceForPaymentPlanChange: Processing Reversal invoice items caused by a payment plan change. Calls the plugin once for each Reversal item created because of the plan change.</p> |

Overriding the Default Placement of Invoice Items on Invoices

Invoice items are placed on invoices by using the `defaultPlacements` object's `place` method to specify the invoice item and the `Placer` object's `on` method to specify the invoice.

```
defaultPlacements.place(anInvoiceItem).on(anInvoice)
```

If a more complex adjustment is required, such as adding a new invoice or new invoice item, the recommended practice is to perform the adjustment in the Payment Plan plugin. For details, see “Payment Plan Plugin” on page 174. The adjustment can be performed in the Invoice Assembler plugin only if the Payment Plan plugin cannot accomplish the desired operation. If the Invoice Assembler plugin is used, the recommended practice is to create and return a new `InvoiceItemPlacements` object that contains the modified objects.

Invoice Stream Plugin

The invoice stream plugin (`IInvoiceStream`) interface lets you to customize invoice streams and customize how BillingCenter chooses the right invoice stream for the invoice items of a charge. To open the plugin registry, in the Project window in Studio, navigate to `configuration` → `config` → `Plugins` → `registry`, and then open `IInvoiceStream.gwp`. The default implementation is the Gosu class `gw.plugin.invoice.impl.InvoiceStream`.

An invoice stream is a series of related invoices that typically have the same periodicity. For example, monthly invoices might bill together separately from mailings for weekly invoices. These become different invoice streams. You specify a periodicity with a typecode from the `Periodicity` typelist, which you can extend.

This topic describes methods of the Invoice Stream plugin interface for the following functions:

- “Get Invoice Stream Periodicity For an Invoice Stream” on page 180
- “Get Existing Matching Invoice Stream” on page 181
- “Customize New Invoice Stream” on page 182
- “Get Anchor Dates for Custom Periodicities” on page 182

General Flow of Calls to the Invoice Stream Plugin

When BillingCenter needs to select an invoice stream for the invoice items of a charge, it calls the `getInvoiceStreamPeriodicityFor` plugin method to choose the periodicity of the stream. Then, BillingCenter calls the `getExistingInvoiceStreamFor` plugin method to select an invoice stream with that periodicity. If the method returns null, there is no existing invoice stream for the charge. BillingCenter creates a new `InvoiceStream` and passes it to the `customizeNewInvoiceStream` plugin method.

For invoice streams that use custom periodicities, BillingCenter calls the `getAnchorDatesForCustomPeriodicity` plugin method to get their custom anchor dates. If you have invoice streams with custom periodicities, change the `getAnchorDatesForCustomPeriodicity` method. Alternatively, you can set the anchor dates on new invoice streams in your implementation of the `customizeNewInvoiceStream` plugin method.

Get Invoice Stream Periodicity For an Invoice Stream

When BillingCenter wants to determine the correct invoice stream to use, it calls the invoice stream plugin. BillingCenter places the invoices and the invoice items on the specified invoice stream.

The default implementation of this plugin method just uses the default invoice stream selection based on the periodicity of the payment plan argument to the plugin methods. For example, if the payment plan is monthly, the default invoice stream is the monthly invoice stream.

However, you can customize this with additional logic. This supports the following types of extra behavior:

- Large commercial accounts with special requirements or exceptions. For example, perhaps some types of loss sensitive receivable must bill on the last Friday of the month. However, all other monthly billings follow regular recurrence rules based on the anchor date.
- Payroll deduction billing, which typically require custom logic to synchronize with the payers' payroll systems.

Note that BillingCenter offers two different fundamental ways to partition sequences of invoices. Think carefully for each case which pattern is the most appropriate:

- **Invoice streams** – Light-weight approach that makes sense if the payer is the same
- **Subaccounts** – Heavy-weight approach for separate payers or if cash must be partitioned

BillingCenter calls the `IInvoiceStream` plugin method `getInvoiceStreamPeriodicityFor` when it needs the periodicity for an invoice stream. Your plugin implementation must implement this method, which has the following method signature:

```
getInvoiceStreamPeriodicityFor(payer : InvoicePayer, paymentPlan : PaymentPlan,  
defaultInvoiceStreamPeriodicity : Periodicity) : Periodicity
```

Your implementation of the method `getInvoiceStreamPeriodicityFor` must return the periodicity of the `InvoiceStream` to use for the given payer and payment plan. By default, an account has four invoice streams for charges that have:

- A `weekly` payment plan.
- An `everyOtherWeek` payment plan.
- A `twicePerMonth` payment plan.
- A payment plan that is monthly or a multiple of monthly, such as `quarterly`, `everyFourMonths`, `everySixMonths`, `everyYear`, or `everyOtherYear`.

By default, a `Producer` has only one `InvoiceStream` and its periodicity is monthly. All charges for a producer are placed into the single `InvoiceStream` for that producer.

If there are any custom `Periodicity` types, then the `InvoicePayer` has an `InvoiceStream` for each custom `Periodicity`.

BillingCenter creates `InvoiceStream` objects only after there is a new charge to invoice in that stream.

Override this method to direct charges which have a given payment plan to an `InvoiceStream` with a different periodicity than the default. For example, if you have a custom periodicity such as `everyFiveMonths`, by default it gets its own stream. Override this method to invoice in one of the default streams, such as the monthly stream.

Get Existing Matching Invoice Stream

BillingCenter calls the `getExistingInvoiceStreamFor` plugin method to get an existing `InvoiceStream` of the payer to use for the given charge and invoice stream periodicity.

By default, this is just the invoice stream of the payer that has the given `invoiceStreamPeriodicity` passed as a parameter to the method. You can return null if there is no existing matching stream. Override this method if a payer can have more than one invoice stream with a given periodicity.

The method signature is:

```
getExistingInvoiceStreamFor(payer : InvoicePayer , charge : Charge,  
invoiceStreamPeriodicity : Periodicity,  
defaultExistingInvoiceStream : InvoiceStream) : InvoiceStream
```

This method returns an invoice stream.

Customize New Invoice Stream

BillingCenter calls the `customizeNewInvoiceStream` plugin method to customize a newly created `InvoiceStream`. By the time BillingCenter calls this method, BillingCenter has already initialized the given invoice stream to a default configuration. This method must add any customizations to that default instance.

Override this method to do any of the following:

- Set any `InvoiceStream` extension fields
- Set the override anchor dates for `InvoiceStream` instances with custom periodicities
- Change anything about the default initialization of invoice streams.

This method has two signatures:

```
customizeNewInvoiceStream(payer : InvoicePayer, newInvoiceStream : InvoiceStream)
customizeNewInvoiceStream(payer : InvoicePayer, charge : Charge, newInvoiceStream : InvoiceStream)
```

The method returns nothing.

Get Anchor Dates for Custom Periodicities

BillingCenter calls the `getAnchorDatesForCustomPeriodicity` plugin method to get anchor dates for existing invoice streams with custom periodicities. The default implementation of this plugin method does nothing.

Override this method if you have invoice streams that use custom periodicities. Alternatively, set the overriding anchor dates for new invoice streams in your implementation of the `customizeNewInvoiceStream` method.

The method signature is:

```
getAnchorDatesForCustomPeriodicity(invoicePayer : InvoicePayer, customPeriodicity : Periodicity) :
    List<AnchorDate>
```

The method returns a list of anchor dates.

Invoice Item Exception Information Plugin

BillingCenter calls this plugin to customize exception information on an invoice item that has changed. The plugin, `IInvoiceItem`, has three methods, each with a different signature. Each method enables changes to be made to the exception information associated with an invoice item. For each method, the `context.Event` argument specifies the change to the invoice item that caused the method to be called.

```
updateExceptionInfo(distItem : BaseDistItem, context : InvoiceItemExceptionContext)
```

Enables changes to the exception information associated with an invoice item that has a distribution item executed or reversed against it.

Possible `context.Event` values are: `Execute`, `Reverse`, `PromiseActivate`, and `PromiseDeactivate`.

```
updateExceptionInfo(writeoff    : Writeoff,
                    invoiceItem : InvoiceItem,
                    context      : InvoiceItemExceptionContext)
```

Enables changes to the exception information associated with an invoice item that has a write-off executed or reversed against it.

Possible `context.Event` values are: `Execute` and `Reverse`.

```
updateExceptionInfo(oldCommissionAmount : MonetaryAmount,
                     invoiceItem       : InvoiceItem,
                     context           : InvoiceItemExceptionContext)
```

Enables changes to the exception information associated with an invoice item that has had its primary active commission or gross amount changed.

Possible `context.Event` values are: `CommissionChange` and `GrossChange`.

Date Sequence Plugin

When BillingCenter creates a series of installments, installments are defined with a periodicity. A periodicity defines a regular series of dates with a certain frequency. For example, a monthly payment, a yearly payment, or every other week. BillingCenter represents a periodicity with values in the `Periodicity` typelist, which is an extendible typelist.

BillingCenter needs to convert a periodicity typecode into a class that knows how to calculate the next date in the sequence. For example, get the next installment date given the last installment date. This task is performed by subclasses of the abstract class `DateSequence` class in the `gw.api.domain.invoice` package. For example, the `WeeklyDateSequence` class knows how to generate the next date in a weekly periodicity. To convert a periodicity typecode into a date sequence class, it calls the `IDateSequence` plugin. It returns a new instance of a date sequence generator subclass.

This conversion typically requires a single anchor date. For example, a monthly periodicity starting on March 7, 2010 generates the next event dates April 7, 2010 and then May 7, 2010.

In some cases, there might be more than one anchor date, such as a twice-per-month periodicity, which allows you to provide an optional second date. If you do not provide the second anchor date, the sequence generates monthly date sequences interspersed with dates one half month in length. If you provide the second anchor date, you can define the date pattern for the every-other-week date. For example, if you provide March 2 and March 20, the next two dates are April 2 and April 20.

By default, BillingCenter knows how to convert the built-in periodicity typecodes. If you want to customize the behavior, write your version of the `IDateSequence` plugin or modify the default implementation (`gw.plugin.invoice.impl.DateSequence`). For example, if you add custom periodicity typecodes, you must provide a new `DateSequence` subclass.

To create a new `DateSequence` subclass, create a new Gosu class that extends the class `gw.api.domain.invoice.DateSequence`. Studio notifies you using compilation errors that you need to define the following methods on the abstract class to make the class fully concrete. Studio offers to automatically generate stubs for the required methods.

Required methods include:

- The `firstAfter` method must take a `DateTime` and return the first date in the Sequence that is after the specified `DateTime`.
- The `datesInInterval` method returns the dates within a time interval (`java.util.Interval`). It returns a sorted set of dates in an object of type `java.util.SortedSet<org.joda.time.DateTime>`. The `SortedSet` type is an interface and a common implementing class is `java.util.TreeSet`.
- The `contains` method determines whether a specific date is in the sequence, returning `true` or `false`.

The superclass `DateSequence` defines various other methods that you do not need to override.

In your `DateSequence` subclass, be warned that various method signatures use the type `org.joda.time.DateTime`. In Gosu, this class must be fully-qualified in every usage. A `uses` statement is not enough to disambiguate with the Gosu built-in `DateTime` class.

Instead of subclassing `DateSequence` directly, you can subclass `PeriodicDateSequence`, which is the built-in class that defines basic periods. Refer to that class in Studio for implementation details.

First, create your own `DateSequence` subclass in Studio. Next, modify the `DateSequence` plugin class, either the built-in class `gw.plugin.invoice.impl.DateSequence` or your own implementation. Assuming you modify the built-in `gw.plugin.invoice.impl.DateSequence` class or base your version on it, look for lines that look like this:

```
    } else if (thePeriodicity == Periodicity.TC_EVERYFOURMONTHS) {  
        return new MonthlyDateSequence(firstAnchorDate, 4)  
    } else if (thePeriodicity == Periodicity.TC_EVERYSIXMONTHS) {  
        return new MonthlyDateSequence(firstAnchorDate, 6)  
    } else if (thePeriodicity == Periodicity.TC_EVERYYEAR) {
```

```
    return new MonthlyDateSequence(firstAnchorDate, 12)
} else if (thePeriodicity == Periodicity.TC_EVERYOTHERYEAR) {
```

If you want to add a new `Periodicity` typecode, add another pair of lines to check for your new typecode, and return a new instance of your date sequence. You probably want to pass at least the first anchor date to your constructor for your date sequence class.

Agency Cycle Distribution Pre-fill Customization Plugin

BillingCenter supports *agency billing*. With agency billing, the carrier does not interact directly with the insured. Instead, an intermediary called a *producer* exists between the carrier and the insured. The producer bills the insured for any charges associated with an account or a policy, and the insured makes payments directly to the producer. The producer then pays the carrier, minus the amount owed to the producer for commission. Alternatively, the insured promises to pay the producer sometime in the future.

See also

- “Agency Bill Processing Overview” on page 339 in the *Application Guide*

Agency Bill Payments versus Agency Bill Promises

BillingCenter represents an agency bill payment made by an insured to a producer with an `AgencyCyclePayment` instance. BillingCenter represents an agency bill promise with an `AgencyCyclePromise` instance. Both entity types descend from the abstract supertype `AgencyCycleDist`. Some BillingCenter APIs and plugin interfaces operate on the supertype to support both payments and promises.

How The BillingCenter Agency Payment Wizard Uses the Pre-fill Plugin

In BillingCenter, agency bill payments are handled manually. The **Agency Payment Wizard** lets a user manually enter a payment amount from a producer and specify how to disburse the funds to statements, promises, and invoice items. For example, a producer might send one large check without specifying how the funds correspond to individual invoice items.

The wizard checks whether the payment amount from a producer matches the total payment amount expected, based on adding all individual invoice items. Before BillingCenter displays the **Agency Payment Wizard**, BillingCenter calls the `IAgencyCycleDist` plugin to pre-fill the wizard with agency cycle properties, such as gross, commission, and net amounts.

Typically, you do not need to override the default pre-fill behavior with your own implementation of the `IAgencyCycleDist` plugin interface. Implement your own version of this plugin for any of these reasons:

- You have a good reason to change the standard pre-fill behavior.
- You add pre-fill options to the wizard. For example, you extend the `AgencyCycleDistPreFill` typelist with more choices.

Implementing Your Own Agency Cycle Distribution Plugin

Your own plugin implementation must implement the `prefillAgencyCycleDist` plugin method. Its main parameter is an `AgencyCycleDist` instance, an abstract type with concrete subtypes for agency bill payments (`AgencyCyclePayment`) and agency bill promises (`AgencyCyclePromise`). Your plugin must support both agency bill payments and agency bill promises. You can implement different pre-fill behavior for payments and promises based on the subtypes `AgencyCyclePayment` and `AgencyCyclePromise`.

Pre-filling the Agency Payment Wizard for Agency Bill Payments

The `AgencyCyclePayment` entity represents a producer that sends the carrier a large check for all the money the producer collected from a set of insured customers. This entity indicates the total amount of the actual payment. The entity also provides details about the payment and what it represents. Specifically, it contains an array of payment (`AgencyPaymentItem`) and promise (`AgencyPaymentItem`) agency invoice item entities. Both entities are concrete subtypes of the `AgencyDistItem` entity. Plugin method signatures take `AgencyDistItem` entities to support both types.

Each payment or promise indicates the amount of funds from the total payment to apply to specific invoice items, broken out in the following properties:

- **Net amount for this invoice item** – The amount the producer pays to the carrier.
- **Commission for this invoice item** – The amount the producer keeps as commission.
- **Gross amount for this invoice item** – The amount the insured paid the producer. Typically, the gross amount is the same as the unpaid amount on the invoice item.

Your plugin determines and sets these amounts for each `AgencyDistItem` entity in the array that is the `AgencyDistItems` property of the `AgencyCycleDist` instance passed to your method. Your plugin returns the `AgencyDistItem` instances with new values in an `AgencyCycleDist` instance. The instance that your plugin returns does not have to be the same instance passed to your method. However, returning the same instance as the parameter is generally your best approach.

Pre-filling the Agency Payment Wizard for Agency Bill Promises

The `AgencyCyclePromise` entity does not represent an actual payment from a producer to the carrier. Instead, an `AgencyCyclePromise` instance represents a producer's promise to pay a carrier in advance of sending an actual payment. The producer sends this promise to the carrier as an estimate for what the producer owes the carrier for the agency cycle periodicity. The carrier often examines the promise to assure consistency with the expected future payment.

Responding to the Pre-fill Typecode Parameter

The `prefillAgencyCycleDist` method takes a second parameter, `prefill`, of type `AgencyCycleDistPreFill`. The `prefill` parameter is a code that indicates the type of pre-fill behavior requested by BillingCenter. In the default configuration, BillingCenter calls the method with these codes for specific pre-fill behaviors:

- `unpaid` – Set distribution items to appropriate amounts due.
- `zero` – Set all distribution item amounts to zero.

You can extend the `AgencyCycleDistPreFill` typelist with additional pre-fill types. If you do extend the typelist, your plugin must implement corresponding new pre-fill behavior.

Note: In the default configuration, BillingCenter never calls the `prefillAgencyCycleDist` method with the save code from the `AgencyCycleDistPreFill` typelist.

Premium Report Customization Plugin

BillingCenter includes a billing instruction called a premium report billing instruction (`PremiumReportBI`). This billing instruction notifies the associated policy period of a premium report. Policy administration systems periodically provide these reports to detail the actual exposure for an insured, which changes over time. BillingCenter can use this information to provide the insured with accurate invoices instead of a bill for an estimated amount. There is a plugin interface called `IPremiumReport` that you can implement to customize how BillingCenter handles these billing instructions.

The base configuration provides a default implementation of the plugin interface. The methods are implemented in the `PremiumReport` class in the package `gw.plugin.premiumreporting.impl`. These methods can be modified if special handling is required.

See also

- For detailed information about the integration of PolicyCenter and BillingCenter, see “Billing Integration” in the *PolicyCenter Integration Guide*.

Matching Due Dates for Premium Reports

When a `PremiumReportBI` billing instruction object is executed, BillingCenter matches the instruction to an existing due date. The due date is necessary to properly process the instruction. The logic that selects a matching due date is implemented in the plugin `findMatchingPremiumReportDueDate` method. The method accepts a `PremiumReportBI` object parameter. It returns a `PremiumReportDueDate` object with the matching due date.

When searching for a due date, the base configuration method compares the billing instruction’s due date to the following fields.

- premium report due date for the associated policy period
- start date for the policy period
- end date for the policy period

The method can be modified to implement special handling. For example, suppose you want to create a new property in the billing instruction that contained the public ID of the due date object. Use a find query that looks up the `PremiumReportDueDate` entity with that public ID. For example:

```
override function findMatchingPremiumReportDueDate(prbi : PremiumReportBI) : PremiumReportDueDate {
    return find (var prdd in PremiumReportDueDate where
        prdd.PremiumReportID == prbi.ID).getAtMostOneRow()
}
```

Because BillingCenter can find the due date object associated with this premium report, BillingCenter can mark that due date as satisfied so that delinquency does not happen.

The following is similar to the default implementation of this plugin method:

```
override function findMatchingPremiumReportDueDate(newPremiumReportBI : PremiumReportBI) :
    PremiumReportDueDate
{
    return find (var prdd in PremiumReportDueDate where
        prdd.PremiumReportDDPolicyPeriod == newPremiumReportBI.PremiumReportPolicyPeriod &&
        prdd.PeriodStartDate == newPremiumReportBI.PeriodStartDate &&
        prdd.PeriodEndDate == newPremiumReportBI.PeriodEndDate).getAtMostOneRow();
}
```

Customizing Failure to Report Delinquency for Premium Reports

If a premium report is not added by its due date, BillingCenter can begin a “Failure to Report” delinquency process. You can customize how BillingCenter starts and stops the delinquency process by modifying the appropriate plugin methods.

- `shouldStartFailureToReportDelinquency` – Determines whether to start a “Failure to Report” delinquency process. Returns `true` to start the process, else `false`. The base configuration returns `true` if no `PremiumReportBI` has been received by the due date. The method is called by the Premium Reporting Report Due batch process.
- `shouldStopFailureToReportDelinquency` – Determines whether to stop an active “Failure to Report” delinquency process. Returns `true` to stop the process, else `false`. The base configuration always returns `true`. The method is called whenever a `PremiumReportBI` billing instruction is received.

Account Information Customization Plugin

From an external system, you can request a summary of an account using a web service API. For more information, see “Get a Billing Object Summary” on page 118. That API returns an account information summary of type `AccountInfo`. Before the account info summary is sent to the external system, you can set properties on it or trigger custom logic. To add such customization, implement the `IAccountInfo` plugin.

For example, use this to populate data model extension properties that you add to the `AccountInfo` non-persistent entity. Suppose you want a separate web portal for customers to query BillingCenter for an account status. Suppose you want to display whether the account is past due for over 90 days. Instead of calculating past due dates in the external integration code, add a Boolean data model extension property. The field represents whether the account is past due for over 90 days. Use this plugin to populate the data model extension property.

The `IAccountInfo` plugin has only one method, `updateAccountInfo`, and you can do whatever is appropriate with the one argument, an `AccountInfo` entity. There is no return result from the method, as there are no specific requirements or tests required by this method.

See also

- For detailed information about integration of PolicyCenter and BillingCenter, see “Billing Integration” in the *PolicyCenter Integration Guide*.

Producer Information Customization Plugin

The producer information plugin interface (`IProducerInfo`) provides a way for customers to add logic to the non-persistent entity `ProducerInfo`. BillingCenter calls this plugin if BillingCenter receives an external integration request from the web services method `BCAPI.getProducerInfo(publicID)`. After BillingCenter creates the entity but before it returns it to the web services client, this plugin updates the `ProducerInfo` entity with custom logic. For example, it could set customer data model extension properties or other related data.

There is only one method, called `updateProducerInfo`. This method takes a `ProducerInfo` entity and returns nothing. Implement the one method and update the `ProducerInfo` entity directly, modifying it in place with whatever updates you require.

The default implementation of this plugin does nothing.

Account Distribution Limit Plugin

The Account Distribution Limit plugin is implemented by the `IAccount` interface. The plugin can customize the distribution limit for a given account. It also informs whether a hold exists on automatic disbursements, either for an entire account or for a specific `Unapplied` fund in the account. For details about the implementation provided in the base configuration, load the plugin source file located at `configuration → gsrc → gw → plugin → account → impl → Account.gs`.

Customizing the Distribution Limit

To customize the distribution limit for a given account, implement the `getDistributionLimitType` method of the `IAccount` interface. The method accepts an argument that references the account. A second argument specifies a distribution limit type, but this argument is reserved for future use and is currently ignored. The method returns the distribution limit type to use for the referenced account.

The default implementation of the plugin determines the appropriate distribution limit type based on the distribution filters defined in the account’s payment allocation plan.

Placing a Hold on Automatic Disbursements

The `IAccount` interface includes two methods that inform the caller on whether a hold exists on disbursements. A hold can be either on the entire account or on a specific `Unapplied` fund in the account. The methods return `True` if a hold exists, and `False` if there is no hold.

The two methods share the same name of `getShouldHoldAutomaticDisbursement`. One method accepts an argument of type `Account` to check for a hold on the entire account. The other method accepts an argument of type `Unapplied` to check for a hold on the specified `Unapplied` fund.

The implementation of these methods in the base configuration return immediately with a value of `False`.

The Automatic Disbursement batch process distributes funds from an account's `Unapplied` T-accounts. During its operation, the batch process invokes the `Unapplied` version of `getShouldHoldAutomaticDisbursement` to see if a hold exists on a particular `Unapplied` fund. If a hold does exist, the batch process does not distribute any money from the fund. For further information about the batch process, see "Automatic Disbursement Batch Processing" on page 123 in the *System Administration Guide*.

Collateral Cash Plugin

Implement the `ICollateral` plugin to customize how BillingCenter deals with collateral cash requirements.

The default implementation performs default behavior, and you can customize this as needed.

The methods you must implement are as follows:

- `createCashRequirementCharge` – Determine whether collateral creates a new charge to force cash requirement compliance. Return `true` to create a new charge. Otherwise, return `false`.
- `getCollateralChargeDate` – Determine the effective date of the collateral requirement charge (a `CollateralRequirement` object). This method must return a `Date` object.
- `getCollateralRequirementsSorted` – Get an ordered list of collateral requirements. Takes a `Collateral` object as an argument, and returns an array of `CollateralRequirement` objects.
- `paySegregatedCashRequirements` – Pay the segregated cash requirements. This method takes a `Collateral` object, a collateral payment distributed transaction object (`CollPmntDistributed`), and a `BigDecimal`. The `BigDecimal` represents the denormalized value of the total cash amount at collateral (the collateral-held `TAccount`). The method returns nothing.

For example:

```
class Collateral implements ICollateral {
    override function createCashRequirementCharge() : Boolean{
        return true;
    }

    override function getCollateralChargeDate(collateralRequirement : CollateralRequirement) : Date {
        if(collateralRequirement != null and collateralRequirement.PolicyPeriod != null){
            if(gw.api.util.DateUtil.verifyDateOnOrAfterToday(
                collateralRequirement.PolicyPeriod.PolicyPerEffDate )){
                return collateralRequirement.PolicyPeriod.PolicyPerEffDate;
            }
        }
        return gw.api.util.DateUtil.currentDate();
    }

    override function getCollateralRequirementsSorted(collateral : Collateral) : CollateralRequirement[]{
        var collateralRequirements = collateral.Requirements
        return collateralRequirements.sortBy( \ c -> c.EffectiveDate )
    }

    override function paySegregatedCashRequirements(collateral : Collateral,
        collPmntDistributed : CollPmntDistributed, totalCashAtCollateral : BigDecimal){
        var requirements = find(cr in CollateralRequirement where cr.Collateral == collateral
            and cr.Segregated
            and (cr.Compliance == ComplianceStatus.TC_COMPLIANT or

```

```
cr.Compliance == ComplianceStatus.TC_NOTCOMPLIANT)).iterator()
var availableCash = collPmntDistributed.Amount
while(requirements.hasNext()){
    var requirement = requirements.next() as CollateralRequirement
    requirement = collateral.Bundle.add( requirement )
    var neededAmount = requirement.Required - requirement.TotalCashValue
    if(neededAmount >0){
        var amountToAdd = BigDecimalUtil.min(availableCash, neededAmount)
        requirement.addToSegregated( amountToAdd, totalCashAtCollateral, collPmntDistributed)
        availableCash = availableCash - amountToAdd
        totalCashAtCollateral = totalCashAtCollateral - amountToAdd
        if(availableCash == 0){
            return
        }
    }
}
}
```

Agency Distribution Disposition Plugin

Implement the `IAgencyDistributionDisposition` plugin to customize how BillingCenter handles *distribution exceptions* during execution of an agency distribution from within the **Agency Payment Wizard** in the BillingCenter user interface. The term “exception” does not refer in any way to a programming exception. Instead, a distribution exception is the BillingCenter term for an invoice item that needs special handling.

For example, your plugin implementation can choose automatic handling provided by BillingCenter, it can carry the invoice item forward to the next invoice, or it can write off the amount. The agency distribution disposition plugin lets you customize the options in drop-down list for handling distribution exceptions in the **Agency Payment Wizard**. You can also use the methods in this plugin as general-purpose hooks for customizing behavior in the Payment wizard.

The the **Agency Payment Wizard** calls the methods of this plugin at the user entry points to certain wizard steps:

- `onEnterPaymentWizardEnterInformationStep` – Called upon entry to the wizard step for entering information
- `onEnterPaymentWizardSummaryStep` – Called upon entry to the wizard step that summarizes the payment
- `onPaymentWizardExecute` – Called upon entry to the wizard step for executing the payment

Policy System Plugin

Most BillingCenter integration points with a policy system start in the policy system, not BillingCenter. For example, if a user adds a new policy or changes a policy in the policy administration system, it sends the billing implications of the change to BillingCenter.

However, there are a few times in the application flow that BillingCenter needs to initiate a request to the policy system. For these cases, BillingCenter calls out to the currently registered implementation of the policy system plugin (`IPolicySystemPlugin`). For example, if an insured fails to pay, BillingCenter tells the policy system to start policy cancellation. If you use PolicyCenter, use the default implementation of this plugin to connect to Guidewire PolicyCenter.

Within BillingCenter, a messaging transport is the code that asynchronously notifies the policy system by calling the registered version of this plugin. The messaging transport is implemented by the `gw.plugin.pas.PASMessageTransport` class.

If you use PolicyCenter, set the currently-registered version of the plugin to the following class:

```
gw.plugin.pas.PCPolicySystemPlugin
```

If you need BillingCenter to work on its own in a demo mode, use the included implementation of the plugin which does not contact a real policy system:

```
gw.plugin.pas.StandAlonePolicySystemPlugin
```

If you use some other policy system, write your own version of this plugin to call out to the external policy system to request certain actions such as cancellation.

There are three methods that your version of this plugin must implement: `requestCancellation`, `rescindCancellation`, and `notifyPaymentReceivedForRenewalOffer`. The rest of this topic discusses these methods, and also a method which BillingCenter calls to confirm a policy period when sufficient payment is received. All of these methods use a transaction ID, which is a `String` value that uniquely identifies this request from BillingCenter. Your plugin code or the downstream system that it represents must be able to determine whether this request has already occurred. In other words, your plugin code or the external system must determine if this is a duplicate request.

For more information about the structure of transaction IDs, see “BillingCenter Data Transfer Objects (DTOs)” on page 84.

Cancellation

The most important method is the `requestCancellation` method, which takes as arguments:

- Policy period entity (a `PolicyPeriod`)
- Delinquency reason (a `DelinquencyReason` typecode)
- Transaction ID (a `String`)

This method must tell the external system to start policy cancellation for this policy.

Rescinding a Cancellation

If a payment comes in later, BillingCenter tells the policy system to rescind the cancellation, which means to cancel the cancellation process as if it had not happened. To handle rescinding the policy in the external system, your plugin must implement the `rescindCancellation` method. It takes the same arguments as the `requestCancellation` method:

- Policy period entity (a `PolicyPeriod`)
- Delinquency reason (a `DelinquencyReason` typecode)
- Transaction ID (a `String`)

Renewal Offers

There is one more method that your plugin method must handle. However, not all carriers might use it. It depends on whether your policy system supports renewal offers. There are two basic strategies for a policy system to handle renewals. The following table summarizes the differences between the bind-and-cancel flow and the renewal offers flow:

| Renewal flow | What happens immediately | What happens if the insured pays | What happens if the insured does not pay |
|-----------------|---|--|---|
| Bind and cancel | A user binds a renewal, which triggers the policy system to send charges to BillingCenter. | If the insured sends a payment, all happens normally. | If BillingCenter does not receive a payment, BillingCenter tells the policy system to perform a flat cancel for non-payment. Money received after any cancellation remains in Account: Unapplied. |
| Renewal offers | <p>Instead of actually binding the renewal, the policy system sends out a renewal offer. The renewal offer has a reference number, which is the renewal job ID.</p> <p>The actual renewal offer letter does not happen in the default integration. However, you can configure the policy system to call the billing system to get a preview of the payments. You might want to include that information on a renewal offer letter. Be sure this represents the actual amount due including installment or invoice fees so the insured knows how much money to send.</p> | <p>As a payment uploads from the bank (or maybe entered into the BillingCenter UI), it includes the reference number. Using that reference number, BillingCenter tells the policy system that it received payment for a particular renewal term that BillingCenter otherwise knows nothing about.</p> <p>Meanwhile, BillingCenter holds the payment as a suspense payment on the account. A suspense payment is a payment that cannot yet be matched to both an account and a policy. The billing system waits for more information or human intervention to help match it, rather than applying it to any current charges. Until the suspense payment is applied, it does not directly affect the account or policy.</p> <p>When the policy system sends the renewal billing information to BillingCenter, it also has the reference number. BillingCenter looks for any suspense payments with the reference number and applies the payment.</p> | <p>The policy system considers it Not Taken if the payment is late or does not arrive.</p> <p>If it arrives later, BillingCenter attempts to tell the policy system about the payment received. the policy system returns an error saying the renewal is no longer capable of renewal. This causes BillingCenter to create an activity for someone to look into the new payment that it cannot apply.</p> |

To handle renewal offers, your plugin must implement the `notifyPaymentReceivedForRenewalOffer` method.

This plugin method must notify the external policy system about a new suspense payment that BillingCenter received. The method takes a suspense payment entity (a `SuspensePayment` entity) and a transaction ID (a `String`).

If the external policy system cannot honor this offer, this plugin method must throw an exception. For example:

- The offer might have expired.
- The policy is in an incompatible state for renewal.

BillingCenter automatically detects any exceptions this plugin throws and creates an activity for someone to investigate the status of this renewal. For instance, whether to refund it or investigate whether renewal is actually possible after some temporary issue.

Confirming a Policy Period

When a payment is received for a policy period, BillingCenter calls the `hasReceivedSufficientPaymentToConfirmPolicyPeriod` method in the `IPolicyPeriod` plugin. If this method returns `true`, and the `ConfirmationNotificationState` property of the `PolicyPeriod` is set to `NotifyUponSufficientPayment`, BillingCenter calls `confirmPolicyPeriod` to send a message to the policy system.

Direct Bill Payment Plugin

The Direct Bill Payment plugin determines how BillingCenter allocates money to individual invoice items. The methods supported by the plugin are declared in the `IDirectBillPayment` interface. BillingCenter implements a base configuration of the interface in the `DirectBillPayment` class. The methods in the class can be modified to allocate money in a customized manner.

The following sections describe the methods implemented in the `DirectBillPayment` class.

Customize the Filter Criteria for Payment Allocations

Before allocating a payment with the `allocatePayment` plugin method, BillingCenter filters invoice items to identify and select the items eligible to be paid.

The filter criteria are initialized in the `addInvoiceItemsDistributionCriteria` method. The criteria can be customized to meet unique allocation requirements.

```
function addInvoiceItemsDistributionCriteria( moneyRcvd : DirectBillMoneyRcvd ) :  
    RestrictionBuilder<InvoiceItem>
```

The `moneyRcvd` argument specifies the payment to allocate.

The returned `RestrictionBuilder` object specifies the filter criteria to use to select eligible invoice items. In the base configuration, the following criteria restrictions are combined to create the `RestrictionBuilder` object.

- Unapplied fund restrictions
- Premium Report billing instruction restrictions that have no distribution limits
- Collateral item restrictions
- Collateral requirement item restrictions

To customize the filter criteria, assign the desired restrictions to the `RestrictionBuilder` object. For information about the `RestrictionBuilder` class, see “[Restriction Builder](#)” on page 185 in the *Gosu Reference Guide*.

The `addInvoiceItemsDistributionCriteria` method can be called when any of the following events occur:

- A user manually enters a new payment or modifies an existing payment in the user interface and selects **Execute**.
- A batch process that distributes money is executed. The applicable batch processes are described in “[Batch Processes That Distribute Payments](#)” on page 294 in the *Application Guide*.
- The `DirectBillPaymentFactory` class `createAllocatedPayment` method is called.

Allocate a Payment

BillingCenter usually receives payments from an external system. Payments can also be entered manually through the BillingCenter user interface. The received payment is not immediately distributed to invoice items. The payment is distributed by the `DirectBillPayment` plugin `allocatePayment` method. The money is allocated based on the settings of the applicable payment allocation plan.

```
function allocatePayment( payment : DirectBillPayment, amount : MonetaryAmount )
```

The `payment` argument is used to determine the invoice items to receive the allocation. The `amount` argument specifies the monetary amount to allocate.

The `allocatePayment` method can be called in the same situations that call the `addInvoiceItemsDistributionCriteria` method. For details, refer to “Customize the Filter Criteria for Payment Allocations” on page 192.

Allocate Payment to a Policy Period

After allocating a payment with the `allocatePayment` plugin method, some payment money may still remain unallocated. The unallocated portion of the payment can be distributed to invoice items from another payer on another policy period by the `allocatePolicyPeriod` method. The money is allocated based on the settings of the applicable payment allocation plan.

```
function allocatePolicyPeriod( payment : DirectBillPayment, amount : MonetaryAmount,  
otherEligiblePayerInvoiceItems : Set<InvoiceItem> )
```

The `payment` argument references the payment processed by the `allocatePayment` method. The `amount` argument specifies the remaining monetary amount to distribute.

The `otherEligiblePayerInvoiceItems` argument specifies a set of additional invoice items from another payer that are eligible to be paid. This argument is not used by the method implemented in the base configuration, but it is available for use by customized versions.

The `allocatePolicyPeriod` method can be called when any of the following events occur:

- A batch process that distributes money is executed. The applicable batch processes are described in “Batch Processes That Distribute Payments” on page 294 in the *Application Guide*.
- The `DirectBillMoneyRcvd` class `distribute` method is called.

Customize the Filter Criteria for Credit Allocations

Before allocating a policy or account credit with the `allocateCredit` plugin method, BillingCenter filters invoice items to identify and select the items eligible to be paid.

Policy Credit Allocation

For a policy credit allocation, one filter criteria is specified by the `Positive Item Qualifier` property contained in the return premium plan. The base configuration provides several property settings that are described in the section “Positive Item Qualifier” on page 196 in the *Application Guide*.

The filter criteria of the `Positive Item Qualifier` are initialized in the `addPositiveItemsDistributionCriteria` method. The criteria can be customized to meet unique requirements.

```
function addPositiveItemsDistributionCriteria( negativeInvoiceItems : Iterable<InvoiceItem> ) :  
RestrictionBuilder<InvoiceItem>
```

The `negativeInvoiceItems` argument specifies the credit invoice items to allocate. The items are used to retrieve the appropriate credit allocation settings, such as the value of the `Positive Item Qualifier` property and the relevant credit handling scheme.

The filter criteria of the Positive Item Qualifier are used to create the returned RestrictionBuilder object. The RestrictionBuilder is subsequently used by BillingCenter to select the invoice items eligible to receive the credit allocation.

To customize the filter criteria, assign the desired restrictions to the RestrictionBuilder object. For information about the RestrictionBuilder class, see “Restriction Builder” on page 185 in the *Gosu Reference Guide*.

The addPositiveItemsDistributionCriteria method can be called when any of the following events occur:

- Allocating a policy credit, such as when processing a payment plan change that adds a credit item.
- Executing a ChargeInstallmentChanger object that adds a credit item.
- The Account class becomePayerOfInvoiceItems method is called.

Account Credit Allocation

For an account credit allocation, the filter criteria are initialized in the addInvoiceItemsDistributionCriteriaForAccountCredits method. The criteria can be customized to meet unique requirements.

```
function addInvoiceItemsDistributionCriteriaForAccountCredits(  
    account : Account, negativeInvoiceItems : Iterable<InvoiceItem> ) :  
    RestrictionBuilder<InvoiceItem>
```

The account argument specifies the applicable account. The negativeInvoiceItems argument specifies the credit invoice items to allocate.

The returned RestrictionBuilder object specifies the filter criteria to use to select eligible invoice items. In the base configuration, the following criteria restrictions are combined to create the RestrictionBuilder object.

- Restrictions applicable to the account’s payment allocation plan that have no distribution limits
- Past due restrictions
- Premium Report billing instruction restrictions
- Collateral item restrictions
- Collateral requirement item restrictions

To customize the filter criteria, assign the desired restrictions to the RestrictionBuilder object. For information about the RestrictionBuilder class, see “Restriction Builder” on page 185 in the *Gosu Reference Guide*.

The addInvoiceItemsDistributionCriteriaForAccountCredits method can be called when any of the following events occur:

- Allocating an account credit.
- The DirectBillPaymentFactory class payFromNegativeInvoiceItems method is called.

Allocate Credits

BillingCenter can allocate negative credit amounts to “pay” for positive invoice items. The process is described in “Credit Handling” on page 191 in the *Application Guide*.

The allocateCredits method applies credit items to “pay” for positive invoice items.

```
function allocateCredits( payment : DirectBillPayment )
```

The payment argument includes the credit items to allocate.

The method validates that a negative credit exists and that there are positive invoice items eligible to receive the payment. If either item does not exist, the allocation process is aborted.

The method determines the appropriate allocation method based on the type of credit being allocated. Account and collateral credits, such as account return fees or collateral charges, are always allocated using the First to Last allocation method. To process this type of credit, the allocateCredits method calls the applicable allocate method.

Policy credits, such as return premium credits, are allocated based on the relevant handling scheme for the type of credit being processed. Each handling scheme implements its own unique version of the `allocate` method. New handling schemes can be created to customize credit allocation for different types of credit. For detailed information and an example, see “Configuring Credit Allocation” on page 454 in the *Configuration Guide*.

When processing policy credits, if all positive invoice items are paid and a credit balance remains, the base configuration places the remaining money in the `Unapplied` account. By customizing the credit allocation process, this remaining money can be handled in alternative manners. Examples that demonstrate disbursing the remaining credit or placing it in a suspense item are described in “Defining New Excess Treatment Settings” on page 452 in the *Configuration Guide*.

The `allocateCredit` method can be called when any of the following events occur:

- A policy is changed that results in a credit.
- An invoice item’s amount is changed, the item is canceled, or the item’s account payer is changed.
- An invoice is billed or becomes due.
- The `DirectBillPaymentFactory` class `payFromNegativeInvoiceItems` method is called.

Customize the Filter Criteria for Unapplied Fund Allocations

Before allocating money with the `allocateFromUnapplied` plugin method, BillingCenter filters invoice items to identify and select the items eligible to be paid.

The filter criteria are initialized in the `addInvoiceItemsDistributionCriteriaForUnappliedFunds` method. The criteria can be customized to meet unique requirements.

```
function addInvoiceItemsDistributionCriteriaForUnappliedFunds( zeroDollarDMR: ZeroDollarDMR ) :  
    RestrictionBuilder<InvoiceItem>
```

The `zeroDollarDMR` argument contains the `Unapplied` fund to allocate money from.

The returned `RestrictionBuilder` object specifies the filter criteria to use to select eligible invoice items. In the base configuration, the following criteria restrictions are combined to create the `RestrictionBuilder` object.

- Restrictions applicable to zero-dollar “payments”
- Premium Report billing instruction restrictions that have no distribution limits
- Collateral item restrictions
- Collateral requirement item restrictions

To customize the filter criteria, assign the desired restrictions to the `RestrictionBuilder` object. For information about the `RestrictionBuilder` class, see “Restriction Builder” on page 185 in the *Gosu Reference Guide*.

The `addInvoiceItemsDistributionCriteriaForUnappliedFunds` method can be called when any of the following events occur:

- Allocating or redistributing money from a default or designated `Unapplied` fund.
- The Automatic Disbursement batch process is run and there is no distribution hold on the money.
- Any of the following `DirectBillPaymentFactory` static methods is called:
 - `payFromDefaultUnapplied`
 - `payUsingUpToAmountOfDefaultUnapplied`
 - `payUsingUpToAmountOfDesignatedUnapplied`

Allocate from Unapplied Fund

The `allocateFromUnapplied` method allocates money stored in the `Unapplied` account. The money is allocated based on the settings of the applicable payment allocation plan.

```
function allocateFromUnapplied( payment : DirectBillPayment, amount : MonetaryAmount )
```

The `payment` argument contains the list of invoice items eligible to receive the money. The `amount` argument specifies the amount of money to allocate.

The `allocateFromUnapplied` method can be called in the same situations that call the `addInvoiceItemsDistributionCriteriaForUnappliedFunds` method. For details, see “Customize the Filter Criteria for Unapplied Fund Allocations” on page 195.

Allocate for Redistribution

The `allocateForRedistribution` method redistributes an executed payment. Redistribution can occur when a payment plan is changed or when a payment has been partially distributed, but some money still remains in an `Unapplied` fund. The method is implemented with the following signatures:

```
function allocateForRedistribution( payment : DirectBillPayment, amount : MonetaryAmount )  
function allocateForRedistribution( payment : DirectBillPayment,  
                                  amountToChargeMap : Map<Charge, AllocationPool> )
```

The `payment` argument includes payment information, including the list of invoice items to receive the redistributed money. The `amount` argument specifies the monetary amount to redistribute. The `amountToChargeMap` argument specifies a map of charges and associated amounts to distribute.

The `allocateForRedistribution(DirectBillPayment, MonetaryAmount)` method signature can be called when any of the following events occur:

- The payment plan is changed causing the invoice items to be redistributed.
- The `Automatic Disbursement` batch process is run. The batch process redistributes payments before issuing disbursements.
- The `Account` class `redistributeAnyUnappliedFunds` method is called.
- The `DirectBillPayment` class `redistribute` method is called.

The `allocateForRedistribution(DirectBillPayment, Map<Charge, AllocationPool>)` method signature is called by the `BaseDist` class `redistributeWithAmounts` method.

Allocate Collateral

The `allocateCollateral` method allocates a specified monetary amount from an `Unapplied` fund to collateral items. The money is allocated based on the settings of the applicable payment allocation plan.

```
function allocateCollateral( payment : DirectBillPayment, amount : MonetaryAmount )
```

The `payment` argument contains the collateral items to receive the `Unapplied` funds. The `amount` argument specifies the monetary amount to allocate.

The `allocateCollateral` method is called by the `Collateral` class `transferAccountUnappliedToCollateral` and `transferDefaultUnappliedToCollateral` methods. The base configuration does not call these methods, but they are available to be called by configuration code.

Performance Improvement if Not Using Collateral

If collateral is not being used, the performance of payment distribution can be improved. To achieve the performance gain, edit the `DirectBillPayment` class by commenting out all calls to the following methods. The methods themselves can also be commented out.

- `makeCollateralItemsRestrictionBuilder`
- `makeCollateralRequirementItemsRestrictionBuilder`

This modification removes the unions for `CollateralItems` and `CollateralRequirementItems` from the queries that determine the invoice items to be paid.

Allocate with Overrides

The `allocateWithOverrides` method allocates payments to invoice items that can have overrides associated with them. The money is allocated based on the settings of the applicable payment allocation plan. The method is implemented with the following signatures.

```
function allocateWithOverrides( payment : DirectBillPayment, amount : MonetaryAmount,  
overrideItemsAndAmounts : Map<InvoiceItem, MonetaryAmount> )  
  
function allocateWithOverrides( payment : DirectBillPayment, amount : MonetaryAmount,  
groupsToDistribute : List<DBPaymentDistItemGroup> )
```

The `payment` argument contains the invoice items to receive the allocation. The `amount` argument specifies the monetary amount to allocate. The argument's value can be larger than the payment amount if the argument includes money stored in the Unapplied T-account. A customized version of the method can choose to allocate only the payment amount or the potentially larger value specified in the `amount` argument. The method must never allocate more than the value specified in the `amount` argument.

The `overrideItemsAndAmounts` argument specifies a map of individual override invoice items and their monetary amounts.

The `groupsToDistribute` argument specifies the overrides in a list of direct bill payment distribution item groups. Each group includes:

- A list of the individual invoice items
- Group information, such as the total unbilled amount of the group

The `allocateWithOverrides` method is called by the `DirectBillPaymentView` class `recalculateDistribution` method. If the `DirectBillPaymentView` object is aggregated by items, the method calls the signature version of `allocateWithOverrides` that accepts a map of individual invoice items. If the object is aggregated by summary, it calls the method signature version that accepts a list of groups. In the base configuration, the `recalculateDistribution` method is called from the `DirectBillAddInvoiceItemsPopup.pcf` user interface file, which is shown when moving a payment from one account to another.

Allocate Write-offs

The `allocateWriteoffs` method processes invoice items that have been written off.

```
function allocateWriteoffs( writeoff : ChargeGrossWriteoff ) : List<InvoiceItemAllocation>
```

The `writeoff` argument contains the eligible invoice items to write off. The method returns a list of invoice items that were processed.

The invoice items are written off based on the applicable `proRataAllocation` method.

The `allocateWriteoffs` method can be called when any of the following events occur:

- A charge is written off using the user interface.
- The `Writeoff Staging` batch process is run.

Suspense Payment Plugin

Payments received by BillingCenter always specify the payer. The payer can be either an account, a producer, or a policy period that is associated with an account or producer.

It is possible for BillingCenter to receive payments before it has been informed about the payer. In such cases, the account, producer, or policy period does not yet exist in BillingCenter, but will presumably be created in the near future.

When a payment is received and the payer is unknown to BillingCenter, the payment is placed in an **Unapplied T-account** of type **SuspensePayment**. Each payment with an unknown payer is placed in its own, separate **Unapplied T-account**. For example, if ten payments with unknown payers arrive then ten **Unapplied SuspensePayment T-accounts** are created.

Subsequently, the **Suspense Payment** batch process searches for the payer of each payment stored in these **Unapplied T-accounts**, checking to see if the account or producer has been created yet. If the payer is found to exist in BillingCenter, the payment is moved from the **Unapplied SuspensePayment T-account** to the appropriate **Unapplied T-account** of the payer.

All operations performed by the **Suspense Payment** batch process are implemented in the configurable **Suspense Payment** plugin. The **Suspense Payment** plugin implements the **ISuspensePayment** plugin interface. For each **Unapplied SuspensePayment T-account** with a status of **Open**, the batch process calls the plugin interface's **apply** method to process the payment and attempt to identify the payer. By implementing the **Suspense Payment** plugin interface's **apply** method, the functionality of the **Suspense Payment** batch process can be configured to perform alternative or additional operations.

The BillingCenter base configuration includes an implementation of the **Suspense Payment** plugin interface. To view the plugin source file in Studio, activate the **Project** window, navigate to **configuration** → **config** → **Plugins** → **registry**, and open **SuspensePayment.gw**. To configure the **Suspense Payment** plugin interface, either modify the plugin source file of the base configuration or register your own implementation.

In the plugin source file of the base configuration, the **SuspensePayment** class implements the **ISuspensePayment** interface and the interface's **apply** method. The **apply** method accepts a single parameter: the suspense payment to process. The method returns no value. The **apply** method performs the following steps in its attempt to identify the applicable payer.

1. If a renewal offer number is specified for the suspense payment, return without processing the payment. A renewal offer number exists only if BillingCenter is integrated with PolicyCenter. The renewal number is a unique ID that corresponds to a PolicyCenter transaction number. For more information about renewal offers, see “Policy System Plugin” on page 189.
2. If an account number is specified for the suspense payment, a search is performed for a matching account. If the account is found, the payment is moved to the default **Unapplied T-account** owned by the account.
3. If a policy number is specified for the suspense payment, a search is performed for a matching policy period. If one or more policy periods are found, the latest policy period is used. The payment is moved to the **Unapplied T-account** associated with the policy period. Depending on the account's billing level, the **Unapplied T-account** will be either the account's default **Unapplied** or the designated **Unapplied** associated with the policy. For further information about default and designated **Unapplied T-accounts**, see “Billing Levels” on page 299 in the *Application Guide*.
4. If a producer name is specified for the suspense payment, a search is performed for a matching producer. If multiple producers exist with the specified name, return without processing the payment. If a single matching producer is found, the payment is moved to the default **Unapplied T-account** owned by the producer.

Funds Tracking Plugin

Guidewire provides the **IFundsTracking** plugin to integrate your own funds allotment logic with funds tracking in BillingCenter. The plugin also lets you integrate your own logic to aggregate payment items into payment item groups. The default configuration of BillingCenter uses a built-in implementation of the **IFundsTracking** plugin, **FundsTracking.gs**. To open the plugin registry, in the **Project** window in Studio, navigate to **configuration** → **config** → **Plugins** → **registry**, and then open **IFundsTracking.gwp**.

Overview of Funds Tracking Integration

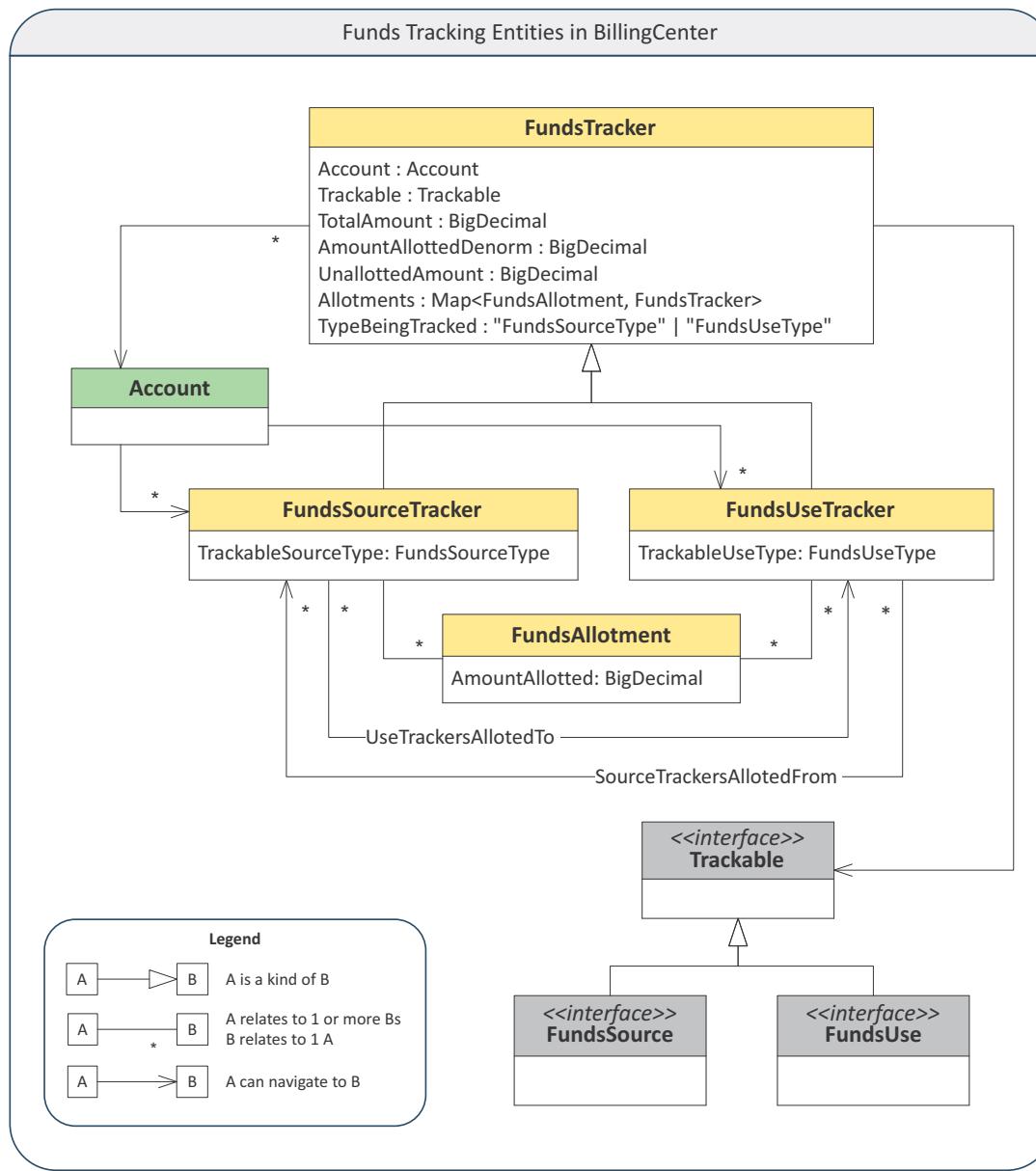
Billing clerks can distribute payments directly within an account, but surplus funds that cannot be distributed directly remain in Account: Unapplied. Funds tracking provides an alternative view of how funds flow through an account, independent of the general ledger view that payment distribution by clerks provides. Funds tracking is completely independent of other parts of BillingCenter. Other BillingCenter features do not depend on funds tracking data. If you enable funds tracking, payment distribution works the same as before.

About Funds Trackers and Trackables

Funds tracking uses the following Gosu interfaces and entity types to track and provide details about funds that flow through an account:

- **funds sources** – Money in, such as a payment, that adds funds to Account: Unapplied
- **funds uses** – Money out, such as an invoice item paid, that subtracts funds from Account: Unapplied
- **funds source trackers** – Funds tracking representations of funds sources
- **funds use trackers** – Funds tracking representations of funds uses

- **funds allotments** – Associations between funds source trackers and funds use trackers



BillingCenter provides the `FundsSource` and `FundsUse` marker interfaces to identify entity types that are funds source or uses. For example, the `DirectBillMoneyRcvd` entity type is a kind of `FundsSource`. The `PaymentItemGroup` entity type, which is a collection of payment items, is a kind of `FundsUse`.

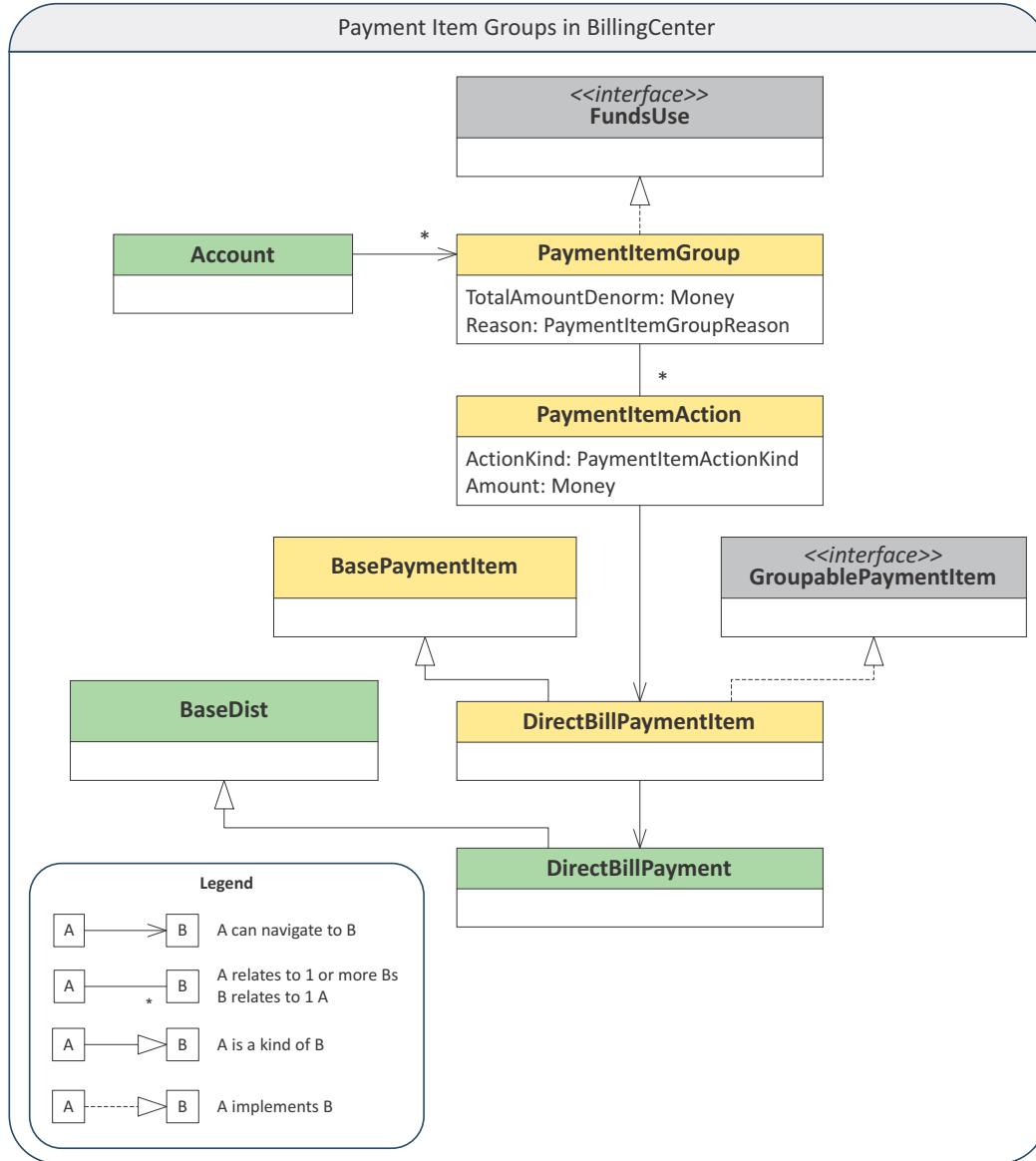
For each source and use of funds, a corresponding `FundsSourceTracker` or `FundsUseTracker` instance provides tracking information, such as the total amount and the amount allotted. The `FundsAllotment` entity type associates funds sources with funds uses.

For a complete list of entity types that are funds sources or funds uses, see “Funds Tracking Basics” on page 241 in the *Application Guide*.

About Payments and Payment Item Groups

Funds tracking uses the following Gosu interfaces and entity types to group payment items into a single, trackable funds use:

- **payment item group** – A trackable funds use that comprises one or more payment actions
- **payment item action** – An association between a payment item group and a payment item
- **groupable payment item** – Any payment item type that can be included in a payment item group
- **payment item** – A funds tracking representation of a payment distribution



BillingCenter provides the **PaymentItemGroup** entity type as single funds use to represent one or more payment distributions. The **Funds Tracking** screen uses payment item groups to show a simplified list of funds uses. For example, the **Funds Uses** tab shows a direct bill payment as a single payment item group, even if the payment was applied to multiple invoice items.

The **PaymentItemAction** entity type associates various types of payment items with a payment item group. The **GroupablePaymentItem** marker interface identifies payment item types that payment item groups can include. In the default configuration, the **DirectBillPaymentItem** type is the only type that a **PaymentItemAction** can associate with a **PaymentItemGroup**. A **DirectBillPaymentItem** instance relates to a single **DirectBillPayment** instance.

You can change the default configuration of funds tracking to let other payment distribution types be included in a payment item group. Write a new Gosu class that extends the `BasePaymentItem` class and implements the `GroupablePaymentItem` marker interface. Include a property to reference an instance of the payment distribution type that your new payment item type represents. The `GroupablePaymentItem` interface lets BillingCenter know which payment distribution types to include in the `PaymentItemNarrative` instances that it passes to the plugin method `createPaymentItemGroupings`. A `PaymentItemNarrative` is a directed acyclic graph of payment actions that occurred in a single, committed bundle.

Note: Unlike payment distributions, other types of transactions, such as account transfers, write-offs, and disbursements are themselves funds sources or funds uses. They implement the `FundsSource` or `FundsUse` interfaces, so they do not need an aggregating entity to be trackable by funds tracking in BillingCenter.

For more information, see “Payment Item Groups” on page 244 in the *Application Guide*.

How BillingCenter Uses the Funds Tracking Plugin

BillingCenter uses the funds tracking plugin for two purposes:

- **Creating payment item groups** – Aggregates related payment items into new payment item groups
- **Allotting funds** – Associates funds sources with funds uses

How BillingCenter Creates Payment Item Groups with the Funds Tracking Plugin

Aggregation of payment items into payment item groups occurs when a bundle commits and it contains an executed or reversed payment item as part of the committed work. When a clerk or automated process distributes money that comes in, BillingCenter creates `PaymentItem` instances to record information about the distributions. When the bundle containing the payment items commits, BillingCenter calls the funds tracking plugin.

For example, a clerk distributes a \$150 payment received to two invoice items on latest premium bill. The clerk distributes \$145 to pay the monthly premium and separately distributes the remaining \$5 to the monthly installment fee. As a result, BillingCenter creates two payment items to track the distribution of funds. When the bundle that contains the distributions and associated payment items commits, the plugin creates a payment item group to track them as a single funds use.

For more information, see “How the Funds Tracking Plugin Creates Payment Item Groups” on page 204.

How BillingCenter Allots Funds with the Funds Tracking Plugin

An allotment process occurs in two ways:

- **Background** – Whenever you schedule the `FundsAllotmentBatchProcess` batch process or distributed work queue to run.
- **Foreground** – When someone accesses the `Funds Tracking` screen or returns from a related popup

The background `FundsAllotmentBatchProcess` can be run on demand or a schedule that you specify. The writer finds accounts with related funds sources that are partially or not at all allotted. The writer creates one work item for each account that meets the criteria. The worker then calls `Account.allotAllFunds` for each work item.

The foreground `AccountDetailFundTracking` screen calls `Account.allotAllFunds` on entering the screen and on return from related pop-ups. The screen and pop-ups invoke this just-in-time allotment so the clerk sees the most recent funds tracking information. The set of transactions for an account may have changed since the batch process last ran.

For example, a clerk records a payment and then goes to the `Funds Tracking` screen. The clerk expects to see the payment allotted to a payment item group. Without the just-in-time allotment, the clerk sees the payment as not allotted.

For details, see “How the Funds Tracking Plugin Allots Funds” on page 204.

Limitations of Funds Tracking in BillingCenter

Consider these limitations in the default configuration of funds tracking:

- **Only direct bill money received funds can be tracked.**

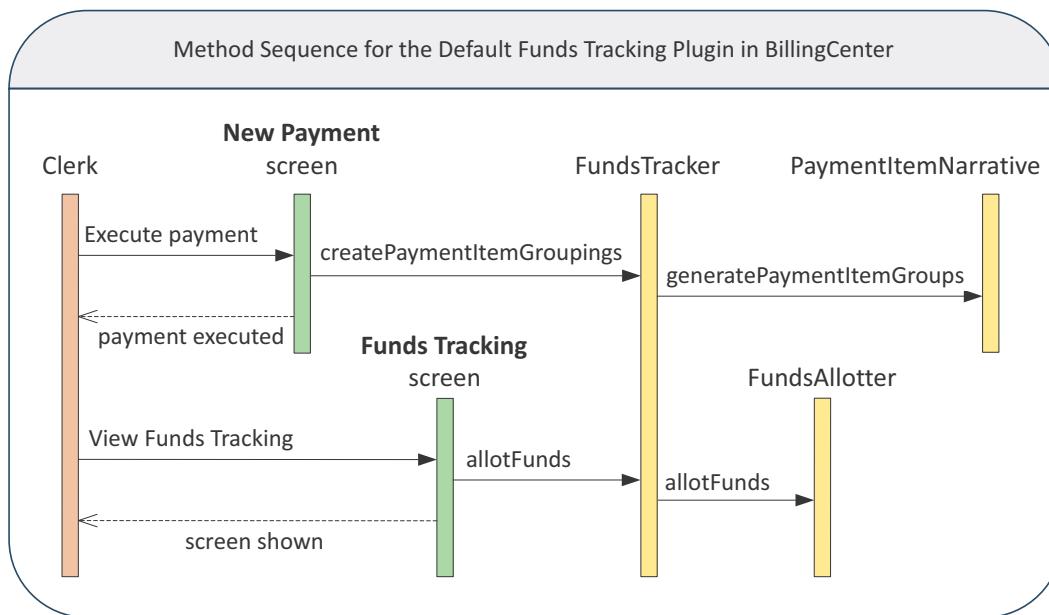
You can change the default configuration to track other kinds of money received funds.

- **Only funds within a single account can be tracked.**

For account-to-account transfers, you must examine both accounts to see the complete flow of tracked funds.

Implementations of the Funds Tracking Plugin

The default configuration of BillingCenter provides a single, built-in, fully functional implementation of the funds tracking plugin, `IFundsTracking`. In the default configuration, the `IFundsTracking` plugin is enabled and configured to use the built-in implementation, `FundsTracking.gs`.



The `FundsTracking` implementation fully contains the logic for creating payment item groups. The `FundsTracking` implementation calls a helper method on the `PaymentItemNarrative` that it receives as the sole parameter passed to its `createPaymentItemGroupings` method.

The `FundsTracking` implementation defers to the `FundsAllotter` class to implement the funds allotment logic. BillingCenter calls `FundsTracker.allotFunds`, which in turn calls `FundsAllotter.allotFunds`. The `FundsAllotter.allotFunds` method implements the two-phase allocation process of priority reallocation, followed by allotment.

See also

- “How the Funds Tracking Plugin Creates Payment Item Groups” on page 204
- “How the Funds Tracking Plugin Allots Funds” on page 204

Writing a Funds Tracking Plugin

To implement your own payment item grouping logic in BillingCenter, you must modify the `FundsTracking` class. It contains all the default logic. The default logic creates payment item groups that mimic the distribution actions taken by clerks and automated BillingCenter actions. Therefore, you are unlikely to want to change this logic.

To implement your own funds allotment logic in BillingCenter, use the built-in Gosu implementation as a starting point. To change the default allotment logic to suit your business needs, you can change or replace entirely the `FundsAllotter` class that the `FundsTracking` class calls. Or, you can change or replace the `FundsTracking` implementation entirely. If you change the `FundsTracking` implementation, your version does not need to use the `FundsAllotter` class.

Methods on the Funds Tracking Plugin

The `IFundsTracking` plugin interface defines two methods:

- `createPaymentItemGroupings` – Takes a `PaymentItemNarrative` and creates one or more `PaymentItemGroups`
- `allotFunds` – Takes a Set of `SourceFundsTrackers` and a Set of `SourceFundsUses` that are not fully allotted and allots funds until sources are exhausted or uses are fully allotted

See also

- “How the Funds Tracking Plugin Creates Payment Item Groups” on page 204
- “How the Funds Tracking Plugin Allots Funds” on page 204

How the Funds Tracking Plugin Creates Payment Item Groups

BillingCenter calls the `createPaymentItemGroupings` method on `IFundsTracking` whenever a bundle commits contains an executed or reversed payment item as part of the committed work. This results in one or more `PaymentItemGroup` instances. A `PaymentItemGroup` is a `FundsSource` or a `FundsUse`, depending on whether the total amount from the payment items in the group is positive or negative.

BillingCenter passes the payment items to group to the plugin in a `PaymentItemsNarrative` instance. The plugin uses the helper method `PaymentItemsNarrative.generatePaymentItemGroups` to create the payment item groups, after the plugin method aggregates the payment actions in the narrative into groupings.

In the default configuration, the plugin aggregates payment items according to the T-account owners of the charge and payer accounts. Depending on your needs, you might want to group items based on lines of business or transaction types.

How the Funds Tracking Plugin Allots Funds

BillingCenter calls the `allotAllFunds` method on an `Account` instance whenever BillingCenter determines that the account has funds that need allocation. The `allotAllFunds` method finds `FundsSourceTracker` and `FundsUseTracker` instances on the account that are only partially or not at all allotted. That is, BillingCenter finds `FundsTrackers` where `AmountAllottedDenorm` properties are less than `TotalAmount` properties.

The `Account.allotAllFunds` method then calls `IFundsTracking.allotFunds`. BillingCenter passes two parameters: `Set<FundsSourceTracker>` and `Set<FundsUseTracker>`. `IFundsTracking` delegates to the helper class `FundsAllotter` by calling `FundsAllotter.allotFunds`, passing the two sets of trackers as parameters.

The `FundsAllotter.allotFunds` method first orders the two sets of trackers in oldest to newest order, based on `EventDate`. Then, `FundsAllotter` divides the allotment process into two phases: priority reallocation and allotment.

You can change the order of allotment to be in newest to oldest order. Modify `FundsAllotter` to invoke the `orderDescending` method on the source and use sets in private functions that loop through them. For example:

```
_unallottedSourceTrackers.orderDescending().each(\ sourceTracker ->
    allotFundsFor(sourceTracker))
```

Phase 1: Priority Reallotment

The first phase of an allotment process is priority reallocation. This allows users to undo previous allotments and reallocate those funds elsewhere. In the default configuration, priority reallocation occurs only for source and use trackers associated with reversals. The reversed funds are allotted to the tracker of the reversed entity.

For example, a direct bill money received is a funds source. Initially, plugin allots the funds to a `PaymentItemGroup` to pay a policy premium. Later, the money received transaction is reversed. The reversal is a use of funds, which is tracked by a funds use tracker. The plugin removes the original allotments of the matching original source and use trackers and reallocates them to their reversals.

Phase 2: Allotment

The second phase of an allotment process is allotment. When an allotment phase begins, all available funds trackers have updated amounts. The priority reallocation phase ensures this.

During the allotment phase the `FundsAllotter` loops through the set of funds sources, finding any with remaining funds to allot. It then consumes funds from the source by allotting them to uses that do not have full allotments. `FundsAllotter` continues to match sources and uses until all sources are allotted or there are no more funds uses without full allotments. Any funds sources that are not allotted remain in Account: Unapplied.

BillingCenter protects against over allotment of funds trackers by throwing an exception if code attempts it.

Reversing Allotted Funds

When an allotment of funds is reversed, a new `FundsSourceTracker` tracks the change. Its `TrackableSourceType` is `FundUseReversal`. The new source tracker relates to the original `FundsUseTracker`, the fund use that is reversed. For example, it allots the reversal of a disbursement to the reversed disbursement.

Authentication Integration

To authenticate BillingCenter users, BillingCenter by default uses the user names and passwords stored in the BillingCenter database. Integration developers can optionally authenticate users against a central directory such as a corporate LDAP directory. Alternatively, use single sign-on systems to avoid repeated requests for passwords if BillingCenter is part of a larger collection of web-based applications. Using an external directory requires a *user authentication plugin*.

To authenticate database connections, you might want BillingCenter to connect to an enterprise database but need flexible database authentication. Or you might be concerned about sending passwords as plaintext passwords openly across the network. You can solve these problems with a *database authentication plugin*. This plugin abstracts database authentication so you can implement it however necessary for your company.

This topic discusses *plugins*, which are software modules that BillingCenter calls to perform an action or calculate a result. For information on plugins, see “Plugin Overview” on page 135. For the complete list of all BillingCenter plugins, see “Summary of All BillingCenter Plugins” on page 153.

This topic includes:

- “Overview of User Authentication Interfaces” on page 207
- “User Authentication Source Creator Plugin” on page 209
- “User Authentication Service Plugin” on page 211
- “Deploying User Authentication Plugins” on page 214
- “Database Authentication Plugins” on page 215
- “ContactManager Authentication” on page 216

Overview of User Authentication Interfaces

There are several mechanisms to log in to BillingCenter:

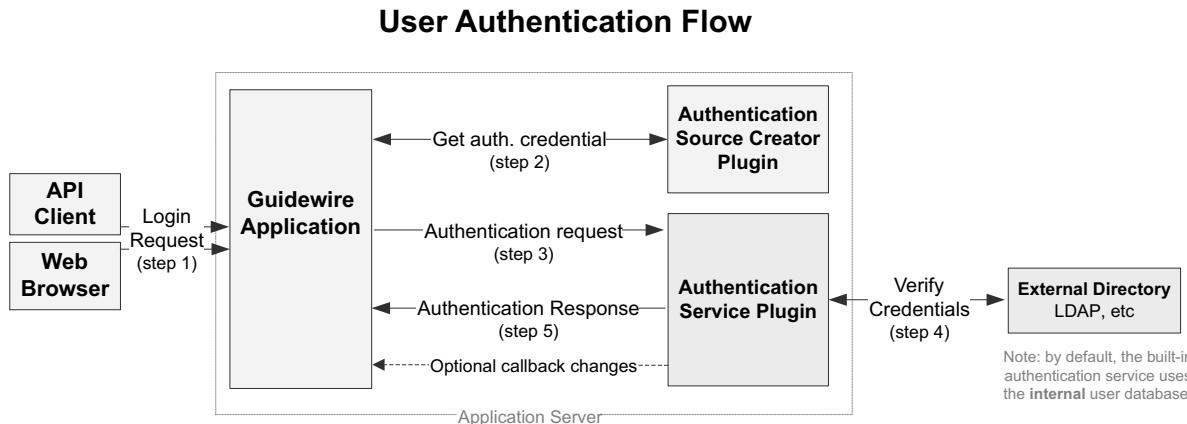
- Log in by using the web application user interface.
- Log in to the server through Guidewire Studio.
- Authenticate WS-I web service calls to the current server.

Authentication plugins must handle all types of logins other than WS-I web services. For authenticating WS-I web services, see the separate plugin interface in “Web Services Authentication Plugin” on page 50.

The authentication of BillingCenter through the user interface is the most complex, and it starts with an initial request from another web application or other HTTP client. To pass authentication parameters to BillingCenter, the HTTP request must submit the username, the password, and any additional properties as *HTTP parameters*. The parameters are name/value pairs submitted within HTTP GET requests as parameters in the URL, or using HTTP POST requests within the HTTP body (not the URL).

Additionally, authentication-related properties can be passed as *attributes*, which are similar to parameters except that they are passed by the servlet container itself, not the requesting HTTP client. For example, suppose Apache Tomcat is the servlet container for BillingCenter. Apache Tomcat can pass authentication-related properties to BillingCenter using attributes that were not on the requesting HTTP client URL from the web browser or other HTTP user agent. Refer to the documentation for your servlet container (such as Apache Tomcat) for details of how to pass attributes to a servlet.

The following diagram gives a conceptual view of the steps that occur during login to BillingCenter:



The chronological flow of user authentication requests is as follows:

1. **An initial login request** – User authentication requests come from web browsers. If the specified user is not currently logged in, BillingCenter attempts to log in the user.
2. **An authentication source creator plugin extracts the request's credentials** – The user authentication information can be initially provided in various ways, such as browser-based form requests or API requests. BillingCenter externalizes the logic for extracting the login information from the initial request and into a structured credential called an *authentication source*. This plugin creates the *authentication source* from information in *HTTPRequests* from browsers and return it to BillingCenter. BillingCenter provides a default implementation that decodes username/password information sent in a web-based form. Exposing this as a plugin allows you to use other forms of authentication credentials such as client certificates or a single sign-on (SSO) credentials. In the reference implementation, the PCF files that handle the login page set the username and password as attributes that the authentication source can extract from the request:


```
String userName = (String) request.getAttribute("username");
String password = (String) request.getAttribute("password");
```
3. **The server requests authentication using an authentication service** – BillingCenter passes the *authentication source* to the *authentication service*. The authentication service is responsible for determining whether or not to permit the user to log in.
4. **The authentication service checks the credentials** – The built-in authentication service checks the provided username and password against information stored in the BillingCenter database. However, a custom implementation of the plugin can check against external authentication directories such as a corporate LDAP directory or other single sign-on system.

5. **Authentication service responds to the request** – The authentication service responds, indicating whether to permit the login attempt. If allowed, BillingCenter sets up the user session and give the user access to the system. If rejected, BillingCenter redirects the user to a login page to try again or return authentication errors to the API client. This response can include connecting to the BillingCenter *callback handler*, which allows the authentication service to search for and update user information as part of the login process. Using the callback handler allows user profile information and user roles to optionally be stored in an external repository and updated each time a user logs in to BillingCenter.

User Authentication Source Creator Plugin

The authentication source creator plugin (`AuthenticationSourceCreatorPlugin`) creates an *authentication source* from an HTTP request. The authentication source is represented by an `AuthenticationSource` object and is typically an encapsulation of username and password. However, it also contains the ability to store a cryptographic hash. The details of how to extract authentication from the request varies based on the web server and your other authentication systems with which BillingCenter must integrate. This plugin is in the `gw.plugin.security` package namespace.

Handling Errors in the Authentication Source Plugin

In typical cases, code in an *authentication source* plugin implementation operates only locally. In contrast, an *authentication service* plugin implementation typically goes across a network and must test authentication credentials with many more possible error conditions.

Try to design your authentication source plugin implementation to not need to throw exceptions.

If you do need to throw exceptions from your authentication source plugin implementation:

- Typically, the login user interface displays a default general message for failure to authenticate and ignores the text in the actual exception.
- The recommended way to display a custom message with an error is to throw the exception class `DisplayableLoginException`:

```
throw new DisplayableLoginException("The application shows this custom message to the user")
```

Optionally you can subclass `DisplayableLoginException` to track specific different errors for logging or other reasons.

- Only if that approach is insufficient for your login exception handling, you can create an entirely custom exception type as follows.
 - a. Create a subclass of `javax.security.auth.login.LoginException` for your authentication source processing exception.
 - b. Create a subclass of `gw.api.util.LoginForm`.
 - c. In your `LoginForm` subclass, override the `getSpecialLoginExceptionMessage(LoginException)` method. BillingCenter only calls this method for exception types that are not built-in. Your version of the method must return the `String` to display to the user for that exception type. Note that the only supported method of `LoginForm` for you to override is `getSpecialLoginExceptionMessage`.
 - d. Modify the `Login.pcf` page, which controls the user interface for that form. That PCF page instantiates `gw.api.util.LoginForm`. Change the PCF page to instantiate your `LoginForm` subclass instead of the default `LoginForm` class.

Authentication Data in HTTP Attributes, Such as the BillingCenter PCF Login Page

In the default of implementation BillingCenter, login-related PCF files set the username and password as HTTP request attributes. HTTP request attributes are hidden values in the request. Do not confuse HTTP attributes with URL parameters. To extract data from the URL itself, see “Authentication Data in Parameters in the URL” on page 210.

The authentication source can extract these attributes from the request in the `HttpServletRequest` object:

```
String userName = (String) request.getAttribute("username");
String password = (String) request.getAttribute("password");
```

This plugin interface provides only one method, which is called `createSourceFromHTTPRequest`. The following example of how to implement this method:

```
public class BasicAuthenticationSourceCreatorPlugin implements AuthenticationSourceCreatorPlugin {
    public void init(String rootDir, String tempDir) {
    }

    public AuthenticationSource createSourceFromHTTPRequest(HttpServletRequest request)
        throws InvalidAuthenticationSourceData {
        AuthenticationSource source;

        // in real code, check for errors and throw InvalidAuthenticationSourceData if errors...
        String userName = (String) request.getAttribute("username");
        String password = (String) request.getAttribute("password");
        source = new UserNamePasswordAuthenticationSource(userName, password);
        return source;
    }
}
```

Authentication Data in Parameters in the URL

If you need to extract parameters from the URL itself, use the `getParameter` method rather than the `getAttribute` method on `HttpServletRequest`.

For example from a URL with the syntax:

```
https://myserver:8080/bc/BillingCenter.do?username=aapplegate&password=sheridan&objectID=12354
```

For example, use the following code:

```
package gw.plugin.security
uses javax.servlet.http.HttpServletRequest
uses com.guidewire.pl.plugin.security.AuthenticationSource
uses com.guidewire.pl.plugin.security.UserNamePasswordAuthenticationSource

class MyAuthenticationSourceCreatorPlugin implements AuthenticationSourceCreatorPlugin {

    override function createSourceFromHTTPRequest(request : HttpServletRequest) : AuthenticationSource {
        var source : AuthenticationSource
        var userName = request.getParameter( "username" )
        var password = request.getParameter( "password" )
        source = new UserNamePasswordAuthenticationSource( userName, password )

        return source
    }
}
```

Authentication Data in HTTP Headers for HTTP Basic Authentication

The `BillingCenter/java-api/examples` directory also contains a simple example implementation of a plugin that gets authentication information from HTTP basic authentication. Basic authentication encodes the data in HTTP headers for some web servers, such as IBM’s WebSeal.

The example implementation turns this information into an Authentication Source if it is sent encoded in the HTTP request header, for example if using IBM’s WebSeal. The plugin decodes a username and password stored in the header and constructs a `UserNamePasswordAuthenticationSource`.

This plugin interface provides only one method, which is called `createSourceFromHTTPRequest`. The following example shows how to implement this method:

```
public class BasicAuthenticationSourceCreatorPlugin implements AuthenticationSourceCreatorPlugin {  
    public void init(String rootDir, String tempDir) {}  
  
    public AuthenticationSource createSourceFromHTTPRequest(HttpServletRequest request)  
        throws InvalidAuthenticationSourceData {  
        AuthenticationSource source;  
        String authString = request.getHeader("Authorization");  
        if (authString != null) {  
            byte[] bytes = authString.substring(6).getBytes();  
            String fullAuth = new String(Base64.decodeBase64(bytes));  
            int colonIndex = fullAuth.indexOf(':');  
            if (colonIndex == -1) {  
                throw new InvalidAuthenticationSourceData("Invalid authorization header format");  
            }  
            String userName = fullAuth.substring(0, colonIndex);  
            String password = fullAuth.substring(colonIndex + 1);  
            if (userName.length() == 0) {  
                throw new InvalidAuthenticationSourceData("Could not find username");  
            }  
            if (password.length() == 0) {  
                throw new InvalidAuthenticationSourceData("Could not find password");  
            }  
            source = new UserNamePasswordAuthenticationSource(userName, password);  
            return source;  
        } else {  
            throw new InvalidAuthenticationSourceData("Could not find authorization header");  
        }  
    }  
}
```

You can implement this authentication source creator interface and store more complex credentials. If you do this, you must also implement an authentication service that knows how to handle these new sources. To do that, implement a user authentication service plugin (`AuthenticationServicePlugin`), described in the next section.

To view the source code to this example, refer to

[BillingCenter/java-api/examples/plugins/authenticationsourcecreator/](#)

User Authentication Service Plugin

An authentication service plugin (`AuthenticationServicePlugin`) implementation defines an external service that could authenticate a user. Typically, this would involve sending authentication credentials encapsulated in an *authentication source* and sending them to some separate centralized server on the network such as an LDAP server. This plugin is in the `gw.plugin.security` package namespace.

There are several mechanisms to log in to BillingCenter:

- Log in by using the web application user interface.
- Log in to the server through Guidewire Studio.
- Authenticate WS-I web service calls to the current server.

Any `AuthenticationServicePlugin` implementation must handle all types of logins other than WS-I web services. For authenticating WS-I web services, see the separate plugin interface in “Web Services Authentication Plugin” on page 50.

For Guidewire Studio users and SOAP API calls, the credentials passed to this plugin are the standard `UserNamePasswordAuthenticationSource`, which contains a basic username and password. In addition, if you design a custom authentication source with data extracted from a web application login request, this plugin must be able to handle those credentials too.

There are three parts of implementing this plugin, each of which is handled by a plugin interface method:

- **Initialization** – All authentication plugins must initialize itself in the plugin’s `init` method.

- **Setting callbacks** – A plugin can look up and modify user information as part of the authentication process using the plugin's `setCallback` method. This method provides the plugin with a call back handler (`CallbackHandler`) in this method. Your plugin must save the callback handler reference in a class variable to use it later to make any changes during authentication.
- **Authentication** – Authentication decisions from a username and password are performed in the `authenticate` method. The logic in this method can be almost anything you want, but typically would consult a central authentication database. The basic example included with the product uses the `CallbackHandler` to check for the user within BillingCenter. The JAAS example calls a JAAS provider to check credentials. Then, it looks up the user's public ID in BillingCenter by username using the `CallbackHandler` to determine which user authenticated.

Every `AuthenticationServicePlugin` must support the default `UserNamePasswordAuthenticationSource` because this source is used by Guidewire Studio if connecting to the BillingCenter server. A custom implementation must also support any other authentication sources that may be created by your custom *authentication source creator plugin*, if any.

Almost every authentication service plugin uses the `CallbackHandler` provided to it, if only to look up the public ID of the user after verifying credentials. Find the Javadoc for this interface in the class `AuthenticationServicePluginCallbackHandler`. This utility class includes four utility methods:

- `findUser` – Lets your code look up a user's public ID based on login user name
- `verifyInternalCredential` – Supports testing a username and password against the values stored in the main user database. This method is used by the default authentication service.
- `modifyUser` – After getting current user data and making changes, perhaps based on contact information stored externally, this method allows the plugin to update the user's information in BillingCenter.

For more details of the method signatures, refer to the Java API Reference Javadoc for `AuthenticationServicePlugin`.

Authentication Service Sample Code

The product includes the following example authentication services in the directory `BillingCenter/java-api/examples/plugins/authenticationservice`.

- **Default BillingCenter authentication example** – This is the basic default service for BillingCenter provided as source code. It checks the username and password against information stored in the BillingCenter database and authenticates the user if a match is found.
- **LDAP authentication example** – `LDAPAuthenticationServicePlugin` is an example of authenticating against an LDAP directory.
- **JAAS authentication example** – `JAASAuthenticationServicePlugin` is an example of how you could write a plugin to authenticate against an external repository using JAAS. JAAS is an API that enables Java code to access authentication services without being tied to those services.

In the JAAS example, the plugin makes a `context.login()` call to the configured JAAS provider. The provider in turn calls back to the `JAASCallbackHandler` to request credential information needed to make a decision. The plugin provides this callback handler to the JAAS provider. This is a JAAS object, different from the callback handler provided by BillingCenter to the plugin as a BillingCenter API. If the user cannot authenticate to JAAS, then the JAAS provider throws an exception. Otherwise, login continues.

Error Handling in Authentication Service Plugins

Your code must be very careful to catch and categorize different types of errors that can happen during authentication. This is important for preserving this information for logging and diagnostic purposes. Additionally, BillingCenter provides built-in features to clarify for the user what went wrong.

In general, the login user interface displays a default general message for failure to authenticate and ignores the text in the actual exception. However, `DisplayableLoginException` allows you to throw a more specific message to show to the user from your authentication service plugin code.

The following table lists exception classes that you can throw from your code for various authentication problems. In general, BillingCenter classes in the `com.guidewire` hierarchy are internal and not for customer use. However, the classes in the `com.guidewire.pl.plugin` hierarchy listed below are supported for customer use.

| Exception name | Description |
|---|--|
| <code>javax.security.auth.login.FailedLoginException</code> | Throw this standard Java exception if the user entered an incorrect password. |
| <code>com.guidewire.pl.plugin.security.InactiveUserException</code> | This user is inactive. |
| <code>com.guidewire.pl.plugin.security.LockedCredentialException</code> | Credential information is inaccessible because it is locked for some reason. For example, if a system is configured to permanently lock a user out after too many attempts. Also see related exception <code>MustWaitToRetryException</code> . |
| <code>com.guidewire.pl.plugin.security.MustWaitToRetryException</code> | The user tried too many times to authenticate and now has to wait for a while before trying again. The user must wait and retry at a later time. This is a temporary condition. Also see related exception <code>LockedCredentialException</code> . |
| <code>com.guidewire.pl.plugin.security.AuthenticationException</code> | Other authentication issues not otherwise specified by other more specific authentication exceptions. |
| <code>com.guidewire.pl.plugin.security.DisplayableLoginException</code> | This is the only authentication exception for which the login user interface uses the text of the actual exception to present to the user. For example, throw the exception as follows: <code>throw new DisplayableLoginException("The application shows this custom message to the user")</code> |

You can make subclasses of the following exception classes:

- If `DisplayableLoginException` does not meet your needs, you can subclass the Java base class `javax.security.auth.login.LoginException`. To ensure the correct text displays for that class:
 - a. Create a subclass of `gw.api.util.LoginForm`.
 - b. In your `LoginForm` subclass, override the `getSpecialLoginExceptionMessage(LoginException)` method. BillingCenter only calls this method for exception types that are not built-in. Your version of the method must return the `String` to display to the user for that exception type. Note that the only supported method of `LoginForm` for you to override is `getSpecialLoginExceptionMessage`.
 - c. Modify the `Login.pcf` page, which controls the user interface for that form. That PCF page instantiates `gw.api.util.LoginForm`. Change the PCF page to instantiate your `LoginForm` subclass instead of the default `LoginForm` class.
- You can also subclass the exception `guidewire.pl.plugin.security.DisplayableLoginException` if necessary for tracking unique types of authentication errors with custom messages.

SOAP API User Permissions and Special-Casing Users

Guidewire recommends creating separate a BillingCenter user (or users) for SOAP API access. This user or set of users must have the minimum permissions allowable to perform SOAP API calls. Guidewire strongly recommends this user have few permissions or no permissions in the web application user interface.

From an authentication service plugin perspective, for those users you could create an exception list in your authentication plugin to implement BillingCenter internal authentication for only those users. For other users, use LDAP or some other authentication service.

Example Authentication Service Authentication

The following sample shows how to verify a user name and password against the BillingCenter database and then modify the user's information as part of the login process.

```
public String authenticate(AuthenticationSource source) throws LoginException {
    if (source instanceof UserNamePasswordAuthenticationSource == false) {
        throw new IllegalArgumentException("Authentication source type " +
            source.getClass().getName() + " is not known to this plugin");
    }

    Assert.checkNotNullParam(_handler, "Callback handler not set");

    UserNamePasswordAuthenticationSource uNameSource = (UserNamePasswordAuthenticationSource) source;

    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "LDAP://" + _serverName + ":" + _serverPort);
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    String userName = uNameSource.getUsername();
    if (StringUtils.isNotBlank(_domainName)) {
        userName = _domainName + "\\" + userName;
    }

    env.put(Context.SECURITY_PRINCIPAL, userName);
    env.put(Context.SECURITY_CREDENTIALS, uNameSource.getPassword());

    try {
        // Try to login.
        new InitialDirContext(env);
        // Here would could get the result to the earlier and
        // modify the user in some way if you needed to
    } catch (NamingException e) {
        throw new LoginException(e.getMessage());
    }

    String username = uNameSource.getUsername();
    String userPublicId = _handler.findUser(username);
    if (userPublicId == null) {
        throw new FailedLoginException("Bad user name " + username);
    }
}

return userPublicId;
```

Deploying User Authentication Plugins

Like other plugins, there are two steps to deploying custom authentication plugins:

1. Move your code to the proper directory.
2. Register your plugins in the Studio plugin editor.

First, move your code to the proper directory. Place your custom `AuthenticationSourceCreator` plugin implementation in the appropriate subdirectory of `BillingCenter/modules/configuration/plugins/authenticationsourcecreator/basic`, referred to later in this paragraph as `AUTHSOURCEROOT`. For example, if your class is `custom.authsource.MyAuthSource`, move the location to `AUTHSOURCEROOT/classes/custom/authsource/MyAuthSource.class`. If your code depends on any Java libraries other than the BillingCenter generated libraries, place the libraries in `AUTHSOURCEROOT/lib/`.

Similarly, place your AuthenticationService plugin in the appropriate subdirectory of BillingCenter/modules/configuration/plugins/authenticationservice/basic, referred to later in this paragraph as *AUTHSERVICEROOT*. For example, if your class is custom.authservice.MyAuthService, move the file to the location *AUTHSERVICEROOT/classes/basic/custom/authservice/MyAuthService.class*. If your code depends on any Java libraries other than BillingCenter generated libraries, place the libraries in *AUTHSERVICEROOT/lib/*.

For BillingCenter to find and use your custom plugins, you must register them in the plugin editor in Studio. See “Messaging Editor” on page 131 in the *Configuration Guide*. Remember that in the editor, the plugin name and the plugin interface name must match, even though Studio permits you to enter a different value for the plugin name. Otherwise, BillingCenter does not recognize the plugin. The value for both must be the name of the plugin interface you are trying to use.

In the main config.xml file but not the plugin registry there is a sessiontimeoutsecs parameter that configures the duration of inactivity in seconds to allow before requiring reauthentication. This timeout period controls both the user’s web (HTTP) session and the user’s session within the BillingCenter application. Users who connect to BillingCenter using the API, for example, do not have an HTTP session, only an application session.

The following is an example of this inactive session timeout parameter:

```
<param name="SessionTimeoutSecs" value="10800"/>
```

Database Authentication Plugins

You might want the BillingCenter server to connect to an Enterprise database, but require a flexible database authentication system. Or, you might be concerned about sending passwords as plaintext passwords openly across the network. Solve either of these problems by implementing a *database authentication plugin*.

A custom database authentication plugin can retrieve name and password information from an external system, encrypt passwords, read password files from the local file system, or any other desired action. The resulting username and password substitutes into the database configuration file anywhere that \${username} or \${password} are found in the database parameter elements.

It is important to understand that *database authentication plugins* are different from *user authentication plugins*. Whereas user authentication plugins authenticate users into BillingCenter (from the user interface or using API), database authentication plugins help the BillingCenter server connect to its database server.

To implement a database authentication plugin, implement a plugin that implements the class DBAuthenticationPlugin, which is defined in the Java package com.guidewire.p1.plugin.dbauth.

This class has only one method you need to implement: retrieveUsernameAndPassword, which must return a username and password. Store the username and password combined together as properties within a single instance of the class UsernamePasswordPair.

The one method parameter for retrieveUsernameAndPassword is the name of the database (as a String) for which the application requests authentication information. This will match the value of the name attribute on the database or archive elements in your config.xml file.

If you need to pass additional optional properties such as properties that vary by server ID, pass parameters to the plugin in the Studio configuration of your plugin. Get these parameters in your plugin implementation using the standard setParameters method of InitializablePlugin. For more information, see “Example Gosu Plugin” on page 141.

The username and password that this method returns need not be a plaintext username and password, and it typically would **not** be plaintext. A plugin like this typically encodes, encrypts, hashes, or otherwise converts the data into a secret format. The only requirement is that your database (or an intermediate proxy server that pretends to be your database) knows how to authenticate against this username and password.

The following example demonstrates this method by pulling this information from a file:

```
public class FileDBAuthPlugin implements DBAuthenticationPlugin, InitializablePlugin {
```

```

private static final String PASSWORD_FILE_PROPERTY = "passwordfile";
private static final String USERNAME_FILE_PROPERTY = "usernamefile";

private String _passwordfile;
private String _usernamefile;

public void setParameters(Map properties) {
    _passwordfile = (String) properties.get(PASSWORD_FILE_PROPERTY);
    _usernamefile = (String) properties.get(USERNAME_FILE_PROPERTY);
}

public UsernamePasswordPair retrieveUsernameAndPassword(String dbName) {
    try {
        String password = null;
        if (_passwordfile != null) {
            password = readLine(new File(_passwordfile));
        }
        String username = null;
        if (_usernamefile != null) {
            username = readLine(new File(_usernamefile));
        }
        return new UsernamePasswordPair(username, password);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

private static String readLine(File file) throws IOException {
    BufferedReader reader = new BufferedReader(FileUtil.getFileReader(file));
    String line = reader.readLine();
    reader.close();
    return line;
}
}

```

Guidewire BillingCenter includes an example database authentication plugin that simply reads a username and password from files specified in the `usernamefile` or `passwordfile` parameters that you define in Studio.

BillingCenter replace the `${username}` and `${password}` values in the `jdbcURL` parameter with values returned by your plugin implementation. For this example, the values to use are the text of the two files (one for username, one for password).

For the source code, refer to the `FileDBAuthPlugin` sample code in the `examples.plugins.dbauthentication` package.

Configuration for Database Authentication Plugins

For BillingCenter to find and use your custom plugins, you must register them in the plugin editor in Studio. See “[Messaging Editor](#)” on page 131 in the *Configuration Guide*. Remember that in the editor, the plugin name and the plugin interface name must match, even though Studio permits you to enter a different value for the plugin name. Otherwise, BillingCenter does not recognize the plugin. The value for both must be the name of the plugin interface you are trying to use. Add parameters as appropriate to pass information to your plugin.

BillingCenter also supports looking up database passwords in a password file by setting “`passwordfile`” as a `<database>` attribute in your main `config.xml` file.

At run time, the username and password returned by your database authentication plugin replaces the `${username}` and `${password}` parts of your database initialization `String` values.

ContactManager Authentication

For more information about ContactManager authentication, see “[Configuring ContactManager Authentication with Core Applications](#)” on page 74 in the *Contact Management Guide*.

Document Management

BillingCenter provides a user interface and integration APIs for creating documents, downloading documents, and producing automated form letters. You can integrate BillingCenter with a separate external document management system (DMS) that stores the documents. Optionally, the external DMS can store the document metadata, such as the list of documents and their file types.

This topic includes:

- “Document Management Overview” on page 217
- “Choices for Storing Document Content and Metadata” on page 219
- “Document Storage Plugin Architecture” on page 221
- “Implementing a Document Content Source for External DMS” on page 222
- “Storing Document Metadata In an External DMS” on page 225
- “The Built-in Document Storage Plugins” on page 226
- “Asynchronous Document Storage” on page 227
- “APIs to Attach Documents to Business Objects” on page 228
- “Retrieval and Rendering of PDF or Other Input Stream Data” on page 229

See also

- For general information on plugins, which are an important part of document management in BillingCenter, see “Plugin Overview” on page 135.
- “Document Production” on page 231

Document Management Overview

The BillingCenter user interface provides a **Documents** section within the policy and account file.

The **Documents** section lists documents such as letters, faxes, emails, and other attachments stored in a document management system (DMS).

Document Storage Overview

BillingCenter can store existing documents in a document management system. For example, you can attach outgoing notification emails, letters, or faxes created by business rules to an insured customer. You can also attach incoming electronic images or scans of paper witness reports, photographs, scans of police reports, or signatures. You can optionally integrate BillingCenter with an external document management system (DMS).

Within the BillingCenter user interface, you can find documents attached to a business object, and view the documents. You can also search the set of all stored documents.

Transfer of large documents through the application server to an external document storage system requires significant memory and resources. Even in the best case scenario of memory and CPU resources, the external document storage system (or intermediate network) may be slow. If so, synchronous actions with large documents may appear unresponsive to a BillingCenter web user. To address these issues, BillingCenter provides a way to asynchronously send documents to the document management system without bringing documents into application server memory. For maximum user interface responsiveness with an external document storage system, choose asynchronous document storage. In the default configuration, asynchronous document storage is enabled. For more information, refer to “Asynchronous Document Storage” on page 227.

It is important to understand the various types of IDs for a Document entity instance:

- **Document.PublicID** – the unique identifier for a single document in the DMS. If the DMS supports versions, the PublicID property does not change for each new version.
- **Document.DocUID** – the unique identifier for a single revision of a single document in the DMS. If the DMS supports versions, the DocUID property might change for each new version, depending on how your DMS works. See also the reference to extension properties later in this bullet list.
- **extension properties** – You can extend the base Document entity to contain version-related properties or other properties if it makes sense for your DMS.
- **Document.Id** – *This is an internal BillingCenter property.* Never use the Id property to identify a document. Never get or set this property for document management.

Document Production Overview

BillingCenter can create new documents from forms, form letters, or other structured data. For example, notification emails, letters, or faxes created by business rules to an insured customer. Typically you would persist the document and attach the resulting new document to a business data object.

There are two ways to create documents:

- BillingCenter can facilitate creation of documents on the user’s desktop using local application such as Microsoft Word or Excel. This is *client-side document production*.
- BillingCenter can create some types of new documents from a server-stored template without user intervention. This is *server-side document production*.

For more information, see “Document Production” on page 231.

Document Retrieval Mode Overview

There are several document retrieval modes, also known as response types:

- **URL** – A URL to a local content store to display the content. The URL response type is the recommended response type if your DMS supports it. The URL response type permits the highest performance for BillingCenter.
- **DOCUMENT_CONTENTS** – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 229

For more information, see “Retrieving Documents” on page 223.

Choices for Storing Document Content and Metadata

There are two separate kinds of data that a document storage system (DMS) processes for BillingCenter. Before implementing a document management integration, you must decide where to store document content and metadata. The following table compares and contrasts these kinds of data.

| Type of data | Description | Plugin interface that handles this data | Default behaviors |
|-------------------|--|---|---|
| Document content | <p>Document content can be anything that represents incoming or outgoing documents.</p> <p>For example:</p> <ul style="list-style-type: none"> • a fax image • a Microsoft Word file • a photograph • a PDF file <p>A <i>document content source</i> is code that stores and retrieves the document content.</p> | IDocumentContentSource | <p>It is important to note that for the highest performance and data integrity, use an external DMS.</p> <p>In the initial BillingCenter configuration, built-in classes implement asynchronous storage onto the local file system. For more information, see “The Built-in Document Storage Plugins” on page 226 and “Asynchronous Document Storage” on page 227. If you use an external DMS, you would not use local file system storage.</p> |
| Document metadata | <p>Document metadata describes the file:</p> <ul style="list-style-type: none"> • name, which typically is the file name • MIME type • description • the full set of metadata on the DMS so users can search for documents. • which business objects are associated with each document • other fields defined by the application or customer extensions, such as recipient, status, or security type <p>A <i>document metadata source</i> is code that manages and searches document metadata.</p> | IDocumentMetadataSource | <p>In the default BillingCenter configuration, this plugin is disabled. In other words, in the Studio user interface for this plugin interface, the Enabled checkbox is unchecked.</p> <p>If this plugin is disabled, BillingCenter stores the document metadata internally in the database, with one Document entity instance for each document. If you store the metadata internally in this way, searching for documents is very fast. Many DMS systems are not designed to withstand high capacity for external searching and metadata lookup. Additionally, if you use internal metadata, the metadata is available even if the DMS is offline.</p> <p>There is a built-in plugin implementation called <code>LocalDocumentMetadataSource</code>. This is for testing only and is unsupported in production. To correctly configure fast internal document metadata, uncheck the Enabled checkbox for this plugin.</p> |

Deciding Where to Store Document Content and Metadata

The best approach for configuring these plugins depends on whether you want to use an external DMS and how your company uses documents. For the highest performance and data integrity, use an external DMS.

To configure BillingCenter for internal document storage (no external DMS):

1. For document content, use the default implementations of the `IDocumentContentSource` plugin. For more information, see “The Built-in Document Storage Plugins” on page 226 and “Asynchronous Document Storage” on page 227.

2. For document metadata, do not enable the `IDocumentMetadataSource` plugin. If this plugin is disabled, BillingCenter stores the document metadata internally in the database, with one `Document` entity instance for each document. Internal metadata storage makes document search fast. Additionally, if you use internal metadata, you can search document metadata even when the DMS is offline.

WARNING Choose your document metadata location carefully. See “Internal Versus External Metadata Permanently Affects Some Objects” on page 220.

To configure BillingCenter for an external DMS:

1. For the document content, write your own implementation of the `IDocumentContentSource` plugin to store and retrieve content. Your plugin implementation stores and retrieves files using the proprietary API for your external DMS. If you use an external DMS, you can still use the built-in asynchronous document storage system. See “Asynchronous Document Storage” on page 227.
2. For the document metadata, the best solution depends on how you use documents:

WARNING Choose your document metadata location carefully. See “Internal Versus External Metadata Permanently Affects Some Objects” on page 220.

- If most documents in the DMS related to BillingCenter exist because users upload or create documents in BillingCenter, use internal metadata storage. Do not enable the `IDocumentMetadataSource` plugin in Studio. If this plugin is disabled, BillingCenter stores the document metadata internally in the database, with one `Document` entity instance for each document. Internal metadata storage using the database makes document search very fast. Additionally, if you use internal metadata, the metadata is available when the DMS is offline.
- If most documents in the DMS related to BillingCenter are added directly into the DMS without BillingCenter, write your own implementation of the `IDocumentMetadataSource` plugin. Your plugin implementation manages and searches the metadata. If you want to share document metadata searching across multiple Guidewire applications, it is best to use external metadata.

Internal Versus External Metadata Permanently Affects Some Objects

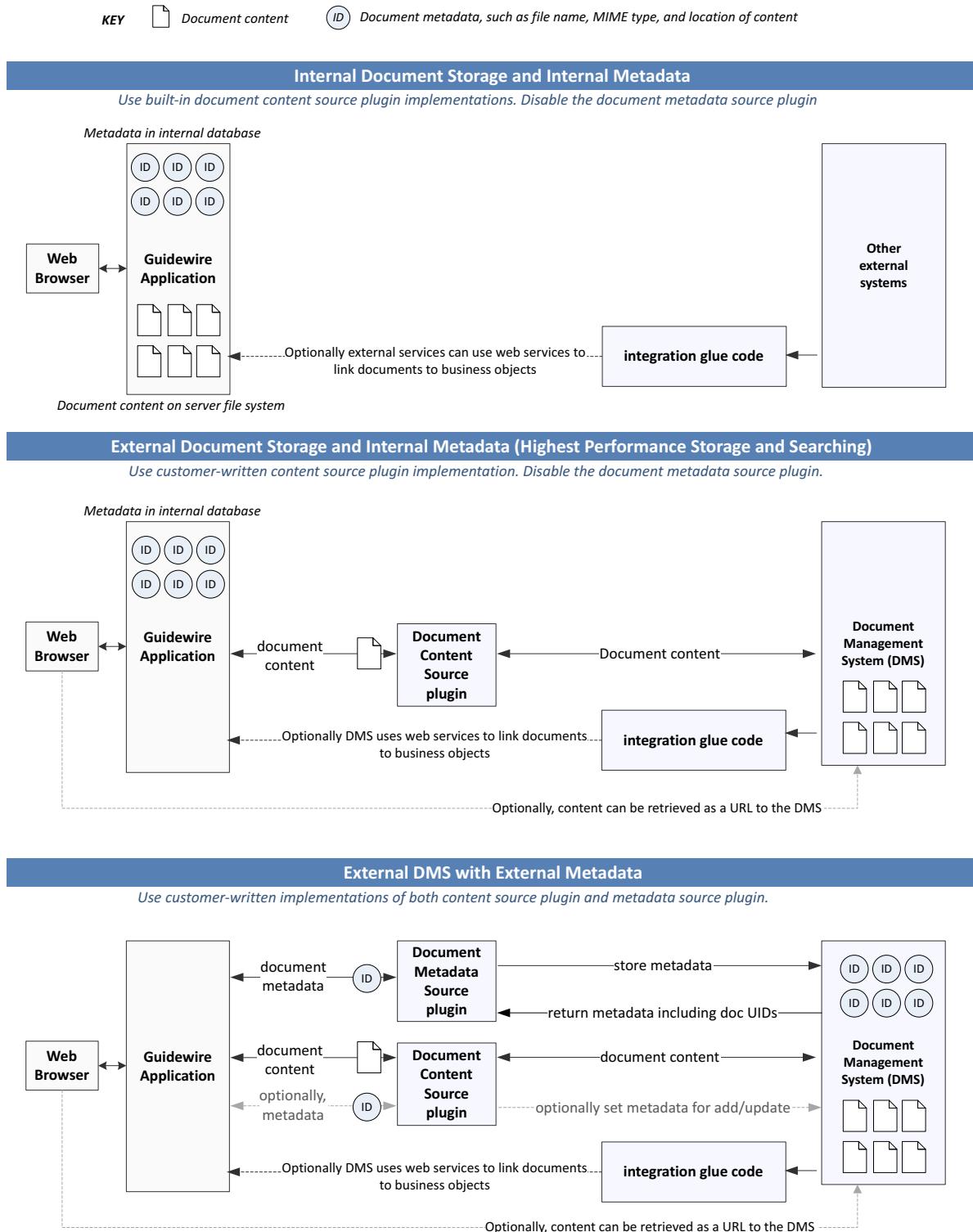
Whichever approach to handling document metadata you choose (internal or external), be aware that you must continue that approach permanently. Note objects and other similar text blocks reference a document within text as a specially-formatted hyperlink within the text itself. The format of this hyperlink within text blocks is slightly different based on whether BillingCenter or a remote system handles the document metadata. If this quality changes from internal to external, or from external to internal, those document hyperlinks fail.

If you are considering this type of transition, contact Guidewire Customer Support for advice before proceeding.

Document Storage Plugin Architecture

The following diagram summarizes implementation architecture options for document management.

Document Storage Architecture Options



Implementing a Document Content Source for External DMS

Document storage systems vary in how they transfer documents and how they display documents in the user interface. To support this variety, BillingCenter supports multiple document retrieval modes called *response types*. For the list of options, see “Document Retrieval Mode Overview” on page 218.

To implement a new document content source, you must write a class that implements the `IDocumentContentSource` plugin interface. The following sections describe the methods that your class must implement.

Adding Documents and Metadata

A new document content source plugin must fulfill a request to add a document by implementing the `addDocument` method. This method adds a new document object to the repository.

IMPORTANT If you enable and implement the `IDocumentDataSource` plugin, BillingCenter also calls this method to update an existing document. Write your code to accommodate being called with an existing document.

The method takes as arguments:

- the document metadata in a `Document` object
- the document contents as an `InputStream` object

The `addDocument` method also may copy some of the metadata contained in the `Document` entity into the external system. Independent of whether the document stores any metadata in the external system, the plugin must update certain metadata properties within the `Document` object:

- the method must set an implementation-specific document universal ID in the `Document.DocUID` property.
- the method must set the modified date in the `Document.DateModified` property.

The method must persist these changes to the BillingCenter database.

The return value from the `addDocument` method is very important:

- If you do not enable and implement the `IDocumentDataSource` plugin, you must return `false`.
- If you enable and implement the `IDocumentDataSource` plugin implementation:
 - If you stored metadata from the `Document` object in addition to document content, return `true`.
 - Otherwise, return `false`. BillingCenter calls the registered `IDocumentDataSource` plugin implementation to store the metadata.

For related discussion, see “Deciding Where to Store Document Content and Metadata” on page 219.

Error Handling

If your plugin code encounters errors during storage before returning from this method, throw an exception from this method. If document storage errors are detected later, such as for a asynchronous document storage, the document content storage plugin must handle errors in an appropriate fashion. For example, send administrative e-mails or create new activities to investigate the problem. You could optionally persist the error information and design a system to track document creation errors and document management problems in a separate user interface for administrators.

Retrieving Documents

A new document content source plugin must fulfill a request to retrieve a document. To fulfill this kind of request, implement the `getDocumentContentsInfo` method. This method takes a `Document` object as a parameter. Use the `Document.DocUID` property to identify the target document within the external repository. Your code could use additional data model extensions on `Document` as needed.

The `getDocumentContentsInfo` method can read the `Document` object but must not modify the `Document` object.

This plugin method must return an instance of `gw.document.DocumentContentsInfo`, which is a simple object with the following properties:

- **Response type** – An enumeration in the `ResponseType` property indicates the type of the data:
 - URL – A URL to a local content store to display the content. For non-editable documents, the URL response type is the recommended response type if your DMS supports it. The URL response type permits the highest performance for BillingCenter. However, the URL response type is problematic for editing documents, and would require modification of PCF code to accommodate it.
 - DOCUMENT_CONTENTS – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 229
- **Hidden target frame as Boolean** – For response types other than DOCUMENT_CONTENTS, the Boolean property `TargetHiddenFrame` specifies whether BillingCenter opens the web page in a hidden frame within the user’s web browser. For example, if the document repository is naturally implemented as a small web page that contains JavaScript, set this property to `true`. JavaScript code from that page might open another web page as a popup window that displays the real document contents.
- **Contents as input stream** – The `InputStream` property contains the document contents in the form of an input stream, which is raw bytes as an `InputStream` object.

The `includeContents` Boolean Parameter

The `getDocumentContentsInfo` method has a Boolean parameter `includeContents`.

- If this parameter is `true`, include an input stream with the contents of the document. Set only the properties, `ResponseType`, `TargetHiddenFrame`, `InputStream`.
- If this parameter is `false`, only set the response type in the property `ResponseType`. Do not set any other properties.

BillingCenter calls some methods in a `IDocumentContentSource` plugin twice for each document: `isDocument` and `getDocumentContentsInfo`. Your plugin implementation must support this design. For the `getDocumentContentsInfo` method, the `includeContents` parameter is set to `false` on the first call and `true` on the second call. Your plugin must return a `DocumentContentsInfo` object with the same response type in response to both calls. Returning any other result is unsupported and results in undefined behavior.

MIME Type

The `DocumentContentsInfo.ResponseMimeType` property includes a MIME type. However, your `getDocumentContentsInfo` method must ignore this property. BillingCenter sets the MIME type based on the `Document` properties after calling the `getDocumentContentsInfo` method. However, it is unsupported to rely on its value at the time that BillingCenter calls `getDocumentContentsInfo`.

Checking for Document Existence

A document content source plugin must fulfill requests to check for the exist of a document. To check document existence, BillingCenter calls the plugin implementation’s `isDocument` method. This method takes a `Document` object as a parameter. Use the `Document.DocUID` property to identify the target document within the repository. If the document with that document UID exists in the external DMS, return `true`. Otherwise return `false`.

The `isDocument` method must not modify the `Document` entity instance in any way.

BillingCenter calls some methods in a `IDocumentContentSource` plugin twice for each document: `isDocument` and `getDocumentContentsInfo`. Your plugin implementation must support this design.

Removing Documents

A new document content source plugin must fulfill a request to remove a document. To remove a document, BillingCenter calls the plugin implementation's `removeDocument` method. This method takes a `Document` object as a parameter. Use the `Document.DocUID` property to identify the target document within the repository.

BillingCenter calls this method to notify the document storage system that the document corresponding to the `Document` object will soon be removed from metadata storage. Your `IDocumentContentSource` plugin implementation can decide how to handle this notification. Choose carefully whether to delete the content, retire it, archive it, or another action that is appropriate for your company. Other `Document` entities may still refer to the same content with the same `Document.DocUID` value, so deletion may be inappropriate.

Be careful writing your document removal code. If the `removeDocument` implementation does not handle metadata storage, then a server problem might cause removal of the metadata to fail even after successful removing document contents.

If the `removeDocument` method removed document metadata from metadata storage and no further action is required by the application, return `true`. Otherwise, return `false`. If you do not enable and implement the `IDocumentMetadataSource` plugin, you must return `false`.

Updating Documents and Metadata

A new document content source plugin must fulfill requests from BillingCenter to update documents. To update documents, implement the `updateDocument` method. This method takes two arguments:

- a replacement document, as a stream of bytes in an `InputStream` object.
- a `Document` object, which contains document metadata. The `Document.DocUID` property identifies the document in the DMS.

At a minimum, the DMS system must update any core properties in the DMS that represent the change itself, such as the date modified, the update time, and the update user. If the document UID property implicitly changed because of the update, set the `DocUID` property on the `Document` argument. BillingCenter persists changes to the `Document` object to the database.

Optionally, your document content source plugin can update other metadata from `Document` properties. The DMS must update the same set of properties as in your document content source plugin `addDocument` method. If your DMS has a concept of versions, you can extend the base `Document` entity to contain a property for the version. If you extend the `Document` entity with version information, `updateDocument` method sets this information as appropriate.

IMPORTANT During document update, you can optionally set `Document.DocUID` and any extension properties that represent versions. However, do not change the `Document.PublicID` property. On a related note, never get or set the `Document.ID` property. See “Document Storage Overview” on page 218 for related discussion.

The return value from the `updateDocument` method is very important:

- If you do not enable and implement the `IDocumentMetadataSource` plugin, you must return `false`.
- If you enable and implement the `IDocumentMetadataSource` plugin implementation:
 - If you stored metadata from the `Document` object in addition to document content, return `true`.
 - Otherwise, return `false`. BillingCenter calls the registered `IDocumentMetadataSource` plugin implementation to store the metadata.

For related discussion, see “Deciding Where to Store Document Content and Metadata” on page 219.

Storing Document Metadata In an External DMS

In the default BillingCenter configuration, the `IDocumentMetadataSource` plugin is disabled. In other words, in the Studio user interface for this plugin interface, the `t` checkbox is unchecked. If this plugin is disabled, BillingCenter stores the document metadata locally in the database, with one `Document` entity instance for each document. If you store the metadata locally in this way, searching for documents is very fast. Many DMS systems are not designed for high capacity for external searching and metadata lookup. Additionally, if you use local metadata, the metadata is available even if the DMS is offline.

Optionally you can store document metadata in the document source content plugin so the content plugin handles the content and the metadata.

IMPORTANT It is critical to carefully read “Choices for Storing Document Content and Metadata” on page 219 before beginning implementation.

If you are sure you want to store document metadata in the DMS, write a class that implements the document metadata source (`IDocumentMetadataSource`) plugin interface. Typically this plugin sends the metadata to the same external system as the document content source plugin, but it does not have to do so.

It is important to understand the various of IDs for a `Document` entity instance:

- `Document.PublicID` – the unique identifier for a single document in the DMS. If the DMS supports versions, the `PublicID` property does not change for each new version.
- `Document.DocUID` – the unique identifier for a single revision of a single document in the DMS. If the DMS supports versions, the `DocUID` property might change for each new version, depending on how your DMS works. See also the reference to extension properties later in this bullet list.
- **extension properties** – You can extend the base `Document` entity to contain version-related properties or other properties if it makes sense for your DMS.
- `Document.Id` – *This is an internal BillingCenter property.* Your implementation of the document metadata source plugin must never track documents by the ID (`Document.Id`) property.

IMPORTANT Always track a document by using the `PublicID` property, not the `Id` property.

The required plugin methods include:

- `saveDocument` – Persist document metadata in a `Document` object to the document repository. If document content source plugin methods `addDocument` or `updateDocument` return false to indicate they did not handle metadata, BillingCenter calls the document metadata source plugin `saveDocument` method.
- `retrieveDocument` – Returns a completely initialized `Document` object with the latest information from the repository.
- `searchDocuments` – Return the set of documents in the repository that match the given set of criteria. This method’s parameters include a `RemotableSearchResultSpec` entity instance. This object contains sorting and paging information for the PCF list view infrastructure to specify results.

IMPORTANT When a user searches for documents, BillingCenter calls the `searchDocuments` plugin method twice. The first invocation retrieves the number of results. The second invocation gets the results.

- `removeDocument` – Remove a document metadata in a `Document` object from the document repository.

See also

- For more details, refer to the API Reference Javadoc documentation for `IDocumentMetadataSource`.
- For more information about the document source content plugin, see “Implementing a Document Content Source for External DMS” on page 222.
- For more information about the included reference implementation, see “The Built-in Document Storage Plugins” on page 226.

The Built-in Document Storage Plugins

BillingCenter includes a reference implementation for the `IDocumentContentSource` plugin. This plugin implementation simply stores documents on the BillingCenter server’s file system. By default, BillingCenter stores the document metadata such as document names in the database. Document names are what people typically think of as document file names. You can override this behavior by implementing the `IDocumentMetadataSource` plugin.

To decide whether to use the built-in document storage plugin for your project, carefully read these topics:

- “Choices for Storing Document Content and Metadata” on page 219
- “Document Storage Plugin Architecture” on page 221

Built-in Document Storage Directory and File Name Patterns

The document content and storage plugins use the `documents.path` parameter in their XML definition files to determine the storage location. If the value of that parameter is an absolute path, then the specific location is used. If the value is a relative path, then the location is determined by using the value of the temporary directory parameter (`javax.servlet.context.tempdir`) is used instead.

The temporary directory property is the root directory of the servlet container’s temp directory structure. In other words, this is a directory that servlets can write to as scratch space but with no guarantee that files persist from one session to another. The temporary directory of the built-in plugins in general are for testing only. Do not rely on temporary directory data in a production (final) BillingCenter system, and even the built-in plugins are for demonstration only, as indicated earlier.

Documents are partitioned into subdirectories by account. The following table explains this directory structure:

| Entity | Directory naming rules | Example |
|----------|---|--|
| Account | <code>Account + accountPublicID</code> | <code>/documents/AccountABC:123/</code> |
| Policy | <code>Policy" + policyPublicID</code> | <code>/documents/PolicyABC:456/</code> |
| Producer | <code>Producer" + producerPublicID</code> | <code>/documents/ProducerABC:789/</code> |

Notice that for all entities, the only ID that defines the location is the public ID.

BillingCenter stores documents by the name you choose in the user interface as you save and add the file. If you add a file from the user interface and it would result in a duplicate file name, BillingCenter warns you. The new file does not quietly overwrite the original file. If you create a document using business rules, the system forces uniqueness of the document name. The server appends an incrementing number in parentheses. For example, if the file name is `myfilename`, the duplicates are called `myfilename(2)`, `myfilename(3)`, and so on.

The built-in document metadata source plugin provides a unique document ID back to BillingCenter. That document ID identifies the file name and relative path within the repository to the document.

For example, an account document’s repository-relative path might look something like this:

`/documents/accountABC:123/myFile.doc`

Remember to Store Public IDs in the External System

In addition to the unique document IDs, remember to store the `PublicID` property for Guidewire entities such as `Document` in external document management systems.

BillingCenter uses public IDs to refer to objects if you later search for the entities or re-populate entities during search or retrieval. If the public ID properties are missing on document entities during search or retrieval, the BillingCenter user interface may have undefined behavior.

Asynchronous Document Storage

Some document production systems generate documents slowly. When many users try to generate documents at the same time, multiple CPU threads compete and that makes the process even slower. One alternative is to create documents asynchronously so that user interaction with the application does not block waiting for document production.

Similarly, transfer of large documents through the application server to an external document storage system requires significant memory and resources. Even in the best case scenario of memory and CPU resources, the external document storage system (or intermediate network) may be slow. If the system is slow, synchronous actions with large documents may appear unresponsive to a BillingCenter user. To address these issues, BillingCenter provides a way to asynchronously send documents to the document management system without bringing documents into application server memory. A separate thread on the batch server sends documents to your real document management system using the messaging system.

To implement this, BillingCenter includes two software components:

- **Asynchronous document content source** – BillingCenter includes a document content source plugin implementation `gw.plugin.document.impl.AsyncDocumentContentSource`. When it gets a request to send the document to the document management system, it immediately saves the file to a temporary directory on the local disk. In a clustered environment, you can map the local path to this temporary directory so that it references a server directory shared by all elements of the cluster. For example, map this to an SMB server.
- **Document storage messaging transport** – A separate part of the system uses BillingCenter messaging to send documents to the external system. In a clustered environment, any server can create (submit) a new message. However, only the batch server runs the messaging plugins to actually send messages across the network. In the built-in configuration, the transport calls the document storage plugin implementation that you implement. As a result, in a clustered environment, your document content source implementation only runs on the batch server.

By default, the asynchronous document storage plugin is enabled. By default, it simply uses the built-in document content storage implementation. To write your own document content source plugin and configure it for use with asynchronous sending, see “Configuring Asynchronous Document Storage” on page 228.

For maximum document data integrity and document management features, use a complete commercial document management system (DMS) and implement new `IDocumentMetadataSource` and `IDocumentContentSource` plugins to communicate with the DMS. Where at all possible, store the metadata in the DMS if it can natively store this information. In all cases, store the metadata in only one location: either in the DMS or BillingCenter built-in metadata storage, but not both. Avoid duplicating metadata in the BillingCenter database itself for production systems.

Configuring Asynchronous Document Storage

To configure asynchronous sending with your own content source plugin

1. Write a content source plugin as described earlier in this topic. Your plugin implementation must implement the interface `IDocumentContentSource` and must send the document synchronously. However, do not register it directly as the plugin implementation for the `IDocumentContentSource` plugin interface in the Plugins Editor in Studio.
2. In the Studio Project window, navigate to `Configuration → config → Plugins → registry`, and then open `IDocumentContentSource.gwp`. Studio displays the Plugins Registry editor for the default implementation of this plugin. In the default configuration, the Gosu class field has the value `gw.plugin.document.impl.AsyncDocumentContentSource`. Do not change that field.
3. In the `Parameters` table, find the `SynchedContentSource` parameter. Set it to the fully-qualified class name of the class that you wrote that implements `IDocumentContentSource`.
4. In the `Parameters` table, find the `documents.path` parameter. Set it to the local file path for temporary storage of documents waiting to be sent to document storage. In a clustered environment, map the local path to this temporary directory so that it references a server directory shared by all elements of the cluster. For example, map this to an SMB server.

To temporarily disable asynchronous sending

1. In the Studio Project window, navigate to `Configuration → config → Plugins → registry`, and then open `IDocumentContentSource.gwp`.
2. In the `Parameters` table, find the `TrySynchedAddFirst` parameter.
3. Set this parameter to `false`.

To permanently disable asynchronous sending

1. In the Studio Project window, navigate to `Configuration → config → Plugins → registry`, and then open `IDocumentContentSource.gwp`.
2. Set the class name to the fully-qualified class name of your own `IDocumentContentSource` implementation. If your class was not implemented in Gosu, click the red minus sign to remove the Gosu plugin configuration. Click the green plus sign and choose either Java or OSGi.
3. Disable the messaging destination:
 - a. In the Studio Project window, navigate to `Configuration → config → Messaging`, and then open `messaging-config.xml`.
 - b. In the table, click the row for the messaging destination with name `DocumentStore`.
 - c. Clear the `Enabled` checkbox.

APIs to Attach Documents to Business Objects

Gosu APIs to Attach Documents to Business Objects

BillingCenter provides document-related Gosu APIs. Your business rules can add new documents to certain types of objects (including a account) using document-related Gosu APIs.

To create a new `Document` object, you can use methods that appear on `Account`, `Policy`, and `Producer`. All those entity types implement the interface `DocumentContainer`, which includes the method `addDocument`.

To create a document on an account, policy, or producer

1. From Gosu or Java, call the `addDocument` method on a `Account`, `Policy`, or `Producer` object:

```
myNewDocument = Account.addDocument()
```

2. Then, set additional the properties on the new Document as needed.

Web Service APIs to Attach Documents to Business Objects

BillingCenter provides a web services API for adding linking business objects (such as a account) to a document. This allows external process and document management systems to work together to inform BillingCenter after creation of new documents related to a business object. For example, in paperless operations, new postal mail might come into a scanning center to be scanned. Some system scans the paper, identifies with a business object, and then loads it into an electronic repository.

The repository can notify BillingCenter of the new document and create a new BillingCenter activity to review the new document. Similarly, after sending outbound correspondence such as an e-mail or fax, BillingCenter can add a reference to the new document. After the external document repository saves the document and assigns an identifier to retrieve it if needed, BillingCenter stores that document ID.

The BCAPI web service includes several methods to add documents to BillingCenter objects:

- `addDocumentToAccount` – Add a new Document to an existing Account
- `addDocumentToPolicyPeriod` – Add a new Document to an existing PolicyPeriod
- `addDocumentToProducer` – Add a new Document to an existing Producer

Retrieval and Rendering of PDF or Other Input Stream Data

You can display arbitrary `InputStream` content in a window. For example, you can display a PDF returned from code that returns PDF data as a byte stream (`byte[]`) from a plugin, encapsulated in a `DocumentContentsInfo` object.

Use the following utility method from Gosu including PCF files:

```
gw.api.document.DocumentsUtil.renderDocumentContentsDirectly(fileName, docInfo)
```

There are other utility methods on the `DocumentsUtil` object that may be useful to you. Refer to the API reference documentation in the Studio Help menu for more details.

For more information about PDF creation, see “Server-side PDF Licensing” on page 238.

Document Production

BillingCenter provides a user interface and integration APIs for creating documents, downloading documents, and producing automated form letters. The APIs include the ability to integrate the BillingCenter document user interface with a separate corporate document management system that can store the documents and optionally the document metadata.

This topic includes:

- “Document Production Overview” on page 231
- “Document Template Descriptors” on page 239
- “Generating Documents from Gosu” on page 250

See also

- For general information on plugins, which are an important part of document management in BillingCenter, see “Plugin Overview” on page 135.
- For an overview of document management and storage, see “Document Management” on page 217

Document Production Overview

BillingCenter can create new documents from forms, form letters, or other structured data. For example, notification emails, letters, or faxes created by business rules to an insured customer. The resulting new document optionally can be attached to business data objects. BillingCenter can create some types of new documents from a server-stored template without user intervention.

Users can create new documents locally from a template and then attach them to a account or other BillingCenter objects. This is useful for generating forms or form letters or any type of structured data in any file format. Or, in some cases BillingCenter creates documents from a server-stored template without user intervention. For example, BillingCenter creates a PDF document of an outgoing notification email and attaches it to the account.

The `IDocumentProduction` interface is the plugin interface used to create new document contents from within the Guidewire application. It interfaces with a document production system in a couple of different ways, depending on whether the new content can be returned immediately or requires delayed processing. These two modes are:

- **Synchronous production** – The document contents are returned immediately from the creation methods.
- **Asynchronous production** – The creation method returns immediately, but the actual creation of the document is performed elsewhere and the document may not exist for some time.

There are different integration requirements for these two types of document production.

There are several document response types:

- **URL** – A URL to a local content store to display the content. The URL response type is the recommended response type if your DMS supports it. The URL response type permits the highest performance for BillingCenter.
- **DOCUMENT_CONTENTS** – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 229

These settings are defined as the `ResponseType` property within `DocumentContentsInfo`, which is the return result from synchronous production methods.

The details of these plugins are discussed in “User Interface Flow for Document Production” on page 232. For locations of templates and descriptors, see “Template Source Reference Implementation” on page 249.

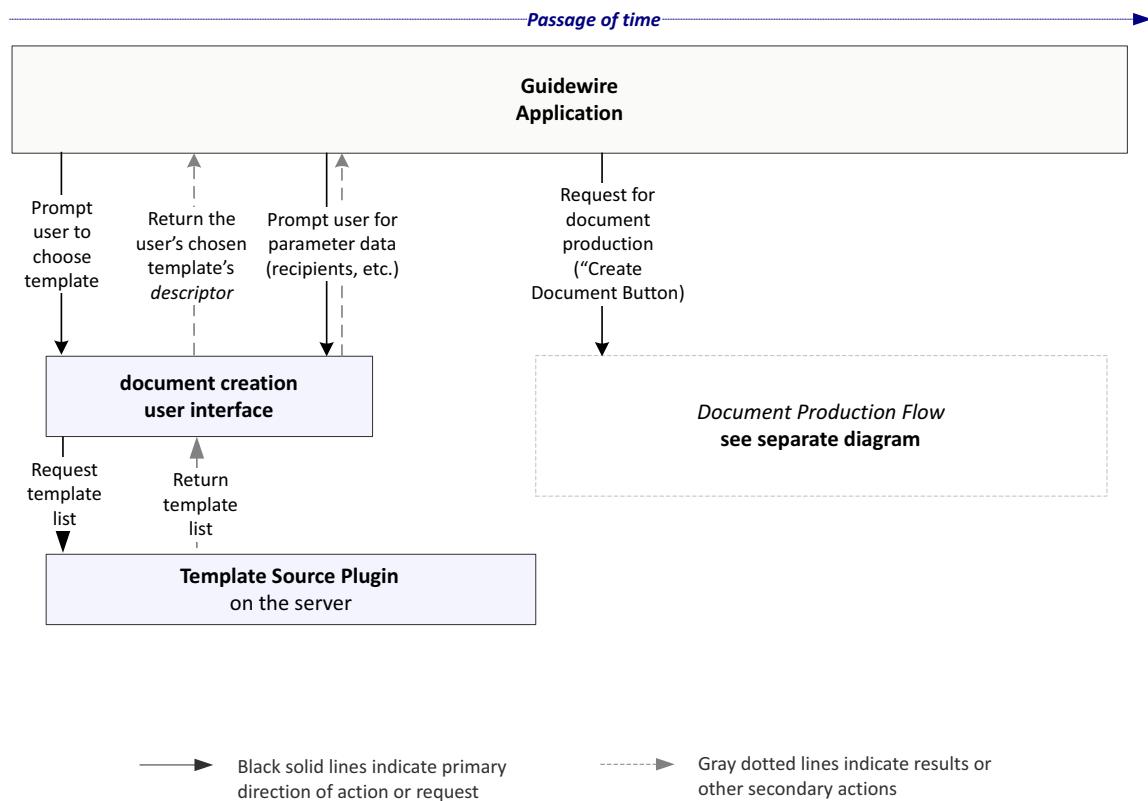
After document production, storage plugins store the document in the document management system.

User Interface Flow for Document Production

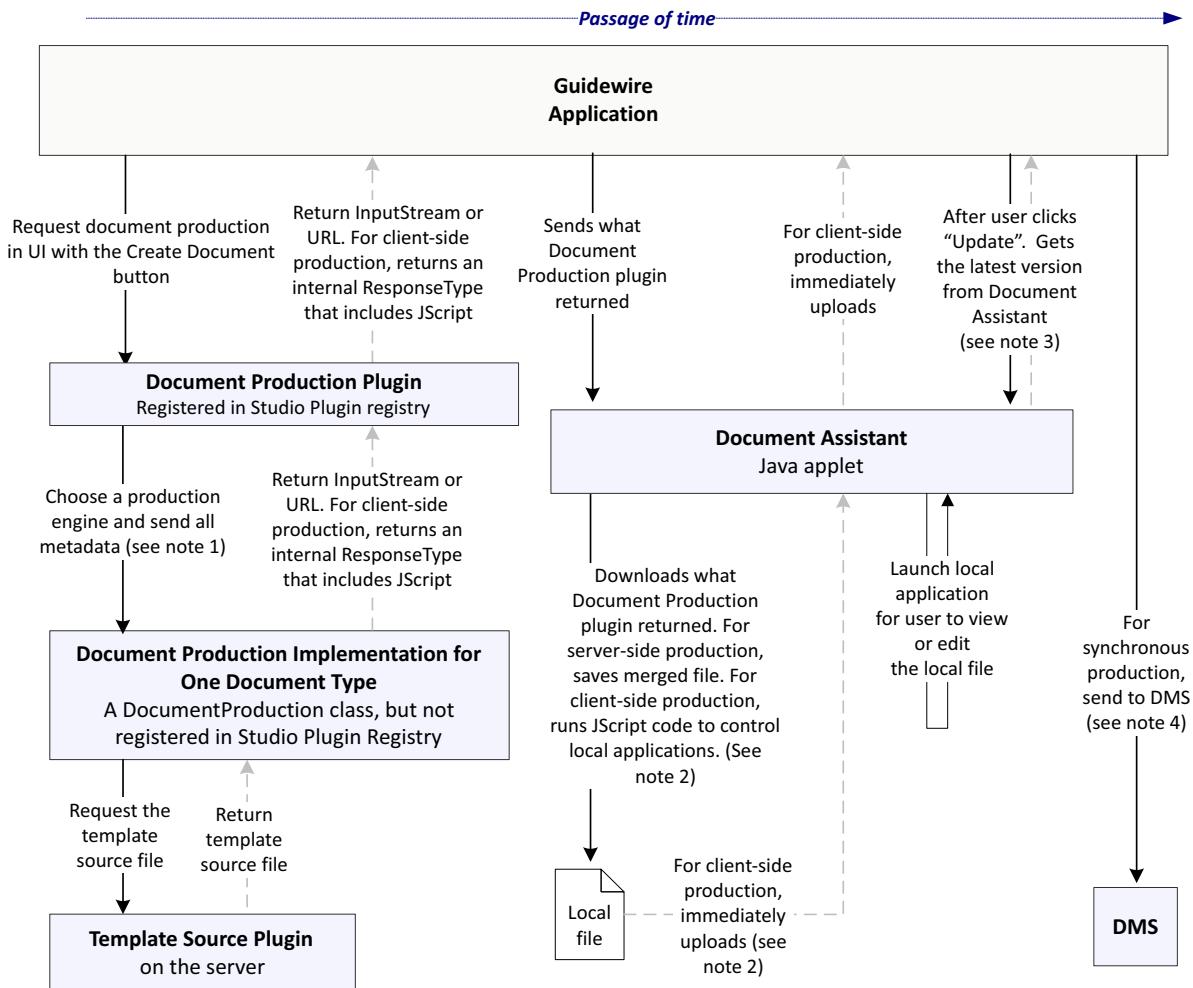
From the BillingCenter user interface, create forms and letters and choose the desired template. You can select other parameters (some optional) to pass to the document production engine. BillingCenter supports Gosu-initiated automatic document creation from business rules. The automatic and manual processes are similar, but the automatic document creation skips the first few steps of the process. The automated creation process skips some steps relating to parameters that the user chooses in the user interface. Instead, the Gosu code that requests the new document sets these values.

The following diagrams illustrates the interactions between BillingCenter, the various plugin implementations, the Document Assistant, and an external document management system to manage the generation of a form or letter.

Interactive Document Creation UI Flow



Document Production Flow



Notes

1. The template descriptor for this request determines the document production implementation. It specifies a template handler String or a MIME type, both of which map to a document production implementation.
2. For client-side production, if Document Assistant is not present, the Guidewire application sends the web browser a JScript file. The JScript file does the equivalent of Document Assistant if you properly configure the PC and approve the request. To upload, the user interface prompts you to locate the merged file on the local file system.
3. If the Document Assistant is not present, you must manually locate the merged file on the local file system to upload.
4. For asynchronous production, after completion, the document production plugin sends the document to the DMS.



Primary direction of action or request



Results or other secondary actions

The chronological steps are as follows:

1. BillingCenter invokes the document creation user interface to let you select a form or letter template.
2. The BillingCenter document creation user interface requests a list of templates from the *template source*, which is a plugin that acts like a repository of templates.
3. The template source returns a list of templates.
4. The document creation user interface displays a list of templates and lets you choose one, perhaps after searching or filtering among the list of templates. Searching and filtering uses the template metadata, which is information about the templates themselves.
5. After you select a template, the document creation user interface returns the *template descriptor* information for the chosen template to BillingCenter.
6. BillingCenter prompts you through the document creation user interface for other document production parameters for the document merge. For example, choose the recipient of a letter or other required or optional properties for the template. After you enter this data, the user interface returns the parameter data to BillingCenter.
7. BillingCenter chooses the appropriate document production plugin from values in the template descriptor. The descriptor file can indicate a specific document production plugin or let one be chosen based on the MIME type of the file. The document production plugin gets a template descriptor and the parameter data values.
8. The document production plugin obtains the actual template file from the template source.
9. The document production plugin takes whatever steps are necessary to launch an editor application or other production engine for merging the template file with the parameter data. BillingCenter includes document production plugins that process Microsoft Word files, Adobe Acrobat (PDF) files, Microsoft Excel files, Gosu templates, and plain text files. For Gosu templates, the Gosu document production plugin executes the Gosu template using the parameter data.

The result of this step is a *merged document*. It is possible to produce documents using applications on the user's computer, a process known as *client-side document production*. For example, the built-in production plugins use client-side production for Microsoft Word and Microsoft Excel documents.
10. If the document was created from the user interface, BillingCenter automatically downloads the file to your local machine if the configuration parameter `AllowDocumentAssistant` is enabled. The file opens in the appropriate application on the desktop. You can print the file, and if the editor application supports editing, you can change the file and save it in the editor application. This describes the reference implementation, but the user interface flow can work differently if you customize the PCF files for a different workflow.
11. If the file is on your PC, you upload the completed, locally merged document, or you can choose an alternative document to upload. You can then specify some additional parameters about the document. If the configuration parameter `AllowDocumentAssistant` is enabled, the upload step by the Guidewire Document Assistant supplies the location of the locally merged document. If `AllowDocumentAssistant` is disabled or the Guidewire Document Assistant control is not installed, then you must browse manually to the local location of the merged document to upload.
12. Additionally, depending on the type of document production, the production engine could add the document to the document storage system. The production system or the storage system could notify BillingCenter using plugin code or from an external system using the web services API. For more information about document storage, see "Document Management Overview" on page 217.

The preceding steps describe the reference implementation, but you can configure many parts of the flow to suite your needs. You can configure the user interface flow to work differently by modifying the user interface PCF configuration files. In some cases, you can configure the flow by modifying the document-related plugins.

Document Production Plugins

As mentioned in “User Interface Flow for Document Production” on page 232, BillingCenter supports two types of document production:

- *synchronous production* – the document contents return immediately
- *asynchronous production* – the creation method returns immediately but creation happens later

These two types of document production have different integration requirements.

For synchronous production, BillingCenter and its installed *document storage* plugins are responsible for persisting the resulting document, including both the document metadata and document contents. In contrast, for asynchronous document creation, the *document production* (`IDocumentProduction`) plugins are responsible for persisting the data.

The `IDocumentProduction` plugin is an interface to a document creation system for a certain type of document. The document creation process may involve extended workflow and/or asynchronous processes, and it may depend on `Document` entity or set properties in the `Document`.

There are some additional related interfaces that assist the `IDocumentProduction` plugin:

- `IDocumentTemplateSource` and `IDocumentTemplateDescriptor` – Encapsulate the basic interface for searching and retrieving templates that describe the document to be created.
The descriptive information includes the basic metadata (name, MIME type, and so on) and a pointer to the template content. Specifically, `IDocumentTemplateDescriptor` describes the templates used to create documents and `IDocumentTemplateSource` plugin actually lists and retrieves the document templates
- `IPDFMergeHandler` – Used in creation of PDF documents. View the built-in implementation to set parameters in the default implementation.

The `IDocumentProduction` plugin has two main methods: `createDocumentSynchronously` and `createDocumentAsynchronously`.

- The `createDocumentSynchronously` method returns document contents as:
 - URL – A URL to a local content store to display the content. The URL response type is the recommended response type if your DMS supports it. The URL response type permits the highest performance for BillingCenter.
 - DOCUMENT_CONTENTS – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 229

These settings are defined as the `ResponseType` property within `DocumentContentsInfo`, which is the return result from synchronous production methods.

For documents created synchronously, the caller of the `createDocumentSynchronously` method must pass the contents to the document content plugin for persistence. The `Document` parameter can be modified if the `DocumentProduction` plugin wants to set any properties for persistence in the database.

- The `createDocumentAsynchronously` method returns the document status, such as a status URL that could display the status of the document creation. In the reference implementation, none of the built-in document production plugins support asynchronous document creation. The default user interface does not actually use the results of the `createDocumentAsynchronously` method. To support this type of status update, customize the user interface PCF files to generate a new user interface. Also, customize the included `DocumentProduction` Gosu class that generates documents from Gosu. By default, `DocumentProduction` does not use the result of the `createDocumentAsynchronously` method.
- For documents created asynchronously, your `createDocumentAsynchronously` method must put the newly created contents into the DMS. Next, your external system can use web service APIs to add the document to notify BillingCenter that the document now exists. See “Web Service APIs to Attach Documents to Business Objects” on page 229.

If your code to add the document is running within the server in local Gosu or Java code, do not call the SOAP APIs to call back the same server. Calling back to the same server using SOAP is not generally safe. Instead, use domain methods on the entity to add the entity. For example:

```
newDocumentEntity = Account.addDocument()
```

In either case, immediately throw an exception if any part of your creation process fails.

It is possible to produce documents using applications on the user's computer, a process known as *client-side document production*. For example, the built-in production plugins use client-side production for Microsoft Word and Microsoft Excel documents. The Document Assistant can call the local application to open the file and merge in necessary properties. On Windows, the Document Assistant sends JScript that can control local applications.

For more information about the built-in production plugins, see "Built-in Document Production Plugins" on page 250.

For more details on specific methods of these plugins, refer to the Java API Reference Javadoc. In the Javadoc, there are two versions of the `IDocumentProduction` interface. There is a base class Guidewire platform version with ".pl." in the fully-qualified package name. To implement the plugin, you must implement the other version, the BillingCenter-specific version, which has ".bc." in its fully-qualified package name.

Configuring Document Production and MIME Types

Document production configuration is defined by the registry for the `IDocumentProduction` plugin in the Plugins editor in Studio. The registry includes a list of parameters that define *template types* and their corresponding `IDocumentProduction` plugin implementations. In the registry, a template type is either a MIME type that BillingCenter recognizes or a text value that you provide to define a template type.

Defining a Template Type for Document Production

You define template types by adding parameters to the list in the registry for the `IDocumentProduction` plugin. Each parameter in the list specifies the name of a template type and a class that implements the `IDocumentProduction` plugin interface to handle production for that template type. Separately, you must deploy the implementation class to the `BillingCenter/modules/configuration/plugins/document/classes` directory and necessary libraries to the `BillingCenter/modules/configuration/plugins/document/lib` directory.

Built-in Implementations of the Document Production Plugin

A typical document production configuration includes parameters for template types that reference the built-in `IDocumentProduction` implementations for common file types. For example, the template type named `application/msword` specifies

`com.guidewire.bc.plugin.document.internal.WordDocumentProductionImpl` as the implementation class.

Refer to the plugin registry for the `IDocumentProduction` plugin to see additional built-in plugin implementations.

How BillingCenter Determines which Plugin Implementation to Use for Document Production

BillingCenter determines which `IDocumentProduction` implementation to use to produce a document from a specific template by following this procedure:

1. BillingCenter searches for a non-MIME-type entry in the plugin parameter list that matches the `documentProductionType` property of the template. If a match is found, BillingCenter proceeds with document production.
2. If no match is found, BillingCenter searches for a MIME-type entry in the plugin parameter list that matches the `mimeType` property of the template. If a match is found, BillingCenter proceeds with document production.
3. If no match is found, document production fails.

You can replace the default dispatching implementation with your own `IDocumentProduction` implementation.

Server-Side PDF Document Production

BillingCenter supports server-side PDF production but not client-side PDF production.

Adding a custom MIME type for Document Production

Adding a custom MIME type that BillingCenter recognizes requires several steps.

1. In `config.xml`, add the new MIME type to the `<mimetypemapping>` section. The information for each `<mimetype>` element contains these attributes:

- `name` – The name of the MIME type. Use the same name as in the plugin registry, such as `text/plain`.
- `extension` – The file extensions to use for the MIME type.

If more than one extension applies, use a pipe symbol ("|") to separate them. BillingCenter uses this information to map between MIME types and file extensions. To map from a MIME type to a file extension, BillingCenter uses the first extension in the list. To map from file extension to MIME type, BillingCenter uses the first `<mimetype>` entry that contains the extension.

- `icon` – The image file for documents of this MIME type. At runtime, the image file must be in the `SERVER/webapps/bc/resources/images` directory.

2. Add the MIME type to the configuration of the application server, if required.

How you add a MIME type depends on the brand of application server. For Tomcat, you configure MIME types in the `web.xml` configuration file by using `<mime-mapping>` elements. Assure the MIME type that you need is not in the list already you try to add it.

See also

- For basic instructions, see “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*.
- For more about information built-in `IDocumentProduction` implementations classes, see “Built-in Document Production Plugins” on page 250.

Server-side PDF Licensing

In the BillingCenter reference implementation, the server-side PDF production software is made by a company called Big Faceless Org (BFO). Without a license, the generated documents contain a large watermark that reads “DEMO” on the face of each generated page, rather than preventing document creation entirely. To remove the watermark, you must obtain a license key through Guidewire Customer Support.

All server-side PDF production licenses are licensed per server *CPU*, not per server.

With your copy of BillingCenter, you are entitled to four CPU licenses for PDF production. However, you must still go through a process to obtain the keys associated with your licenses. In addition, you may purchase additional licenses through the same process.

To obtain a license key:

1. Contact your Customer Support Partner or email support@guidewire.com with your request for a PDF production license for BillingCenter.
2. A support engineer sends you an Authorization Form.
3. Fill out the Authorization Form, including the number of CPUs to use. For multi-CPU servers, include the total number of CPUs.
4. Fax the filled-out form to Guidewire prior to issuing the license.
5. The Guidewire support engineer requests license keys from the appropriate departments.

6. Once the license keys are obtained, Guidewire emails the designated customer contact (per the information on the form) with the license information.

If you have additional questions about this process, email support@guidewire.com.

Configuration

The default configuration appears in the Plugins registry in Studio. In the Project window, navigate to Configuration → config → Plugins → registry, and then open IPDFMergeHandler.gwp. For information about using the plugins editor, see “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*.

Ensure the implementation class is set to `gw.plugin.document.impl.BFOMergeHandler`, which is the default.

In the list of plugin parameters, set the following two plugin parameter values:

- Set plugin parameter `BatchServerOnly` value to `true` for typical usage. The `BatchServerOnly` parameter determines whether PDF production happens on each server or solely on the batch server.
- Set plugin parameter `LicenseKey` value to your server key. The `LicenseKey` value is empty in the default configuration. You must acquire your own BFO licenses from customer support. In a clustered environment you must also set the license key value in the `PDFMergeHandlerLicenseKey` parameter in `config.xml`.

BillingCenter uses the `IPDFMergeHandler` plugin interface only for server-side PDF production in the default implementation of the `IDocumentProduction` interface.

Document Template Descriptors

A *document template descriptor* describes an actual document template, such as a Microsoft Word mail merge template, an Adobe PDF form, or a Gosu template. An interface called `IDocumentTemplateDescriptor` defines the API for an object that represents a document template descriptor.

IMPORTANT In most cases, it is best to use the built-in implementation of the `IDocumentTemplateDescriptor` interface. The built-in implementation reads template information from a standard XML file. You can modify XML data if desired. However, most customers do not need to modify the code that reads or writes the XML file. Be aware that some information in this documentation topic is included for the rare case that you might need to write your own implementation of `IDocumentTemplateDescriptor`.

A template descriptor contains four different kinds of information:

- **Template metadata** – Template metadata is metadata about the template itself, not the file. Template metadata includes the template ID, the template name, and the calendar dates that limit the availability of the template.
- **Document metadata defaults** – Document metadata defaults are attributes that are applied to documents after their creation from the template, or as part of their creation, for example the default document status.
- **Context objects** – Context objects are objects that can be inserted into the document template, or have properties extracted from them before inserting them into the document template. For example, an email document template might include To and CC recipients as context objects, each of which is of the type `Contact`. The context objects include default values to be inserted, as well as a set of legal alternative values for use in the BillingCenter document creation user interface. A context object is an object attached to the merge request that can either be inserted into the document or certain properties within the object extracted from it.
- **Property names and values to merge** – Each descriptor defines a set of template field names and values to insert into the document template, including optional formatting information. Effectively, this describes which BillingCenter data values to merge into which fields within the document template. For example, an email document template might have To and CC recipients as context objects called `To` and `CC` of type `Contact`. The template might have a context object called `InsuredName` that extract the value `To.DisplayName`.

The `IDocumentTemplateDescriptor` interface is closely tied to the XML file format which corresponds to the default implementation of the `IDocumentTemplateSerializer` interface. The `IDocumentTemplateDescriptor` API consists mostly of property getters, with one setter (for `DateModified`) and some additional utility methods.

For locations of templates and descriptors, see “Template Source Reference Implementation” on page 249.

Template Descriptor Fields for Metadata About Each Template

The following list contains properties and methods that relate to metadata as defined in the `IDocumentTemplateDescriptor` interface. The second column specifies whether that property is required or optional within the XML files that the default implementation uses.

| Property | Required in the XML file in the default implementation | Description |
|-------------------------------------|--|---|
| <code>templateId</code> | Required | The unique ID of the template |
| <code>name</code> | Required | A human-readable name for the template |
| <code>identifier</code> | Optional | An additional human-readable identifier for the template. This often corresponds to a well-known domain-specific document code. For example, to indicate a state-mandated form for this template. |
| <code>scope</code> | Required | The contexts that this template supports. Possible values are <ul style="list-style-type: none"> • <code>ui</code> – the document template must only be used from the document creation user interface • <code>rules</code> – the document template must only be used from rules or other Gosu and must not appear in a list the user interface template. • <code>all</code> – the document template may be used from any context |
| <code>description</code> | Required | A human-readable description of the template and/or the document it creates |
| <code>password</code> | Optional | If present, holds the password which is required for the user to create a document from the template. May not be supported by all document formats (for example, not supported for Gosu templates). In Microsoft Word, use the option to Protect Document... and select the Forms option. Then use the same password in the descriptor file as used to protect the document. To merge account data into the form, BillingCenter needs to unlock it, merge the data, and relock the form. BillingCenter checks whether a form is locked and attempts to unlock it (and later relock it) using the password provided. |
| <code>mimeType</code> | Required | The type of document to create from this document template. In the built-in implementation of document source, this determines which <code>IDocumentProduction</code> implementation to use to create documents from this template. Also see <code>documentProductionType</code> . |
| <code>documentProductionType</code> | Optional | If present, a specified document production type specifies which implementation of <code>IDocumentProduction</code> to use to create a new document from the template. This is not the only way to select a document production plugin implementation. You can also use a MIME type (see <code>mimeType</code>). |
| <code>dateModified</code> | Optional | The date the template was last modified. In the default implementation, this is set from the information on the XML file itself. Both getter and setter methods exist for this property so that the date can be set by the <code>IDocumentTemplateSource</code> implementation. This property is not present in the XML file. However, the built-in implementation of the <code>IDocumentTemplateDescriptor</code> interface generates this property |

| Property | Required in the XML file in the default implemen- tation | Description |
|-------------------------------|---|---|
| dateEffective, dateExpiration | Required | The effective and expiration dates for the template. If you search for a template, BillingCenter displays only those for which the specified date falls between the effective and expiration dates. However, this does not support different versions of templates with the same ID. The ID for each template must be unique. You can still create documents from templates from Gosu (from PCF files or rules) independent of these date values. Gosu-based document creation uses template IDs but ignores effective dates and expiration dates |
| keywords | Optional | A set of keywords search for within the template. Delimit keywords with the pipe () symbol in the XML file. |
| requiredPermission | Optional | The code of the SystemPermissionType value required for the user to see and use this template. Templates for which the user does not have the appropriate permission do not appear in the user interface. This setting does not prevent creation of a document by Gosu (PCF files or rules). |
| mailmergetype | Optional | Optional configuration of pagination of client-side Microsoft Word production. By default, BillingCenter uses Microsoft Word <i>catalog pagination</i> , which correctly trims the extra blank page at the end. However, catalog pagination forbids template substitution in headers and footers. In contrast, <i>standard pagination</i> adds a blank page to the end of the file but enables template substitution in headers and footers. Set this attribute to the value <i>catalog</i> to use catalog pagination. To use standard pagination, do not set this attribute. |

Methods in the `IDocumentTemplateDescriptor` interface related to metadata

| | | |
|------------------------------------|-----|---|
| getMetadataPropertyNames method | n/a | This method returns the set of extra metadata properties which exist in the document template definition. This is used in conjunction with <code>getMetadataPropertyValue()</code> as a flexible extension mechanism. You can add arbitrary new fields to document template descriptors. BillingCenter passes new properties to the internal entities that display document templates in the user interface. Also, if the extra property names correspond to properties on the Document entity, the BillingCenter passes values to documents created from the template. |
| getMetadataPropertyNames method | n/a | This method returns the set of extra metadata properties which exist in the document template definition. This is used in conjunction with <code>getMetadataPropertyValue()</code> as a flexible extension mechanism. You can add arbitrary new fields to document template descriptors. BillingCenter passes new properties to the internal entities that display document templates in the user interface. Also, if the extra property names correspond to properties on the Document entity, the BillingCenter passes values to documents created from the template. |
| getMetadataPropertyValue method | n/a | This method gets a property value. The method takes one argument, which is a property name as a <code>String</code> value. See <code>getMetadataPropertyNames()</code> . |

Template Descriptor Fields and Defaults for Document Metadata

The following template descriptor fields define the defaults for document type and security types:

| Property | Required in the XML file in the default implemen- tation | Description |
|--------------------------------|---|---|
| templateType / type (see note) | Required | Corresponds to the DocumentType typelist. Documents created from this template have their type fields set to this value. This is the only property in the XML file format with a property name difference from the property in the interface: <ul style="list-style-type: none"> • In the XML this property is called type • In the IDocumentTemplateDescriptor interface, this property is called templateType |
| defaultSecurityType | Required | Security type in the DocumentSecurityType typelist. This is the security type that becomes the default value for the corresponding document metadata fields for documents created using this template. |

Template Descriptor Fields and Context Objects

As you write templates that include Gosu expressions, you need to reference business data such as the current Account entity. Reference entities within templates as objects called *context objects*. Context objects create variables that form field expressions can refer to by name.

In addition to context objects that you define, form fields can always reference the account using the account symbol, for example, account.property or account.methodName().

Only having access to one or two high-level root objects would get challenging. For example, suppose you want to address a account acknowledgement letter to the main contact on the account. Without context objects, each form field would have to repeat the same prefix (Account.MainContact.) many times:

```
<FormField name="ToName">Policy.GetInsured().DisplayName</FormField>
<FormField name="ToCity">Policy.GetInsured().PrimaryAddress.City</FormField>
<FormField name="ToState">Policy.GetInsured().PrimaryAddress.State</FormField>
...

```

This could become hard to read with complex lengthy prefixes. You can simplify template code by creating a context object that refers to the intended recipient. Each context object must have two attributes: a unique name (such as "To") and a type (as a string, such as "Contact"; see the following discussion for the legal values). In addition, each context object must contain a DefaultObjectValue tag. This tag can contain any valid Gosu expression that identifies the default value to use for this ContextObject. You can construct a context object for your recipient as follows:

```
<ContextObject name="To" type="Contact">
<DefaultObjectValue>Account.GetInsured()</DefaultObjectValue>
</ContextObject>
```

You can simplify the form fields to the following:

```
<FormField name="ToName">To.DisplayName</FormField>
<FormField name="ToInsuredCity">To.PrimaryAddress.City</FormField>
<FormField name="ToState">To.PrimaryAddress.State</FormField>
<FormField name="ToZip">To.PrimaryAddress.PostalCode</FormField>
...

```

Context objects serve a second purpose by allowing you to manually specify the final value of each context object within the user interface. If you select a document template from the chooser, BillingCenter displays a series of choices, one for each context object. The name of the context object appears as a label on the left, and the default value appears on the right. The document wizard can optionally allow users to pick from a list of possible values using the `PossibleObjectValues` tag as follows:

```
<ContextObject name="To" type="Contact">
  <DefaultObjectValue>Account.getInsured()</DefaultObjectValue>
  <PossibleObjectValues>Account.getContacts()</PossibleObjectValues>
</ContextObject>
```

The `PossibleObjectValues` value must be a Gosu expression that evaluates to an array, typically an array of Guidewire entities although that is not required. BillingCenter does not enforce that the `DefaultObjectValue` is of the same type as `PossibleObjectValues`. While this makes it possible to have two different types for one context object, generally Guidewire recommends against this approach. If you were to do so, you must write form field expressions that work with two different types for that context object.

Context objects must be of one of the following types:

| Context object type | Meaning |
|--------------------------|--|
| <code>EntityName</code> | The name of any BillingCenter entity such as <code>Account</code> , or <code>Activity</code> . The possible object values appear as a drop-down list of options. If that type of entity has a special type of picker, it also displays. For example, if you set the context object type to "Contact", users can use the Contact picker to search for a different value. Similarly, if you set the context object type to "User", BillingCenter displays a user picker. |
| <code>Entity</code> | This is a sort of "wildcard" type that indicates support for any keyable entity. Unlike the entry listed earlier with a specific entity name, this type is actually the literal string "Entity". This is useful for heterogeneous lists of objects. |
| <code>TypekeyName</code> | The name of any BillingCenter typekey such as "YesNo". |
| <code>string</code> | A case-sensitive all-lower-case string to appear in the user interface as a single line of text. The <code><DefaultObjectValue></code> tag must be present, and its contents indicates the default text for this context object. Ignores the <code><PossibleObjectValues></code> tag. |
| <code>text</code> | Case sensitive, must be all lower case. Appears in the user interface as several lines of text. The <code><DefaultObjectValue></code> tag must be present, and its contents indicates the default text for this context object. If you use this, BillingCenter ignores the <code><PossibleObjectValues></code> tag. |

By default, the `name` attribute of the context object becomes the user-visible name. If you want to use a different user-visible name, set the context object's `display-name` attribute to the text that you want to be visible to the user.

The `type` attribute on the `ContextObject` is used to indicate how the user interface presents the object in the document creation user interface. Valid options include: `String`, `text`, `Contact`, `User`, `Entity`, `Account`, or any other BillingCenter entity name or typekey type.

If the context object is of type `String`, then the user would typically be given a single-line text entry box.

If the context object has type `Text`, the user sees a larger text area. However, if the `ContextObject` definition includes a `PossibleObjectValues` tag containing Gosu that returns a `Collection` or array of `String` objects, the user interface displays a selection picker. For example, use this approach to offer a list of postal codes from which to choose. If the object is of type `Contact` or `User`, in addition to the drop-down box, you see a picker button to search for a particular contact or user. All other types (`Entity` is the default if none is specified) are presented as a drop-down list of options. If the `ContextObject` is a typekey type, then the default value and possible values fields must generate Gosu objects that resolve to `TypeKey` objects, not text versions of typecodes values.

There are a few instances in BillingCenter system in which entity types and typekey types have the same name, such as `Contact`. In this case, Gosu assumes you mean the entity type. If you want the typelist type, or want clearer code, use the fully qualified name of the form `entity.EntityName` or `typekey.TypeKeyName`.

The `IDocumentTemplateDescriptor` interface includes the following methods related to context objects:

- `String[] getContextObjectNames()` – Returns the set of context object names defined in the document template. See the XML document format for more information on what the underlying configuration looks like.
- `String getContextObjectType(String objName)` – Returns the type of the specified context object. Possible values include "string", "text", "Entity" (for any entity type), or the name of an entity type such as "Account".
- `boolean getContextObjectAllowsNull(String objName)` – Returns `true` if `null` is a legal value, `false` otherwise.
- `String getContextObjectDisplayName(String objName)` – Returns a human-readable name for the given context object, to display in the document creation user interface.
- `String getContextObjectDefaultValueExpression(String objName)` – Returns a Gosu expression which evaluates to the desired default value for the context object. Use this to set the default for the document creation user interface, or as the value if a document is created automatically.
- `String getContextObjectPossibleValuesExpression(String objName)` – Returns a Gosu expression that evaluates to the desired set of legal values for the given context object. Used to display a list of options for the user in the document creation user interface.
- `getCompiledContextObjectDefaultValueExpression` and `getFormFieldCompiledExpression` – In the rare case you need to implement the `IDocumentTemplateDescriptor` interface, these two methods require special return result types. For information about implementing these methods, contact Guidewire Customer Support.

Template Descriptor Fields Related to Form Fields and Values to Merge

Form fields dictate a mapping between BillingCenter data and the merge fields in the document template.

For example, you might want to merge the account number into a document field using the simple Gosu expression "Account.AccountNumber".

The full set of template descriptor fields relating to form fields are as follows:

- `String[] getFormFieldNames()` – Returns the set of form fields defined in the document template. Refer to the following XML document format for more information on what the underlying configuration looks like.
- `String getFormFieldValueExpression(String fieldName)` – Returns a Gosu expression that evaluates to the desired value for the form field. The Gosu expression is usually written in terms of one or more Context Objects, but any legal Gosu expression is allowed.
- `String getFormFieldDisplayValue(String fieldName, Object value)` – Returns the string to insert into the completed document given the field name and the value. The value is typically the result of evaluating the expression returned from the method `getFormFieldValueExpression`. Use this method to rewrite values if necessary, such as substituting text. For example, display text that means "not applicable" ("<n/a>") instead of `null`, or format date fields in a specific way.

XML Format of Built-in IDocumentTemplateSerializer

The default implementation of `IDocumentTemplateSerializer` uses an XML format that closely matches the fields in the `DocumentTemplateDescriptor` interface. This is intentional. The purpose of `IDocumentTemplateSerializer` is to serialize template descriptors and let you define the templates within simple XML files. The XML format is suitable in typical implementations. BillingCenter optionally supports different potentially elaborate implementations that might directly interact with a document management system storing the template configuration information.

IMPORTANT Many of the fields in this section are defined in more detail in the previous section, “Document Template Descriptors” on page 239. The XML format described in this section is basically a serialization of the fields in the `IDocumentTemplateDescriptor` interface. In the reference implementation, the built-in `IDocumentTemplateDescriptor` and `IDocumentTemplateSerializer` classes implement the serialization.

The default implementation `IDocumentTemplateSerializer` is configured by a file named `document-template.xsd`. The default implementation uses XML that looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<DocumentTemplateDescriptor
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="document-template.xsd"
    id="ReservationRights.doc"
    name="Reservation Rights"
    description="The initial contact letter/template."
    type="letter_sent"
    lob="GL"
    state="CA"
    mime-type="application/msword"
    date-effective="Apr 3, 2003"
    date-expires="Apr 3, 2004"
    keywords="CA, reservation">
    <ContextObject name="To" type="Contact">
        <DefaultObjectValue>Policy.GetInsured()</DefaultObjectValue>
        <PossibleObjectValues>Account.getRelatedContacts()</PossibleObjectValues>
    </ContextObject>
    <ContextObject name="From" type="Contact">
        <DefaultObjectValue>Account.AssignedUser.Contact</DefaultObjectValue>
        <PossibleObjectValues>Account.getRelatedUserContacts()</PossibleObjectValues>
    </ContextObject>
    <ContextObject name="CC" type="Contact">
        <DefaultObjectValue>Account.Driver</DefaultObjectValue>
        <PossibleObjectValues>Account.getRelatedContacts()</PossibleObjectValues>
    </ContextObject>
    <FormFieldGroup name="main">
        <DisplayValues>
            <NullDisplayValue>No Contact Found</NullDisplayValue>
            <TrueDisplayValue>Yes</TrueDisplayValue>
            <FalseDisplayValue>No</FalseDisplayValue>
        </DisplayValues>
        <FormField name="AccountNumber">Account.AccountNumber</FormField>
    </FormFieldGroup>
</DocumentTemplateDescriptor>
```

At run time, this XSD is referenced from a path relative to the module `config\resources\doctemplates` directory. If you want to change this value, in the plugin registry for this plugin interface, in Studio in the Plugins editor, set the `DocumentTemplateDescriptorXSDLocation` parameter. To use the default XSD in the default location, set that parameter to the value “`document-template.xsd`” assuming you keep it in its original directory.

The attributes on the `DocumentTemplateDescriptor` element correspond to the properties on the `IDocumentTemplateDescriptor` API.

Date Formats in the Document Template XML File

Date values may be specified in the XML file in any of the following formats for systems in the English locale:

| Date format | Example |
|---|--------------------------------|
| MMM d, yyyy | Jun 3, 2005 |
| MMMM d, yyyy Note: Four M characters mean the entire month name) | June 3, 2005 |
| MM/dd/yy | 10/30/06 |
| MM/dd/yyyy | 10/30/2006 |
| MM/dd/yy hh:mm a | 10/30/06 10:20 pm |
| yyyy-MM-dd HH:mm:ss.SSS | 2005-06-09 15:25:56.845 |
| yyyy-MM-dd HH:mm:ss | 2005-06-09 15:25:56 |
| yyyy-MM-dd'T'HH:mm:ss zzz | 2005-06-09T15:25:56 -0700 |
| EEE MMM dd HH:mm:ss zzz yyyy | Thu Jun 09 15:24:40 -0700 2005 |

Refer to the following codes for the date formats listed in the earlier table:

- a = AM or PM
- d = day
- E = Day in week (abbrev.)
- h = hour (24 hour clock)
- m = minute
- M = month (MMMM is entire month name)
- s = second
- S = fraction of a second
- T = parse as time (ISO8601)
- y = year
- z = Time Zone offset.

The first three formats typically are the most useful because templates typically expire at the end of a particular day rather than at a particular time. If you use any of the `template_tools` command line tools commands, you cannot rely on the date format in input files remaining. Although BillingCenter preserves the values, the date format may change.

For text elements, such as month names, BillingCenter requires the text representations of the values to match the current international locale settings of the server. For example, if the server is in the French locale, you must provide the month April as "Avr", which is short for Avril, the French word for April.

Context Objects in the Document Template Descriptor XML File

A context object is an object attached to the merge request. You can insert a context object into the document or insert only certain fields within the object to the document.

For each `ContextObject` tag, the `DefaultObjectValues` expression determines the object (a contact in this case) to initially select. The `PossibleObjectValues` expression determines the set of objects to display in the select control.

Notice that all Gosu expression are in the contents of the tag, rather than attributes on the tag, which makes formatting issues somewhat easier. Also, there is still always a `Account` entity called `account` in context as the template runs. If you have time-intensive lookup operations, perform lookups once for the entire document, rather than once for each field that uses the results. To do this, write Gosu classes that implement the lookup logic and cache the lookup results as needed.

Form Fields in the Document Template Descriptor XML File

Form fields dictate a mapping between BillingCenter data and merge fields in the document template. For example, to merge the account number into the field `AccountInfo`, use the following expression:

```
<FormField name="AccountInfo">Account.AccountNumber</FormField>
```

`FormField` tags can contain any valid Gosu expression. This capability is most useful to combine with Studio-based utility libraries or class extensions. For example, to render the account number and also other information about the account, encapsulate that logic into an entity enhancement method called `Account.getAccountInformation()`. Reference that function in the form field as follows:

```
<FormField name="AccountInfo">Account.getAccountInformation()</FormField>
```

Form fields can have two additional attributes: `prefix` and `suffix`. Both attributes are simple text values. Use these in cases in which you want to always display text such as “The account information is ____.”, so you can rewrite the form field to look like:

```
<FormField name="MainContactName"
  prefix="The account information is "
  suffix=". ">
  Account.getAccountInformation()</FormField>
```

Form Fields Groups in the Document Template Descriptor XML File

You can optionally define form field groups that logically group a set of form fields. Groups are most useful for defining common attributes across the set of form fields, such as a common date string format. You can also use groups to define a `String` to display for the values `null`, `true`, or `false`. For example, you could check a Boolean value and display the text “`Police report was filed.`” and “`Police report was NOT filed.`” based on the results. Or, display a special message if a property is `null`. For example, for a doctor name field, display “`No Doctor`” if there is no doctor.

Form field groups are implemented as a `FormFieldGroup` element composed of one or more `FormField` elements. A descriptor file can have any number of `FormFieldGroup` elements. Typically, the Gosu in the `FormField` tags refer to a context object defined earlier in the file (see “Template Descriptor Fields and Context Objects” on page 242).

You can group display values for multiple fields by defining a `DisplayValues` tag within the `FormFieldGroup`. You can specify only one value (for example, `NullDisplayValue`) or some other smaller subset of values; you do not need to specify all possible display values. The possible choices are `NullDisplayValue` (if `null`), `TrueDisplayValue` (if `true`), and `FalseDisplay` (if `false`).

For Gosu expressions with subobjects that are pure entity path expressions and any object in the path evaluates to `null`, the expression silently evaluates to `null`. For example, if `Obj` has the value `null`, then `Obj.SubObject.Subsubobject` always evaluates to `null`. Consequently, the `NullDisplayValue` is a simple way of displaying something better if any part of the a multi-step field path expression is `null`.

BillingCenter also uses the `NullDisplayValue` if an invalid array index is encountered. For example, the expression “`MyEntity.doctor[0].DisplayName`” results in displaying the `NullDisplayValue` if the `doctor` array is empty.

However, this approach does not work with method invocations on a `null` expression. For example, the expression `Obj.SubObject.Field1()` throws an exception if `Obj` is `null`.

In addition to checking for specific values, the `DisplayValues` tag can contain a `NumberFormat` tag, and either a `DateFormat` tag or a `TimeFormat` tag. These tags allow the user to specify a number format (such as “`###,##.##`”) or Date format (such as “`MM/dd/yyyy`”). The number formats modify the display of all form field values that are numbers or dates. This is useful in cases in which every number value in a form must be formatted in a particular way. Specify the format just once rather than repeatedly. Number format codes must contain the # symbol for each possible digit. Use any other character to separate the digits.

The BCAPI interface includes methods to add documents to BillingCenter objects:

- `addDocumentToAccount` – Add a new Document to an existing Account

- `addDocumentToPolicyPeriod` – Add a new Document to an existing PolicyPeriod
- `addDocumentToProducer` – Add a new Document to an existing Producer

Customization of the XML Descriptor Mechanism

If the default XML-based template descriptor mechanism is used, the set of attributes can still be modified to suit your needs. To extend the set of attributes on document templates, a few steps are required.

First, modify the `document-template.xsd` file, or create a new one. The location of the `.xsd` file used to validate the document is specified by the `DocumentTemplateDescriptorXSDLocation` parameter within the application `config.xml` file. This location is specified relative to the `WEB_APPLICATION/WEB-INF/platform` directory in the deployed web application directory.

Any number of attributes can be added to the definition of the `DocumentTemplateDescriptor` element. This is the only element which can be modified in this file, and the only legal way in which it can be modified. For example, you could add an attribute named `myattribute` as shown in bold in the following example:

```
<xsd:element name="DocumentTemplateDescriptor">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="ContextObject" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="HtmlTable" minOccurs="0" maxOccurs="unbounded" />
      <xsd:element ref="FormFieldGroup" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="identifier" type="xsd:string" use="optional"/>
    <xsd:attribute name="scope" use="optional">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="all"/>
          <xsd:enumeration value="gosu"/>
          <xsd:enumeration value="ui"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="password" type="xsd:string" use="optional"/>
    <xsd:attribute name="description" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="lob" type="xsd:string" use="required"/>
    <xsd:attribute name="myattribute" type="xsd:string" use="optional"/>
    <xsd:attribute name="section" type="xsd:string" use="optional"/>
    <xsd:attribute name="state" type="xsd:string" use="required"/>
    <xsd:attribute name="mime-type" type="xsd:string" use="required"/>
    <xsd:attribute name="date-modified" type="xsd:string" use="optional"/>
    <xsd:attribute name="date-effective" type="xsd:string" use="required"/>
    <xsd:attribute name="date-expires" type="xsd:string" use="required"/>
    <xsd:attribute name="keywords" type="xsd:string" use="required"/>
    <xsd:attribute name="required-permission" type="xsd:string" use="optional"/>
    <xsd:attribute name="default-security-type" type="xsd:string" use="optional"/>
  </xsd:complexType>
</xsd:element>
```

Next, enable the user to search on the `myattribute` attribute and see the results. Add an item with the same name (`myattribute`) to the PCF files for the document template search criteria and results. See the “Using the PCF Editor” on page 269 in the *Configuration Guide* for more information about PCF configuration.

Now the new `myattribute` property shows up in the template search dialog. The search criteria processing happens in the `IDocumentTemplateSource` implementation. The default implementation, `LocalDocumentTemplateSource`, automatically handles new attributes by attempting an exact match of the attribute value from the search criteria. If the specified value in the descriptor XML file contains commas, it splits the value on the commas and tries to match any of the resulting values.

For example, if the value in the XML is `test`, then only a search for "test" or a search that not specifying a value for that attribute finds the template. If the value in the XML file is "test,hello,purple", then a search for any of "test", "hello", or "purple" finds that template.

Finally, once BillingCenter creates the merged document, it tries to match attributes on the document template with properties on the document entity. For each match found, BillingCenter copies the value of the attribute in the template descriptor to the newly created Document entity. The user can either accept the default or change it to review the newly-created document.

Template Source Reference Implementation

The BillingCenter reference implementation keeps a set of document templates on the BillingCenter application server. The document templates reside initially in the directory:

`BillingCenter/modules/configuration/config/resources/doctemplates`

At run time, the document templates are typically at the path:

`SERVER/webapps/bc/modules/configuration/config/resources/doctemplates`

For each template, there are two files:

| File | Naming | Example | Purpose |
|-------------------------|-------------------------------|-----------------------|---|
| The template | Normal file name | "Test.doc" | Contains the text of the letter, plus named fields that fill in with data from BillingCenter. |
| The template descriptor | Template name + ".descriptor" | "Test.doc.descriptor" | This XML-formatted file provides various information about the template: <ul style="list-style-type: none">• How to search and find this template• Form fields that define what information populates each merge field• Form fields that define context objects• Form fields that define root objects to merge |

To set up a new form or letter, you must create a template file and a template descriptor file. Then, deploy them to the `templates` directory on the web server.

See also

- For details on the XML formatting of template descriptor files, see:
 - “Document Template Descriptors” on page 239
 - “XML Format of Built-in IDocumentTemplateSerializer” on page 245.

Document Template Descriptor Optional Cache

By default, BillingCenter calculates the list of document templates from files locally on disk each time the application needs them. If you have only a small list of document templates, this is a quick process. However, if you have a large number of document templates, you can tell BillingCenter to cache the list for better performance.

You might prefer to use the default behavior (no caching) during development, particularly if you are frequently changing templates while the application is running. However, for production, set the optional parameter in the document template source plugin to cache the list of templates.

To enable document template descriptor caching

1. In Project window in Guidewire Studio, navigate to Configuration → config → Plugins → registry, and then open `IDocumentTemplateSource.gwp`.
2. Under the plugin parameters editor in the right pane, add the `cacheDescriptors` parameter with the value `true`.

Built-in Document Production Plugins

Each application uses a different mechanism for defining fields to be filled in by the document production plugins. The following table describes the built-in document production plugins and their associated formats:

| Application or format | Description of built-in template |
|-----------------------|---|
| Microsoft Word | The document production plugin takes advantage of Microsoft Word's built-in mail merge functionality to generate a new document from a template and a descriptor file. BillingCenter includes with a sample Reservation of Rights Word document and an associated CSV file. That CSV file is present so that you can manually open the Word template. BillingCenter does not require or use this document. |
| Adobe Acrobat (PDF) | PDF file production happens on the server and requires a license to server-based software. See "Server-side PDF Licensing" on page 238. |
| Microsoft Excel | The document production plugin takes advantage of Microsoft Excel's built-in named fields functionality. The names in the Excel spreadsheet must match the <code>FormField</code> names in the template descriptor. After the merge, the values of the named cells become the values extracted from the descriptor file. |
| Gosu template | <p>Gosu provides you with a sophisticated way to generate any kind of text file. This includes plain text, RTF, CSV, HTML, or any other text-based format.</p> <p>The document production plugin retrieves the template and uses the built-in Gosu language to populate the document from Gosu code and values defined in the template descriptor file. Based on the file's MIME type and the local computer's settings, the system opens the resultant document in the appropriate application. See "Data Extraction Integration" on page 441.</p> |

Generating Documents from Gosu

BillingCenter can generate PDF documents and Gosu-based forms without user intervention from Gosu business rules or from any other place Gosu runs.

BillingCenter automatic form generation does not support any client-side creation such as Microsoft Word and Microsoft Excel templates. In client-side creation, the Document Assistant generates the new document locally. On Windows, the Document Assistant uses JScript to control local applications on the client PC. At the time business rules run, there may or may not be a user manipulating the application user interface. Because there may be no client to run the client-side code, client-side document production is impossible for automatic document creation.

However, server-side creation is possible using text formats:

- You can convert Word documents convert to the text-based RTF format
- You can convert many Excel templates to CSV files.

Therefore, you can convert most Microsoft application documents into equivalent Gosu templates that can generate text in RTF or CSV format. You can also use Gosu to generate any other text-based format, most notably plain text and HTML.

The automatic form generation process is very similar to the manual document generation process, but effectively skips some steps previously described for manual form generation.

The following list describes differences in automatic form generation compared to steps in "User Interface Flow for Document Production" on page 232. See that section for the step numbers:

- **Automatic template choosing** – Because template choosing is automatic, this effectively skips step 1 through step 6. Instead, you specify the appropriate template and parameter information within Gosu code.
- **No user editing** – Since there is no user intervention, there is no optional step for user intervention within the middle of this flow of step 9
- **No user uploading** – Because the Gosu code is executed on the BillingCenter server, there is no user uploading step, which is step 11 in that section.

To create a document, first create a map of values that specifies the value for each context object. Using `java.util.HashMap` is recommended, but any `Map` type is legal. This value map must be non-null. The values in this map are unconstrained by either the default object value or the possible object values. Be careful to pass valid objects of the correct type.

Within business rules, the Gosu class that handles document production is called `DocumentProduction`. It has methods for synchronous or asynchronous creation, which call the respective synchronous or asynchronous methods of the appropriate document production plugin, `createDocumentSynchronously` or `createDocumentAsynchronously`. Additionally there is a method to store the document: `createAndStoreDocumentSynchronously`. You can modify this Gosu class as needed.

If synchronous document creation fails, the `DocumentProduction` class throws an exception, which can be caught and handled appropriately by the caller. If document storage errors happen later, such as for asynchronous document storage, the document content storage plugin must handle errors appropriately. For example, the plugin could send administrative e-mails or create new activities using SOAP APIs to investigate the issues. Or you could design a system to track document creation errors and document management problems in a separate user interface for administrators. In the latter case, the plugin could register any document creation errors with that system.

If the synchronous document creation succeeds, next your code must attach the document to the account by calling `myAccount.addDocument(theDocument)`.

IMPORTANT New documents always use the latest in-memory versions of entity instances at the time the rules run, not the versions as persisted in the database.

Be careful creating documents within Pre-Update business rules or in other cases where changes can be rolled back due to errors (Gosu exceptions) or validation problems. If errors occur that roll back the database transaction even though rules added a document to an external document management system, the externally-stored document is *orphaned*. The document exists in the external system but no BillingCenter persisted data links to it.

See also

- “Important Notes About Cached Document UIDs” on page 252.

Example Document Creation After Sending an Email

You can use code like the following to send a standard email and then create a corresponding document:

```
// First, construct the email
var toContact : Contact = myAccount.PrimaryPayer
var subject : String = "Email Subject"
var body : String = "Email Body"

// Next, actually *send* the email
gw.api.email.EmailUtil.sendEmailWithBody(myAccount, toContact, fromContact, subject, body)

// Next, create the document that records the email
var document : Document = new Document(myAccount)
myAccount.addDocument(theDocument)
document.Name = "Create by a Rule"
document.Type = "letter_sent"
document.Status = "draft"
// ...perhaps add more property settings here

// Create some "context objects"
var parameters = new java.util.HashMap()
parameters.put("To", toContact)
parameters.put("From", fromContact)
parameters.put("Subject", subject)
parameters.put("Body", body)
parameters.put("RelatedTo", myAccount)
parameters.put("Account", myAccount)

// Create and store the document using the context objects
DocumentProduction.createAndStoreDocumentSynchronously("EmailSent.gosu.htm", parameters, document)
```

This particular example assumes that the document production plugin for that template supports synchronous production. This is true for all built-in document production plugins but not necessarily for all document production plugins. The calling code must either know in advance whether the document production plugin for that template type supports synchronous and/or asynchronous creation, or checks beforehand. If necessary, you can check which types of production are supported with code such as:

```
var plugin : IDocumentProduction;  
plugin = PluginRegistry.getPluginRegistry().getPlugin(IDocumentProduction) as IDocumentProduction;  
if (plugin.synchronousCreationSupported(templateDescriptor)) {  
    ...  
}
```

For more examples of creating documents from Gosu, see “Document Creation” on page 63 in the *Rules Guide*.

Important Notes About Cached Document UIDs

If a new document is created and an error occurs within the same database transaction, the error typically causes the database transaction to roll back. This means that no database data was changed. However, if the local BillingCenter transaction rolls back, there is no stored reference in the BillingCenter database to the document unique ID (UID). The UID describes the location of the document in the external system. This information is stored in the Document entity in BillingCenter in the same transaction, so the Document entity was not committed to the database. The new document in the external system is *orphaned*, and additional attempts to change BillingCenter data regenerates a new, duplicate version of the document.

For the common case of validation errors, BillingCenter avoids this problem. If a validatable entity fails validation, BillingCenter saves the document UID in local memory. If the user fixes the validation error in that user session, BillingCenter adds the document information as expected so no externally-stored documents are orphaned.

However, if other errors occur that cause the transaction to roll back (such as uncaught Gosu exceptions), externally-stored documents associated with the current transaction could be orphaned. The document is stored externally but the BillingCenter database contains no Document entity that references the document UID for it. Avoiding orphaned documents is a good reason to ensure your Gosu rules properly catches exceptions and handles errors elegantly and thoroughly. Write good error-handling code and logging code always, but particularly carefully in document production code.

Encryption Integration

You can store certain data model properties encrypted in the database. For example, you can hide important data, such as bank account numbers or private personal data, by storing the data in the database in a non-plaintext format. This topic is about how to integrate BillingCenter with your own encryption code.

This topic includes:

- “Encryption Integration Overview” on page 253
- “Changing Your Encryption Algorithm Later” on page 258
- “Encryption Features for Staging Tables” on page 259

Encryption Integration Overview

You can store certain data model properties encrypted in the database. For example, hide important data, such as bank account numbers or private personal data, in the database in a non-plaintext format. The actual encryption and decryption is implemented through the `IEncryption` plugin. You can define your own algorithm.

BillingCenter does not provide an encryption algorithm in the product. BillingCenter simply calls this plugin implementation, which is responsible for encoding an unencrypted `String` or reversing that process.

IMPORTANT BillingCenter provides a sample encryption algorithm that simply reverses the string value. You must disable this plugin if you do not wish to use encryption. You must implement the plugin to properly encrypt your data with your own algorithm.

All encryption and decryption is done within BillingCenter automatically as the application accesses and loads entities from the database. All Gosu rules and web services automatically operate on plain text (unencrypted text) without special coding to support encryption within rules, web services, and messaging plugins.

From Gosu, any encrypted fields appear unencrypted. If you must avoid sending this information as plain text to external systems, you must design your own integrations to use a secure protocol or encrypt that data.

For example, use the plugin registry and use your own encryption plugin to encrypt and decrypt data across the wire in your integration code. Create classes to contain your integration data for each integration point. You can make some properties contain encrypted versions of data that require extra security between systems. Such properties do not need to be database-backed. You can implement enhancement properties on entities that dynamically return the encrypted version. If your messaging layer uses encryption (for example, SSL/HTTPS) or is on a secure network, additional encryption might not be necessary. It depends on the details of your security requirements.

WARNING From Gosu, any encrypted fields appear unencrypted. Carefully consider security implications of any integrations with external systems for properties that normally are encrypted.

For communication between Guidewire applications, the built-in integrations do not encrypt any encrypted properties during the web services interaction. This affects the following integrations:

- PolicyCenter and BillingCenter

IMPORTANT The built-in integrations between Guidewire applications do not encrypt properties in the web service connection between the applications. To add additional security, you must customize the integration to use HTTPS or add additional encryption of properties sent across the network.

Setting Encrypted Properties

You can change the encryption settings for a column in data model files by overriding the column information (adding a *column override*). You can only set encryption for character-based column types such as `varchar`. Encryption is unsupported on other types, including binary types (`varbinary`) and date types.

There are two ways to mark the column as encrypted:

- Make a column encrypted by specifying its type as `encrypted`:

```
<column name="encrypted_column" type="encrypted" size="10"/>
```

This makes the column `encrypted_column` have the type `EncryptedString`. BillingCenter generates visible masks for all input widgets for this value.

- Alternatively, mark a column as encrypted through the `encryption` attribute, which only applies to types based on `varchar`. This masks values that are accessed directly as entity properties. For example:

```
<column name="encrypted_column" type="varchar" size="10" encryption="true"/>
```

This example makes the property `encrypted_column` to be the type `String`. In this case, input widgets for this value are masked only if the value is a entity property access. For example, suppose the earlier column is within an entity. BillingCenter generates an input mask if the value expression is the form `entity.encrypted_column`. However, there is no input mask if the expression is a method invocation like `pageHelper.getMyEncryptedColumnValue()`.

BillingCenter prevents you from encrypting properties with a denormalized column. In other words, no encryption on a property that creates a secondary column to support case-insensitive search. For example, the contact property `LastnameDenorm` is a denormalized column added to mirror the `Lastname` property, and thus the `Lastname` property cannot be encrypted.

Querying Encrypted Properties

Gosu database query builder queries (and older-style `find` expressions) work as expected if you compare the property with a literal value. For example, looking up a social security number or other unique ID is comparing with a literal value. However, the encrypted data for every record is not decrypted from the database to test the results of the query. It actually works in the opposite way. If you compare a constant value against an encrypted property, BillingCenter encrypts the constant in the SQL query. The query embeds the equality comparison logic directly into the SQL.

For comparison logic, the SQL that Gosu generates always compares either:

- Two properties from the database, either both encrypted or both unencrypted
- A static literal, encrypted if necessary, and a property from the database.

One side effect of this is that equality comparison is the only supported comparison for encrypted properties. For example you cannot use “greater than” comparisons or “less than” comparisons.

This means that your `find` queries can compare encrypted properties against other values in only two ways:

- Direct equality comparison (`==`) or not equals (`!=`) a Gosu expression of type of `String`
- Direct equality comparison (`==`) or not equals (`!=`) with a field path to another encrypted property

For example, suppose `Account.SecretVal` and `Account.SecretVal2` are encrypted properties, suppose `Account.OtherVal` and `Account.OtherVal2` are unencrypted properties, and suppose `tempValue` is a local variable containing a `String`.

The following queries work:

```
q.compare("SecretVal", Equals, "123")
q.compare("SecretVal", NotEquals, "123")
q.compare("SecretVal", Equals, tempValue)
q.compare("SecretVal", NotEquals, tempValue)
q.compare("SecretVal", Equals, q.getColumnRef("SecretVal2")) // both encrypted
q.compare("SecretVal", NotEquals, q.getColumnRef("SecretVal2")) // both encrypted
q.compare("OtherVal", Equals, q.getColumnRef("OtherVal2")) // both unencrypted
q.compare("OtherVal", NotEquals, q.getColumnRef("OtherVal2")) // both unencrypted
```

The following queries do not work:

```
q.compare("SecretVal", Equals, q.getColumnRef("OtherVal")) // bad mix of encrypted and unencrypted
q.compare("SecretVal", NotEquals, q.getColumnRef("OtherVal")) // bad mix of encrypted and unencrypted
q.compare("SecretVal", GreaterThan, "123") // no greater than or less than
```

IMPORTANT If a Gosu `find` query involves encrypted properties, you can use equality comparison with other encrypted properties or against `String` values constants in the query. However, there are limitations with what you can do in the query. Refer to the instructions in this section for details.

Writing Your Encryption Plugin

Write and implement a class that implements the `IEncryption` plugin interface. Its responsibility is to encrypt and decrypt data with one encryption algorithm.

To encrypt, implement an `encrypt` method that takes an unencrypted `String` and returns an encrypted `String`, which may or may not be a different length than the original text. If you want to use strong encryption and are permitted to do so legally, you are free to do so. BillingCenter does not include any actual encryption algorithm.

To decrypt, implement a `decrypt` method that takes an encrypted `String` and returns the original unencrypted `String`.

You must also specify the maximum length of the encrypted string by implementing the `getEncryptedLength` method. Its argument is the length of decrypted data. It must return the maximum length of the encrypted data. BillingCenter primarily uses the encryption length at application startup time during upgrades. During the upgrade process, the application must determine the required length of encrypted columns. If the length of the column must increase to accommodate inflation of encrypted data, then this method helps BillingCenter know how far to increase space for the database column.

To uniquely identify your encryption algorithm, your plugin must return an encryption ID. Implement a `getEncryptionIdentifier` method to return a unique identifier for your encryption algorithm. This identifier tracks encryption-related change control. This is exposed to Gosu as the property `EncryptionIdentifier`:

```
override property get EncryptionIdentifier() : String {
    return "ABC:DES3"
}
```

The encryption ID is very important and must be unique among all encryption plugins in your implementation. The application decides whether to upgrade the encryption data with a new algorithm by comparing:

- The encryption ID of the current encryption plugin
- The encryption ID associated with the database last time the server ran.

For important details, see “[Changing Your Encryption Algorithm Later](#)” on page 258.

IMPORTANT Be careful that your encryption plugin returns an appropriate encryption ID and that it is unique among all your encryption plugins.

The following example is a simple demonstration encryption plugin. It simple fake encryption algorithm is to append a reversed version of the unencrypted text to the end of the text. For example, encrypting the text "hello" becomes "helloolleh".

To decrypt the text, it merely removes the second half of the string. The second half of the string is the reversed part of the text that it appended earlier when encrypting the string. It is important to note that this fake algorithm doubles the size of the encrypted data, hence the `getEncryptedLength` doubles and returns the input size argument.

The following Gosu code implements this plugin

```
package Plugins
uses gw.plugin.util.IEncryption
uses java.lang.StringBuilder

class EncryptionByReversePlugin implements IEncryption {

    override function encrypt(value:String) : String {
        return reverse(value)
    }

    override function decrypt(value:String) : String {
        return value.subString(value.length() / 2, value.length());
    }

    override function getEncryptedLength(size:int) : int {
        return size * 2 // encrypting doubles the size
    }

    override property get EncryptionIdentifier() : String {
        return "mycompany:reverse"
    }

    private function reverse(value:String) : String {
        var buffer = new StringBuilder(value)
        return buffer.reverse().append(value).toString()
    }
}
```

Detecting Accidental Duplication of Encryption or Decryption

You can mitigate the risk of accidentally encrypting already-encrypted data if you design your encryption algorithm to authoritatively detect whether the property data is already encrypted.

For example, suppose you put a known series of special unusual characters that could not appear in the data both before and after your encrypted data. You can now detect whether data is encrypted or unencrypted:

- During an encryption request, if your encryption plugin detects that the data is already encrypted, throw an exception.
- During an decryption request, if your encryption plugin detects that the data is already decrypted, throw an exception.

If You Import Data from Staging Tables

If you import data from staging tables and use encrypted columns, there are special tools that you need to know about. See “Encryption Features for Staging Tables” on page 259.

Installing Your Encryption Plugin

After you write your implementation of the `IEncryption` plugin, you must register your plugin implementation. You can register multiple `IEncryption` plugin implementations if you need to support changing the encryption algorithm.

To enable your encryption plugin implementation

1. Create a new class that implements the `IEncryption` plugin interface. Be certain that your `EncryptionIdentifier` method returns a unique encryption identifier. If you change your encryption algorithm, it is critical that you change the encryption identifier for your new implementation.

IMPORTANT Guidewire strongly recommends you set the encryption ID for your current encryption plugin to a name that describes or names the algorithm itself. For example, "encryptDES3".

2. In the Project window in Studio, navigate to `configuration` → `config` → `Plugins` → `registry`.
3. Right-click on `registry`, and choose `New` → `Plugin`.
4. Studio prompts you to name your new plugin. Remember that you can have than one registered implementation of an `IEncryption` plugin interface. The name field must be unique among all your plugins. This is called the *plugin name* and is particularly important for encryption plugins.

IMPORTANT Guidewire strongly recommends you set the plugin name for your current encryption plugin to a name that describes the algorithm itself. For example, `encryptDES3`.

5. In the interface field, type `IEncryption`.
6. Edit standard plugin fields in the Plugins editor in Studio. See “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*. Also see “Plugin Overview” on page 135.
7. In `config.xml`, set the `CurrentEncryptionPlugin` parameter to the plugin name.

The `CurrentEncryptionPlugin` parameter specifies which encryption plugin is the current encryption algorithm for the main database. Specify the plugin name, not the class name nor the encryption ID.

WARNING If the `CurrentEncryptionPlugin` parameter is missing or specifies an implementation that does not exist, the server does not start.

8. Start the server.

If the upgrader detects data model fields marked as encrypted but the database contains unencrypted versions, the upgrader encrypts the field in the main database using the current encryption plugin.

See also

“Changing Your Encryption Algorithm Later” on page 258

Adding or Removing Encrypted Properties

If you later add or remove encrypted properties in the data model, upgrader automatically runs on server startup to update the main database to the new data model.

Changing Your Encryption Algorithm Later

You can register any number of `IEncryption` plugins. For an original upgrade of your database to a new encryption algorithm, you register two implementations at the same time.

However, only one encryption plugin is the *current encryption plugin*. The `config.xml` configuration parameter `CurrentEncryptionPlugin` controls this setting. It specifies which encryption plugin, among potentially multiple implementations, is the current encryption algorithm for the main database. Set the parameter to the plugin name, not the class name nor the encryption ID, for the current encryption plugin.

Note: When you use the Plugins editor in Studio to add an implementation of `IEncryption`, Studio prompts you for a text value to use as the plugin name for this implementation. Guidewire strongly recommends you set the plugin name for encryption plugins to names that describe the algorithm. For example, "encryptDES3" or "encryptRSA128". Any legacy encryption plugins (if you did not originally enter a name) have the name "`IEncryption`".

During server startup, the upgrader checks the encryption ID of data in the main database. The server compares this encryption ID with the encryption ID associated with the current encryption plugin. If the encryption IDs are different, the upgrader decrypts encrypted fields with the old encryption plugin, found by its encryption ID. Next, the server encrypts the fields to be encrypted with the new encryption plugin, found by its plugin name as specified by the parameter `CurrentEncryptionPlugin`.

The most important things to remember whenever you change encryption algorithms are:

- All encryption plugins must return their appropriate encryption IDs correctly.
- All encryption plugins must implement `getEncryptedLength` correctly.
- You must set `CurrentEncryptionPlugin` to the correct plugin name.

The server uses an internal lookup table to map all previously used encryption IDs to an incrementing integer value. This value is stored with database data. Internally, the upgrader manages this lookup table to determine whether data needs to be upgraded to the latest encryption algorithm. Do not attempt to manage this table directly. Instead, assure every encryption plugin returns its appropriate encryption ID, and assure `CurrentEncryptionPlugin` specifies the correct plugin name.

Be careful not to confuse the encryption ID of a plugin implementation with its plugin name or class name. The server relies on the encryption ID saved with the database and the encryption ID of the current encryption plugin to identify whether the encryption algorithm changed.

Changing Your Encryption Algorithm

The following procedure describes how to change your encryption algorithm. It is extremely important to follow it exactly and very carefully. If you have questions about this before doing it, contact Guidewire Professional Services before proceeding.

WARNING Do not follow this procedure until you are sure you understand it and test your encryption algorithm code. Before proceeding, be confident of your encryption code, particularly your implementation of the plugin method `getEncryptedLength`. Failure to perform this procedure correctly risks data corruption.

1. Shut down your server.
2. Register a new plugin implementation of the `IEncryption` plugin for your new algorithm. When you add an implementation of the plugin in Studio, it prompts you for a plugin name for your new implementation. Name it appropriately to match the algorithm. For example, "encryptDES3".
3. Be sure your plugin returns an appropriate and unique encryption ID. Name it appropriately to match the algorithm. For example, "encryptDES3".

4. Set the config.xml configuration parameter CurrentEncryptionPlugin to the plugin name of your new encryption plugin.
5. Start the server. The upgrader uses the old encryption plugin to decrypt your data and then encrypts it with the new algorithm.

See also

- “Writing Your Encryption Plugin” on page 255

Encryption Features for Staging Tables

If you need to import records using staging tables and any data has encrypted properties, you must encrypt those properties before importing them into operational tables. To support this feature, there is a special tool that encrypts any encrypted fields in staging tables before import. This tool encrypts only the columns in the data model marked as encrypted.

During upgrades, be careful about the order that you make data model modifications. Before beginning staging table import work, do any data model changes including changing encryption settings. Next, let the server upgrade the database. After you modify the data model and increment the data model version number, BillingCenter automatically encrypts data in the operational tables during server launch as part of upgrade routines. During upgrade, BillingCenter drops (deletes) all staging tables.

Guidewire strongly recommends you encrypt your staging table data after you complete all work on your data conversion tools and your data passes all integrity checks. Note that staging table integrity checks never depend on the encryption status of encrypted properties. You can run integrity checks independent of whether the encrypted properties are currently encrypted.

There are several ways to encrypt your data:

- Synchronously with web services using `TableImportAPI.encryptDataOnStagingTables()`. This method takes no arguments and returns nothing.
- Synchronously with the command line tool `table_import` with the option `-encryptstagingtbls`. For more information, see “Table Import Command” on page 195 in the *System Administration Guide*.
- Asynchronously with web services using the `TableImportAPI` web service method `encryptDataOnStagingTablesAsBatchProcess`. This method takes no arguments and returns a process ID (`ProcessID`). Use the returned process ID to check the status of the encryption batch process or to terminate the encryption batch process. Use this process ID with methods on the web service `MaintenanceToolsAPI`. See “Maintenance Tools Web Service” on page 128 for details.

To encrypt any encrypted properties in staging tables

1. Write and register an encryption plugin as discussed in “Encryption Integration Overview” on page 253. Carefully upgrade your existing production data after your data model change.
2. Perform integrity checks until they all pass. During development of conversion programs that populate the staging tables, you may need to repeatedly run integrity checks and modify your code.
3. Use one of the web service APIs or command line tools that encrypt columns in staging tables, as discussed earlier in this topic.

WARNING If the encryption process succeeds, be careful not to run the process a second time because you would corrupt your data by encrypting already-encrypted data. You can mitigate this problem by careful design of your encryption code. See “Detecting Accidental Duplication of Encryption or Decryption” on page 256

If the encryption process fails in any way, no changes are committed to the database. All changes happen in one database transaction. If an exception or other error occurs, partial work is undone. You can safely repeat the process after you fix any errors. For example, if the encryption plugin changes the length of the data field, this process could fail at the database level and throw an exception, which rolls back all changes. Be careful when fixing problems. See the earlier discussion about performing data model changes and upgrades before beginning encryption work.

4. After the encryption process succeeds, use regular staging table loading web service APIs or command line tools. For example, the web service method `integrityCheckStagingTableContentsAndLoadSourceTables` or the `table_import` command line tool. These methods run integrity checks again. Integrity checks do not behave differently with encrypted properties.
5. If you need to change your encryption algorithm later, see “Changing Your Encryption Algorithm Later” on page 258.

For related information, see these sections:

- For general information about staging table import, see “Importing from Database Staging Tables” on page 367.
- If you need to change your encryption algorithm after data is encrypted, see “Changing Your Encryption Algorithm Later” on page 258.
- “Table Import Command” on page 195 in the *System Administration Guide*

Management Integration

This topic discusses *management*, which is a special type of API that allows external code to control BillingCenter in certain ways through the JMX protocol or some similar protocol. Types of control include viewing or changing cache sizes and modifying configuration parameters dynamically. This topic focuses on writing a new management plugin.

This topic includes:

- “Management Integration Overview” on page 261
- “The Abstract Management Plugin Interface” on page 262
- “Integrating With the Included JMX Management Plugin” on page 263

See also

- For more information about plugins, see “Plugin Overview” on page 135.
- For the complete list of all BillingCenter plugins, see “Summary of All BillingCenter Plugins” on page 153.

Management Integration Overview

BillingCenter provides APIs to view and change some BillingCenter settings from remote management consoles or APIs from remote integration code:

- **Configuration parameters** – View the values of all configuration parameters, and change the value of certain parameters. These changes take effect in the server immediately, without requiring the server to restart. After the server shuts down, BillingCenter discards dynamic changes like this. After the server starts next, the server rereads parameters in the `config.xml` configuration file.
- **Batch processes** – View batch processes currently running (if any).
- **Users** – View the number of current user sessions and their user names
- **Database connections** – View the number of active and idle database connections
- **Notifications** – View notifications about locking out users due to excessive login failures.

BillingCenter exposes this data through three mechanisms:

- **User interface in Server Tools tab, on the Management Beans page** – All of these items are exposed in the user interface on the Management Beans page of the BillingCenter Server Tools tab. This tab is accessible only to users with the soapadmin permission. For more information, see “Management Beans” on page 171 in the *System Administration Guide*.
- **Abstract management plugin interface** – BillingCenter also provides an abstract interface called the *management plugin*. This plugin exposes *management beans* to the external world. Management beans are interfaces to read or control system settings. For example, the management plugin could expose the management beans using the Java Management Extension (JMX) protocol, using the Simple Network Management Protocol (SNMP), or something else entirely.
- **A supported JMX management plugin for Apache Tomcat** – BillingCenter includes an example plugin that implements JMX using the management plugin. This plugin is compiled and automatically installed in `BillingCenter/modules/configuration/plugins` so that you need only to enable it in Studio. The included JMX management plugin works only with the Apache Tomcat web server. The source code for this example is included in at this path:

`BillingCenter\java-api\examples\src\examples\plugins\management`

The Abstract Management Plugin Interface

The following table lists the important interfaces and classes used by the abstract management plugin interface.

| Interface or class | Description |
|---|--|
| <code>ManagementPlugin</code> | The management interface for Guidewire systems. This is the interface that the included JMX plugin implements. |
| <code>GWMBean</code> | The interface for managed beans exposed by the system. The management plugin must expose these internal management beans to the outside world using JMX, SNMP, or some other management interface. |
| <code>GWMBeanInfo</code> | Meta-information about a Guidewire managed bean (<code>GWMBean</code>), such as its name and description. |
| <code>ManagementAuthorizationCallbackHandler</code> | A callback handler interface that BillingCenter invokes if a user attempts management operations. |
| <code>NotificationSenderMarker</code> | A marker interface that indicates that a <code>GWMBean</code> can send notifications. |
| <code>Attribute</code> | The <code>Attribute</code> class encapsulates two strings that represent a system attribute name and its value. |
| <code>AttributeInfo</code> | Metadata about an <code>Attribute</code> , such as name, description, type, and whether the attribute is readable and/or writable. |
| <code>Notification</code> | A BillingCenter notification, such as those sent if BillingCenter locks out a user due to too many failed login attempts. |
| <code>NotificationInfo</code> | Metadata about a notification, such as a message string, a sequence number, and a notification type. |

The main task of the management plugin is to register `GWMBean` objects. Registering the object means that the plugin provides that service to the outside world in whatever way makes sense for that service. For example, the management plugin might create a wrapper for the `GWMBean` and expose it using JMX or SNMP using its own published service.

For details, refer to these Java source files in the directory `BillingCenter/java-api/examples`:

- `JMXManagementPlugin.java`
- `GWMBeanWrapper.java`
- `JMXAuthenticatorImpl.java`
- `JSR160Connector.java`

See also

- To write code for an external system to communicate with the built-in JMX management plugin, see “Integrating With the Included JMX Management Plugin” on page 263.

Integrating With the Included JMX Management Plugin

BillingCenter includes an example plugin that implements JMX using the management plugin interface. This plugin is compiled and automatically installed in `BillingCenter/modules/configuration/plugins` so that you need only to change one setting in the Plugins Registry editor in Studio to enable it.

In the Project window in Studio, select **Configuration** → **config** → **Plugins** → **registry**, and then open `ManagementPlugin.gwp`. Select the **Enabled** checkbox. This supported JMX management plugin works only with the Apache Tomcat web server. The source code for this example is included as part of the BillingCenter examples directory, at the following path:

```
BillingCenter\java-api\examples\src\examples\plugins\management
```

Note: JMX support differs between web servers, but the JMX management plugin is designed for and supported in Guidewire production environments that use Apache Tomcat only. The JMX management plugin is supported for jetty in development environments. Modifications to the source code and use of different JMX libraries could enable the management plugin for other web servers. Guidewire provides the source code in case such changes are desired.

After you enable the JMX plugin, it publishes a JMX service called a *JMX JSR160 connector* under Apache Tomcat. A remote management console or your integration code can call the JMX JSR160 connector by creating *JMX connector client* code that connects to the published service.

The following example demonstrates a simple Java application that can retrieve a system attribute programmatically from a remote system. This example relies on the following conditions:

- The BillingCenter server must be running under the Apache Tomcat web server.
- The JMX plugin must be installed (which it is by default).
- The JMX plugin must be enabled in the Plugins Registry editor

This example retrieves the `HolidayList` system attribute programmatically from a remote system:

```
package com.mycompany.bc.integration.jmx;

import java.util.HashMap;
import java.util.Map;

import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.naming.Context;

// Create a "JMX JSR160 connector" to connect to BillingCenter
//
public class TestJMXClientConnector {

    public static void main(String[] args) {

        try {
            // The address of the connector server
            JMXServiceURL address = new JMXServiceURL(
                "service:jmx:rmi:///jndi/rmi://akitio:1099/jrmp");

            // The creation environment map, null in this case
            Map creationEnvironment = null;

            // Create the JMXConnectorServer
            JMXConnector cctor = JMXConnectorFactory.newJMXConnector(address,
                creationEnvironment);
        }
    }
}
```

```
// May contain - for example - user's credentials
Map environment = new HashMap();
environment.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.rmi.registry.RegistryContextFactory");
environment.put(Context.PROVIDER_URL, "rmi://localhost:1099");
String[] credentials = new String[]{"su", "cc"};
environment.put(JMXConnector.CREDENTIALS, credentials);

// Connect
cntor.connect(environment);

// Obtain a "stub" for the remote MBeanServer
MBeanServerConnection mbsc = cntor.getMBeanServerConnection();

// Call the remote MBeanServer
String domain = mbsc.getDefaultDomain();
ObjectName delegate =
    ObjectName.getInstance("com.guidewire.pl.system.configuration:type=configuration");
String holidayList = (String)mbsc.getAttribute(delegate, "HolidayList");
System.out.println(holidayList);
}
catch (Exception e) {
    e.printStackTrace();
}
}
}
```

Other Plugin Interfaces

Plugins are software modules that BillingCenter calls to perform an action or calculate a result. This topic describes plugin interfaces that are not discussed in detail elsewhere in this documentation.

This topic includes:

- “Automatic Address Completion and Fill-in Plugin” on page 265
- “Phone Number Normalizer Plugin” on page 266
- “Testing Clock Plugin (Only For Non-Production Servers)” on page 266
- “Work Item Priority Plugin” on page 268
- “Official IDs Mapped to Tax IDs Plugin” on page 268
- “Preupdate Handler Plugin” on page 268
- “Defining Base URLs for Fully-Qualified Domain Names” on page 270
- “Exception and Escalation Plugins” on page 271

See also

- For general information about plugins, see “Plugin Overview” on page 135.
- For messaging plugins, see “Messaging and Events” on page 303.
- For authentication plugins, see “Authentication Integration” on page 207.
- For document and form plugins, see “Document Management” on page 217.

Automatic Address Completion and Fill-in Plugin

To customize automatic address completion, you can create a class that implements the `IAddressAutocompletePlugin` plugin interface.

In the base configuration, the class `DefaultAddressAutocompletePlugin` implements this interface. This class provides the default behavior, which uses `address-config.xml` and `zone-config.xml` files.

You can write your own plugin implementation if you want to handle address auto-completion and auto-fill differently. For example, you might want to access an address data service directly instead of having to import zone data files.

See the Javadoc for `IAddressAutocompletePlugin` for more information on this plugin interface.

See also

- “Address Autocompletion and Autofill” on page 143 in the *Globalization Guide*
- “Configuring Zone Information” on page 126 in the *Globalization Guide*

Phone Number Normalizer Plugin

To support automatic address completion, implement the `IPhoneNormalizerPlugin` plugin interface.

There is a default implementation called `DefaultPhoneNormalizerPlugin`, which handles the default behavior.

Contact Guidewire Customer Support for more information.

See also

- “DefaultNANPACountryCode” on page 50 in the *Configuration Guide*

Testing Clock Plugin (Only For Non-Production Servers)

WARNING The `ITestingClock` plugin is supported only for testing on non-production development servers. Do not register this plugin on production servers.

Testing BillingCenter behavior over a long span of time during multiple billing cycles can be challenging. For testing on development servers only, you can implement the `ITestingClock` plugin and programmatically change the system time to simulate the passing of time. For example, you can define a plugin that returns the real time except in special cases in which you artificially increase the time to represent a time delay. The delay could be one week, one month, or one year.

This plugin interface has only two methods, `getCurrentTime` and `setCurrentTime`, which get and set the current time using the standard BillingCenter format of milliseconds stored in a `long` integer.

If you cannot set the time in the `setCurrentTime` function, for example if you are using an external “time server” and it temporarily cannot be reached, throw the exception `java.lang.IllegalArgumentException`.

Time must always increase, not go back in time. Going back in time is likely to cause unpredictable behavior in BillingCenter.

Notes:

- The plugin method `setCurrentTime` advances the time only for `gw.api.util.DateUtil.currentTimeMillis`. The class `java.util.Date` always states the server time. Therefore, you must use `gw.api.util.DateUtil.currentTimeMillis` in your implementation code.
- The advanced date does not change in all parts of the product. There are certain areas that intentionally do not use the rolled-over date. For example, the time shown in the next scheduled run for batch processes is not affected.
- Today’s date on the `Calendar` does not change.

Using the Testing Clock Plugin

The base application has an implementation of the `ITestingClock` plugin interface. This topic shows you how to use it.

1. Start BillingCenter Studio.
2. In the Project window, navigate to `configuration` → `config` → `Plugins` → `registry`.
3. Right-click `registry` and click `New` → `New Plugin`.
4. In the `Name` field, type `ITestingClock`.
5. In the `Interface` field, type `ITestingClock`.
6. Click `OK`.
7. In the Plugins Registry editor, click `Add plugin` , and then choose `Java Plugin`.
8. In the `Java class` field, type the following:
`com.guidewire.pl.plugin.system.internal.OffsetTestingClock`
9. In the Project window, navigate to `configuration` → `config`, and then open `config.xml`.
10. In `config.xml` under `Environment Parameters`, set `EnableInternalDebugTools` to `true`. If you see a message asking if you want to edit the file, click `Yes`.

```
<!-- Enable internal debug tools page http://localhost:8080/app/InternalTools.do -->
<param name="EnableInternalDebugTools" value="true"/>
```
11. Save your changes.
12. Stop then restart BillingCenter.
13. Log in as user `su` with password `gw`.
14. To advance the clock a fixed number of days, enter the following text in the `Quick Jump` field on the upper right and then click `Go`. Change the number of days from 10 to the desired number of days as needed:
`Run Clock addDays 10`
In the yellow message area near the top of the screen, a message shows the new current date.
15. To advance the clock using a picker, type `Alt+Shift+T`, and then click the `Internal Tools` tab. In the sidebar, click `Testing System Clock`. Set the clock as needed by clicking one of the `Add` buttons or entering a specific date and time. Finally, click `Change Date`.

Testing Clock Plugin in BillingCenter Clusters

If you are operating a cluster of BillingCenter servers, you must use the following procedure to change the time.

To change the testing clock time for BillingCenter clusters

1. Stop all servers with the exception of the *batch server*.
2. Advance the testing clock.
3. Restart all the cluster nodes.

Work Item Priority Plugin

Configure how BillingCenter calculates work item priority for workflow steps by implementing the Work Item Priority plugin (`IWorkItemPriorityPlugin`). The interface has a single method, `getWorkItemPriority`, which takes a `WorkflowWorkItem` parameter. Your method implementation returns a priority as a non-negative integer. A higher priority integer indicates workflow steps to process before other workflow steps with lower priorities. If there is no plugin implementation, the default workflow step priority is zero.

IMPORTANT Prioritization affects only work items of type `WorkflowWorkItem` or its derivatives.

Official IDs Mapped to Tax IDs Plugin

Contacts in the real world have many official IDs. However, only one of a contact's official IDs is used as a tax ID. In the default BillingCenter data model, each `Contact` instance has a single `TaxID` field. Also, each `Contact` instance has an `OfficialIDs` array field. The array holds all sorts of official IDs for a contact, including the contact's official tax ID.

You configure BillingCenter with official ID types in the `OfficialID` typelist. For example, the base configuration includes type keys for the U.S. Social Security Number (SSN) and the U.S. Federal Employer Identification Number (FEIN). The `OfficialIDs` array on a `Contact` instance contains instances of the `OfficialID` entity type, which has a field for the `OfficialIDType` typekey of the instance.

Create a Gosu implementation of the `OfficialIdToTaxIdMappingPlugin` interface to configure which typekeys from the `OfficialID` typelist you want to treat as official tax IDs. The plugin interface has one method, `isTaxId`, which takes a typekey from the `OfficialIDType` typelist (`oIdType`). The method returns `true` if you want to treat that official ID type as a tax ID. The default Java implementation that BillingCenter provides always returns `false`.

The following sample Gosu implementation of the `OfficialIdToTaxIdMappingPlugin` interface returns `true` for official ID typekeys that have SSN or FEIN as codes.

```
uses gw.plugin.contact.OfficialIdToTaxIdMappingPlugin

class MyOfficialIdToTaxIdMappingPlugin implements OfficialIdToTaxIdMappingPlugin {

    /**
     * Return true if the official ID type is either an SSN or a FEIN.
     */
    override function isTaxId(oIdType : OfficialIDType) : boolean {
        return OfficialIDType.TC_SSN == oIdType or OfficialIDType.TC_FEIN == oIdType
    }

}
```

Preupdate Handler Plugin

In some cases it makes sense to implement preupdate handling in plugin code rather than in a rule set. To implement preupdate handling in plugin code, register an implementation of the `IPreUpdateHandler` plugin interface.

It is best to use `IPreUpdateHandler` plugin instead of Preupdate Rules in the following cases:

- **If you need to create entity instances** – If you have tasks that must create new entity instances, the plugin approach is best. The application calls the `IPreUpdateHandler` plugin before calling the rule set. The Preupdate rules run on all new entities created by the `IPreUpdateHandler` plugin implementation. You can separate your code into tasks that create objects in the plugin, and other tasks in the rule set. Without using the plugin, a Preupdate rule that creates a new entity must incorporate the logic in Preupdate rules that explicitly check the new entity instances also.
- **If you need to handle removed or deleted entity instances** – Similarly, this plugin is good for tasks that must deal with removed objects, which are entity instances that BillingCenter will delete or retire on commit. A Preupdate rule set for a particular entity type does not run simply for removal of the deleted object. Only add or change of an entity instance triggers Preupdate and Validation rule sets for that entity type. However, the `IPreUpdateHandler` has direct access to removed entity instances in the bundle. Therefore, it is best that work that involves removed objects, and optionally new and changed objects, happen in the `IPreUpdateHandler` plugin implementation.
- **If you need to process entity instances within the bundle in a particular order** – Use the plugin to process entity instances in the bundle in any order. For example, sort entity instances by entity type or other sorting criteria.

Registering and enabling this plugin is sufficient for BillingCenter to call the plugin for preupdate, independent of the server `config.xml` parameter `UseOldStylePreUpdate`.

Next, if the server `config.xml` parameter `UseOldStylePreUpdate` is set to `true`, BillingCenter additionally calls the validation-graph based Preupdate rule set after calling the plugin.

Your plugin implementation must implement one method, `executePreUpdate`. This method takes a single preupdate context object, `PreUpdateContext`, and returns nothing.

The `PreUpdateContext` object has several properties you can get, which return a list (`java.util.List`) of entities in the current database transaction.

From Gosu they look like the following properties:

- `InsertedBeans` – An unordered list of entities added in this transaction.
- `UpdatedBeans` – An unordered list of entities changed in this transaction.
- `RemovedBeans` – An unordered list of entities added in this transaction.

From Java, these properties appear as three methods: `getInsertedBeans`, `getUpdatedBeans`, and `getRemovedBeans`.

For an overview of preupdate rules, see “Preupdate” on page 38 in the *Rules Guide*

Default Plugin Implementation

In the base configuration, BillingCenter provides the plugin implementation `gw.plugin.preupdate.impl.PreUpdateHandlerImpl`.

By default, this plugin implementation class collects all inserted and updated beans, as well as accounts and jobs if any assignments changed.

Any exception cause the bounding database transaction to roll back, effectively undoing the update.

PreUpdateUtil

It is possible to maintain some preupdate code in this plugin but still call preupdate rules in some cases.

BillingCenter includes the `PreUpdateUtil` class in the `gw.api.preupdate` package to let you call preupdate rules from your plugin. The `PreUpdateUtil` class provides a single method, `executePreUpdateRules`. That method executes preupdate rules on an entity if it has an associated preupdate rule set. It does nothing if a preupdate rule does not exist.

Defining Base URLs for Fully-Qualified Domain Names

If BillingCenter generates HTML pages, it typically generates a *base URL* for the HTML page using a `<base href="...">` at the top of the page. In almost all cases, BillingCenter generates the most appropriate base URL, based on settings in `config.xml`.

In some cases, this behavior is inappropriate. For example, suppose you hide BillingCenter behind a load balancing router that handles Secure Socket Layer (SSL) communication. In such a case, the external URL would include the prefix `https://`. The load balancer handles security and forwards a non-secure HTTP request to BillingCenter with a URL prefix `http://`. The default implementation of the base URL includes the URL prefix `http://`.

The load balancer would not typically parse the HTML enough to know about this problem, so the base URL at the user starts with `http` instead of `https`. This breaks image loading and display because the browser tries to load the images relative to the `http` URL. The load balancer rejects the requests because they are insecure because they do not use HTTPS/SSL.

Avoid this problem by writing a custom base URL builder plugin (`IBaseURLBuilder`) plugin and registering it with the system.

You can base your implementation on the built-in example implementation found at the path:

```
BillingCenter/java-api/examples/src/examples/plugins/baseurlbuilder
```

To handle the load balancer case mentioned earlier, the base URL builder plugin can look at the HTTP request's header. If a property that you designate exists to indicate that the request came from the load balancer, return a base URL with the prefix `https` instead of `http`.

The built-in plugin implementation provides a parameter `FqdnForUrlRewrite` which is not set by default. If you enable browser-side integration features, you must specify this parameter to rewrite the URL for the external fully-qualified domain name (FQDN). The JavaScript security model prevents access across different domains. Therefore, if BillingCenter and other third-party applications are installed on different hosts, the URLs must contain fully-qualified domain names. The fully-qualified domain name must be in the same domain. If the `FqdnForUrlRewrite` parameter is not set, the end user is responsible for entering a URL with a fully-qualified domain name.

There is another parameter called `auto` which tries to auto-configure the domain name. This setting is not recommended for clustering environments. For example, do not use this if the web server and application server are not on the same machine, or if multiple virtual hosts live in the same machine. In these cases, it is unlikely for the plugin to figure out the fully-qualified domain name automatically.

In Project window in Studio, navigate to `configuration` → `config` → `Plugins` → `registry`, and the open `IBaseURLBuilder`. Add the parameter `FqdnForUrlRewrite` with the value of your domain name, such as "`mycompany.com`". The domain name must specify the Fully Qualified Domain Name to be enforced in the URL. If the value is set to "`auto`", the default plugin implementation makes the best effort to calculate the server FQDN from the underlying configuration.

Implement `IBaseURLBuilder` and `InitializablePlugin`

Your `IBaseURLBuilder` plugin must explicitly implement `InitializablePlugin` in the class definition. Otherwise, BillingCenter does not initialize your plugin.

For example, suppose your plugin implementation's first line looks like this:

```
class MyURLBuilder implements IBaseURLBuilder {
```

Change it to this:

```
class MyURLBuilder implements IBaseURLBuilder, InitializablePlugin {
```

To conform to the new interface, you must also implement a `setParameters` method even if you do not need parameters from the plugin registry:

```
function setParameters(map: java.util.Map) { // this is part of InitializablePlugin
```

```
// access values in the MAP to get parameters defined in plugin registry in Studio  
}
```

Exception and Escalation Plugins

There are several optional exception and escalation plugins. By default, they just call the associated rule sets and perform no other function. Implement your own version of the plugin and register it in Studio if you want something other than the default behavior.

Use these plugins for the following tasks:

- Add additional logic before or after calling the rule set definitions in Studio
- Completely replace the logic of the rule set definitions in Studio.

One reason you might want to completely replace the logic of the rule set definitions in Studio is to make your code more easily tested using unit tests.

The following table lists the exception and escalation plugins:

| Name | Plugin interface | Default action |
|---------------------|---------------------------|--|
| Activity escalation | IActivityEscalationPlugin | Calls the activity escalation rule set |
| Group exceptions | IGroupExceptionPlugin | Calls the group exception rule set |
| User exceptions | IUserExceptionPlugin | Calls the user exception rule set |

Startable Plugins

A *startable plugin* is a special type of code that runs without human intervention in the application server as a background process, beginning at server startup.

This topic includes:

- “Startable Plugins Overview” on page 273
- “Writing a Startable Plugin” on page 274
- “Configuring Startable Plugins to Run on All Servers” on page 277
- “Java and Startable Plugins” on page 280
- “Persistence and Startable Plugins” on page 280

See also

- For an alternative mechanism to startable plugins, see “Developing Custom Batch Processes” on page 420.

Startable Plugins Overview

You can register custom code that runs at server startup in the form of a startable plugin implementation. You can use a startable plugin as a daemon, such as listener to a JMS queue. You can use a startable plugin for periodic batch processing, such as deleting expired files from a file system folder. You can use a startable plugin as an initializer, such as loading configuration data into the application database.

You can start and stop startable plugins as circumstances require. You cannot start or stop standard types of plugins. Instead, code in a standard Guidewire plugin executes only when other code invokes its methods.

Startable Plugins as Background Processes

Typically, a startable plugin behaves as a daemon, such as a listener that runs continuously as a background process. Startable plugins have similarities with the messaging reply plugin. A messaging transport's main task is to send a message to an external system and listen for the reply acknowledgment. This metaphor might not apply to integrations that do not need to send outgoing messages about data changes using the event and messaging system. However, if you need listener code and it must initialize with server startup, use a startable plugin.

Note: Some integrations require external code to trigger an action within BillingCenter but do not need custom listener code. Depending on your needs, consider publishing a BillingCenter web service instead of a startable plugin. For more information, see “Web Services Introduction” on page 27.

Registering Startable Plugins

Register your startable plugin in the Plugins Registry in Studio, just like you register standard plugins. In the Project window, navigate to **configuration** → **config** → **Plugins** → **registry**. All startable plugin implementations that you register in the Plugins Registry and that are enabled are listed on the **Server Tools** → **Startable Plugin** page.

See also

- For details on how to register a startable plugin, see “Working with Plugins” on page 110 in the *Configuration Guide*.

Starting, Stopping, and Managing Startable Plugins

If you have administration privileges, you can view the operational status of registered startable plugins on the **Server Tools** → **Startable Plugin** page. In the Actions column, use the **Start** and **Stop** buttons to start and stop startable plugins. You can modify the PCF files for the **Startable Plugin** page to show additional information about your startable plugins, their underlying transport mechanisms, or any other information.

See also

- For information on using a web service to start and stop startable plugins, see “Stopping Startable Plugins Using Web Services” on page 129.

Startable Plugins in a Clustered Configuration

By default for a cluster, startable plugins operate only on the batch server. However, you can configure a startable plugin to run on all servers in the cluster.

See also

- “Configuring Startable Plugins to Run on All Servers” on page 277

Writing a Startable Plugin

Most of your code of a startable plugin implements your custom listener code or other special service. Write a new class that implements the **IStartablePlugin** interface and implement the following methods:

- A **start** method to start your service
- A **stop** method to stop your service
- A property accessor function to get the **State** property from your startable plugin.

This method returns a typecode from the typelist **StartablePluginState**: the value **Stopped** or **Started**. The administration user interface uses this property accessor to show the state to the user. Define a private variable to hold the current state and your property accessor (get) function can look simple:

```
// private variables...
```

```
var _state = StartablePluginState.Stopped;  
...  
// property accessor (get) function...  
override property get State() : StartablePluginState {  
    return _state // return our private variable  
}
```

Alternatively, combine the variable definition with the shortcut keyword to simplify your code. You can combine the variable definition with the property definition can be combined with the single variable definition:

```
var _state : StartablePluginState as State
```

The plugin does include a constructor called on system startup. However, start your service code in the `start` method, not the constructor. Your start method must appropriately set the state (started or stopped) using an internal variable that you define.

At minimum, your start method must set your internal variable that tracks your started or stopped state, with code such as:

```
_state = Started
```

Additionally, in your `start` method, start any listener code or threads such as a JMS queue listener.

Your `start` method has a method parameter that is a callback handler of type `StartablePluginCallbackHandler`. This callback is important if your startable plugin modifies any entity data, which is likely the case for most real-world startable plugins.

Your plugin must run any code that affects entity data within a code block that you pass to the callback handler method called `execute`. The `execute` method takes as its argument a *Gosu block*, which is a special type of in-line function. The Gosu block you pass to the `execute` method takes no arguments. For more information, see “Gosu Blocks” on page 239 in the *Gosu Reference Guide*.

Note: If you do not need a user context, use the simplest version of the callback handler method `execute`, whose one argument is the Gosu block. To run your code as a specific user, see “User Contexts for Startable Plugins” on page 276.

You do not need to create a separate bundle. If you create new entities to the current bundle, they are in the correct (default) database transaction the application sets up for you. Use the code `Transaction.getCurrent()` to get the current bundle if you need a reference to the current (writable) bundle. For more information, see “Bundles and Database Transactions” on page 341 in the *Gosu Reference Guide*.

IMPORTANT The Java language does not directly support blocks. If you implement your plugin in Java, you cannot use a Gosu block but you can use an anonymous class to do the same thing. For more information, see “Defining Startable Plugins In Java” on page 280.

The plugin’s `start` method also includes a boolean variable that indicates whether the server is starting. If `true`, the server is starting up. If `false`, the start request comes from the `Server Tools` user interface.

The following shows a simple Gosu block that changes entity data. This example assumes your listener defined a variable `messageBody` with information from your remote system. If you get entities from a database query, remember to add them to the current bundle. Refer to “Updating Entity Instances in Query Results” on page 174 in the *Gosu Reference Guide* for more information about bundles.

This simple example queries all `User` entities and sets a property on results of the query:

```
//variable definition earlier in your class...  
var _callback : StartablePluginCallbackHandler;  
...  
override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) : void  
{  
    _callback = cbh  
    _callback.execute( \ -> {  
        var q = gw.api.database.Query.make(User) // run a query
```

```

var b = gw.Transaction.Transaction.Current // get the current bundle
for (e in q.select()) {
    // add entity instance to writable bundle, then save and only modify that result
    var writable_object = bundle.add(e)

    // modify properties as desired on the result of bundle.add(e)
    writable_object.Department = "Example of setting a property on a writable entity instance."
}

//Note: You do not need to commit the bundle here. The execute method commits after your block runs.
}

```

Note that to make an entity instance writable, save and use the return result of the bundle add method. For more information, see “Adding Entity Instances to Bundles” on page 345 in the *Gosu Reference Guide*.

Just like your `start` method must set your internal variable that sets its state to started, your `stop` method must set your internal state variable to `StartablePluginState.Stopped`. Additionally, stop whatever background processes or listeners you started in your `start` method. For example, if your `start` method creates a new JMS queue listener, your `stop` method destroys the listener. Similar to the `start` method’s `isStartingUp` parameter, the `stop` method includes a boolean variable that indicates whether the server is shutting down now. If `true`, the server is shutting down. If `false`, the stop request comes from the `Server Tools` user interface.

User Contexts for Startable Plugins

If you use the simplest method signature for the `execute` method on `StartablePluginCallbackHandler`, your code does not run with a current BillingCenter user. Any code that directly or indirectly runs due to this plugin (including preupdate rules or any other code) must be prepared for the current user to be `null`. You must not rely on non-null values for current user if you use this version.

However, there are alternate method signatures for the `execute` method. Use these to perform your startable plugin tasks as a specific User. Depending on the method variant, pass either a user name or the actual `User` entity.

On a related note, the `gw.transaction.Transaction` class has an alternate version of the `runWithNewBundle` method to create a bundle with a specific user associated with it. You can use this in contexts in which there is no built-in user context or you need to use different users for different parts of your tasks. The method signature is:

```
gw.transaction.Transaction.runWithNewBundle(\ bundle -> YOUR_BLOCK_BODY, user)
```

For the second method argument to `runWithNewBundle`, pass either a `User` entity or a `String` that is the user name.

Simple Startable Plugin Example

The following is a complete simple startable plugin. This example does not do anything useful in its `start` method, but demonstrates the basic structure of creating a block that you pass to the callback handler’s `execute` method:

```

package gw.api.startableplugin
uses gw.api.startable.IStartablePlugin
uses gw.api.startable.StartablePluginCallbackHandler
uses gw.api.startable.StartablePluginState

class HelloWorldStartablePlugin implements IStartablePlugin
{
    var _state = StartablePluginState.Stopped;
    var _callback : StartablePluginCallbackHandler;

    construct()
    {

    }

    override property get State() : StartablePluginState
    {
        return _state
    }
}

```

```
override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) : void
{
    _callback = cbh
    _callback.execute( \ -> {
        // Do some work:
        // [...]
    } )
    _state = Started
    _callback.log( "**** From HelloWorldStartablePlugin: Hello world." )
}

override function stop( isShuttingDown: boolean ) : void
{
    _callback.log( "**** From HelloWorldStartablePlugin: Goodbye." )
    _callback = null
    _state = Stopped
}
```

Startable Plugins and Run Levels

On server startup, by default BillingCenter starts all startable plugins when the server gets to the `maintenance` run level, which is also called `nodaemons` in some APIs.

If you want your startable plugin to start at an earlier or later run level, add the annotation `@gw.api.server.Availability` on the plugin implementation class declaration. Pass a run level (`gw.api.server.AvailabilityLevel`) as an argument to the annotation constructor.

For example:

```
uses gw.api.server.Availability
uses gw.api.server.AvailabilityLevel

@Availability(AvailabilityLevel.MULTIUSER)
class HelloWorldStartablePlugin implements IStartablePlugin {

    ...
}
```

This annotation is meaningful for startable plugins but not other types of plugins. Other types of plugins are instantiated and initialized on first usage, independent of the run level.

Configuring Startable Plugins to Run on All Servers

By default for a cluster, startable plugins run only on the batch server. However, you can configure them to run on all servers in the cluster by declaring the startable plugin is *distributed*. To declare the plugin is distributed, add the `@Distributed` annotation to the startable plugin implementation class.

Guidewire strongly recommends that startable plugins very clearly and consistently save the state (started or stopped) across all servers in a cluster. This approach handles edge cases like a server joining the cluster late after other servers have started. To keep the state consistent across all servers, the state must persist in the database. Even though the state information in the database is not automatically persisted, the broadcast to other servers happens automatically.

The series of steps is as follows:

1. On any server, the server starts. The plugin method reads the second argument to the `start` method. Because the server is starting up, this value is `true`. The plugin implementation detects this condition and reads the database status of the started or stopped state and sets its own state appropriately.
2. Later, the state may change on a server to start or stop from the user interface or through APIs. The application propagates this information to all servers in the cluster. Each server in the cluster decides whether it is appropriate to respect this request:

- If the startable plugin is distributed, the server respects the request and calls the plugin `start` or `stop` method as appropriate.
- If the startable plugin is not distributed, only the batch server handles the request.

However, if the startable plugin is not distributed and the original changed server is the batch server, the behavior is different. The batch server does not bother with the distributed broadcast system because it is the only running version of the startable plugin. Instead, that server immediately calls the plugin `start` or `stop` method as appropriate.

3. On any server, the server shuts down. The plugin method reads the second argument to the `stop` method. Because the server is shutting down, this value is `true`. The plugin implementation detects this condition and simply stops its local state but does not set the database status in the database.

Implement getting and setting the state information in the database using the plugin callback handler object, which is an instance of `StartablePluginCallbackHandler`. This object is a parameter to the `start` method of the plugin. Save a copy of this object as a private variable and then call the following methods on the object as needed:

- `getState` – Gets the state information in the database: `true` for started, `false` for stopped. If this is the first time in history that this startable plugin has ever run on this server-database cluster combination, its state information is not in the database. This method takes one arguments that represents the default assumption to use the very first time this query is checked. For example, if you expect this startable plugin to always run, pass the value `Started`. If this startable plugin runs only in rare situations and you expect only an administrator or API to trigger it to start, pass the value `Stopped`. If the server state has ever been set in the database, `getState` ignores this argument.
- `setState` – Sets the state information in the database for this startable plugin. This method takes one argument, which is true if and only if the startable plugin is running. This request attempts to set this information in the database if it is not yet set to the expected value. If the startable plugin state is already matching the database value for the state according to the `getState` method, do not call `setState`. Be careful to catch any exceptions. You must create your own bundle to make this change, see the example for how to use the `runWithNewBundle` API.
- `logStart` – Writes a line in the log about the plugin starting. This method takes one argument, which is line to log, as a `String`. Include the plugin name and the state, such as `"HelloWorldDistributedStartablePlugin:Started"`.
- `logStop` – Writes a line in the log about the plugin stopping. This method takes one argument, which is line to log, as a `String`. Include the plugin name and the state, such as `"HelloWorldDistributedStartablePlugin:Stopped"`.

The following is an example implementation of the `start` and `stop` methods that implement a trivial thread and the proper setting of startable plugin state.

```
package gw.api.startableplugin
uses gw.api.startable.IStartablePlugin
uses gw.api.startable.StartablePluginCallbackHandler
uses gw.api.startable.StartablePluginState
uses gw.transaction.Transaction
uses java.lang.Thread
uses gw.api.util.Logger

@Distributed
class HelloWorldDistributedStartablePlugin implements IStartablePlugin {
    var _state : StartablePluginState;
    var _startedHowManyTimes = 0
    var _callback : StartablePluginCallbackHandler;
    var _thread : Thread

    override property get State() : StartablePluginState {
        return _state
    }

    override function start( handler : StartablePluginCallbackHandler, serverStartup: boolean ) : void {
        _callback = handler
        if (serverStartup) {
            // if the server is starting up, read the value from the database, and default to stopped
        }
    }
}
```

```
// if this plugin NEVER saved its state before. You might want to change this to Started instead.
_state = _callback.getState(Stopped)

if (_state == Started) {
    _callback.logStart("HelloWorldDistributedStartablePlugin:Started")
}
else {
    _callback.logStart("HelloWorldDistributedStartablePlugin:Stopped")
}
else {
    _state = Started
    if (_callback.State != Started) {
        changeState(Started) // call our internal function that sets the database state if necessary
    }
    _callback.logStart("HelloWorldDistributedStartablePlugin")
}

// if the thread already existed for some reason, briefly stop it before starting it again
if (_state == Started) {
    if (_thread != null) {
        _thread.stop()
    }
    _startedHowManyTimes++
}

// create your thread and start it
var t = new Thread() {
    function run() {
        print("hello!")
    }
}
t.Daemon=true

}

override function stop( serverStopping : boolean ) : void {
    if (_thread != null) {
        _thread.stop()
        _thread = null
    }
    if (_callback != null) {
        if (serverStopping) {
            if (_state == Started) {
                _callback.logStop("HelloWorldDistributedStartablePlugin:Started")
            }
            else {
                _callback.logStop("HelloWorldDistributedStartablePlugin:Stopped")
            }
            _callback = null
        }
        else {
            if (_callback.State != Stopped) {
                changeState(Stopped) // call our internal function that sets the database state if necessary
            }
            _callback.logStop("HelloWorldDistributedStartablePlugin")
        }
    }
    _state = Stopped
}

// our internal function that sets the database state if necessary. if there are exceptions,
// this implementation tries 5 times. You might want a different behavior.
private function changeState(newState : StartablePluginState) {
    var tryCount = 0
    while (_callback.State != newState && tryCount < 5) {
        try {
            Transaction.runWithNewBundle(\ bundle -> { _callback.setState(bundle, newState)},
                User.util.UnrestrictedUser)
        }
        catch (e : java.lang.Exception) {
            tryCount++
            _callback.log(this.IntrinsicType.Name + " on attempt " + tryCount +
                " caught " + (typeof e).Name + ":" + e.Message)
        }
    }
}
```

Java and Startable Plugins

You can develop your custom startable plugin in Java, but special considerations apply.

Defining Startable Plugins In Java

In Gosu, your startable plugin must call the `execute` method on the callback handler object, as discussed in previous topics:

```
override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) : void {  
    _callback = cbh  
    _callback.execute( \ -> {  
        //...  
    }  
}
```

However, the Java language does not directly support blocks. If you implement your plugin in Java, you cannot use a Gosu block. However, instead you can use an anonymous class.

From Java, the method signatures for the `execute` methods (there are multiple variants) take a `GWRunnable` for the block argument. `GWRunnable` is a simple interface that contains a single method, called `run`. Instead of using a block, you can define an in-line anonymous Java class that implements the `run` method. This is analogous to the standard Java design pattern for creating an anonymous class to use the standard class `java.lang.Runnable`.

For example:

```
GWRunnable myBlock=new GWRunnable() {  
    public void run() {  
        System.out.println("I am startable plugin code running in an anonymous inner class");  
        // add more code here...  
    }  
  
    _callbackHandler.execute(myBlock);
```

For information about Gosu blocks and inner classes, see “Gosu Block Shortcut for Anonymous Classes Implementing an Interface” on page 215 in the *Gosu Reference Guide*.

Location of Java Files for Startable Plugins

If you have Java files for your startable plugin, place your Java class and libraries files in the same places as with other plugin types.

The instructions are slightly different depending on whether you define the plugin interface implementation itself in Java or in Gosu:

- If your main startable plugin class is a Java class, see “Special Notes For Java Plugins” on page 142.
- If your main startable plugin class is a Gosu class, see “If Your Gosu Plugin Needs Java Classes and Library Files” on page 142.

Persistence and Startable Plugins

Your startable plugin can manipulate Guidewire entity data. If your startable plugin needs to maintain state for itself, do one of the following:

- Create your own custom persistent entity types that track the internal state information of your startable plugin.
- Use the system parameter table for persistence.

Multi-threaded Inbound Integration

BillingCenter provides a plugin interface that supports high performance multi-threaded processing of inbound requests. BillingCenter includes default implementations for the most common usages: reading text file data and receiving JMS messages.

This topic includes:

- “Multi-threaded Inbound Integration Overview” on page 281
- “Inbound Integration Configuration XML File” on page 283
- “Inbound File Integration” on page 285
- “Inbound JMS Integration” on page 290
- “Custom Inbound Integrations” on page 292
- “Understanding the Polling Interval and Throttle Interval” on page 299

Multi-threaded Inbound Integration Overview

There are sometimes situations that require high-performance data throughput for inbound integrations that require special threading or transaction features from the hosting J2EE/JEE application environment. It is difficult to interact with the application server’s transactional facilities and write correct, thread-safe, high-performing code. BillingCenter includes tools that help you write such inbound integrations. You can focus on your own business logic rather than how to write thread-safe code that works safely in each application server.

Inbound Integration Configuration XML File

There is a configuration file for inbound integrations called `inbound-integration-config.xml`. Edit this file in Studio to define configuration settings for every inbound integration that you want to use. Each inbound integration requires configuration parameters such as references to the registered plugin implementations. See “Inbound Integration Configuration XML File” on page 283.

Inbound Integration Core Plugin Interfaces

BillingCenter provides the multi-threaded inbound integration system in two different plugin interfaces for different use cases. Each defines a contract between BillingCenter and inbound integration high-performance multi-threaded processing of input data:

- `IInboundIntegrationMessageReply` – Inbound high-performance multi-threaded processing of replies to messages sent by a `MessageTransport` implementation. This is a subinterface of `MessageReply`.
- `IInboundIntegrationStartablePlugin` – Inbound high-performance multi-threaded processing of input data as a startable plugin. A startable plugin is for all contexts other than handling replies to messages sent by a message transport (`MessageTransport`) implementation. This is a subinterface of `IStartablePlugin`.

You might not need to write your own implementation of these main plugin interfaces. BillingCenter includes plugin implementations that are supported for production servers that support common use cases. Both are provided in variants for message reply and startable plugin use.

| Type of input data | Description | Related topic |
|---------------------------|--|---|
| File inbound integrations | Use this integration to read text data from local files. Poll a directory in the local file system for new files at a specified interval. Send new files to integration code and process incoming files line by line, or file by file. See “Inbound File Integration” on page 285. You provide your own code that processes one chunk of work, either one line, or one file, depending on how you configure it. Your code is called a <i>handler</i> plugin. | “Inbound File Integration” on page 285 |
| JMS inbound integration | Use this integration to get objects from a JMS message queue. See “Inbound JMS Integration” on page 290. You provide your own code that processes the next message on the JMS message queue. Your code is called a <i>handler</i> plugin. | “Inbound JMS Integration” on page 290 |
| Custom integration | If you process incoming data other than files or JMS messages, write your own version of the <code>IInboundIntegrationMessageReply</code> or <code>IInboundIntegrationStartablePlugin</code> plugin interface. In both cases, for custom integrations you must write multiple classes that implement helper interfaces such as <code>WorkAgent</code> . | “Custom Inbound Integrations” on page 292 |

IMPORTANT You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (Java as an OSGi bundle). If you use Java or if you require third-party libraries, Guidewire recommends implementing your code as an OSGi plugin.

In all cases, you must register and configure plugin implementations in the Studio Plugins Registry. See each topic for more information about which implementation classes to register. Additionally, for file and JMS integrations, you write handler classes. See “Inbound Integration Handlers for File and JMS Integrations” on page 282.

When registering a plugin implementation in the Plugins Registry, you must add a plugin parameter called `integrationservice`. That `integrationservice` parameter defines how BillingCenter finds configuration information within the inbound integration configuration XML file.

To configure the inbound integration configuration XML file, see “Inbound Integration Configuration XML File” on page 283.

For general information about the Plugins Registry, see “Registering a Plugin Implementation Class” on page 139.

Inbound Integration Handlers for File and JMS Integrations

Whether you use the built-in integrations or write your own, you must write code that handles one chunk of data.

To write a custom integration that supports data other than files or JMS messages, your code primarily implements the interface `InboundIntegrationMessageReply` or `InboundIntegrationStartablePlugin`. For custom integrations, “Custom Inbound Integrations” on page 292 and skip the rest of this topic.

In contrast, if you use the built-in file or JMS inbound integrations, you register a built-in plugin implementation of `InboundIntegrationMessageReply` or `InboundIntegrationStartablePlugin`, depending on whether you are using messaging. BillingCenter includes plugin implementations of those interfaces that know how to process files or JMS data, depending on which one you choose.

For file or JMS inbound integrations, you must write a *handler class* that processes one chunk of data. BillingCenter defines a handler plugin interface called `InboundIntegrationHandlerPlugin` that contains one method called `process`. That method handles one chunk of data that BillingCenter passes as a method of type `java.lang.Object`. Write a handler class that implements the `process` method. Downcast the `Object` to the necessary type:

- for file handling, downcast to `java.nio.file.Path` or a `String`, depending how you configure the integration to process by files or by line
- for JMS handling, downcast to a JMS message of type `javax.jms.Message`

Next, register the plugin in the Studio Plugins Registry. See “Registering a Plugin Implementation Class” on page 139.

IMPORTANT When registering your plugin implementation, you must also add one plugin parameter called `integrationservice`. That plugin parameter links your plugin implementation to one XML element within the `inbound-integration-config.xml` file. See “Inbound Integration Configuration XML File” on page 283.

If you are using either the file or JMS integrations as the startable plugin variant, your class must implement the `InboundIntegrationHandlerPlugin` interface.

If you are using either the file or JMS integrations as the message reply variant, your class must implement the `InboundIntegrationMessageReplyHandler` interface. This is a subinterface of `InboundIntegrationHandlerPlugin`. Implement the basic `process` method as well as all methods of the `MessageReply` plugin. For example, implement `MessageReply` methods `initTools`, `suspend`, `shutdown`, `resume`, and `setDestinationID`. Save the parameters to your `initTools` method into private variables. Use those private variables during your `process` method find and acknowledge the original `Message` object. For related information, see “Implementing a Message Reply Plugin” on page 350.

You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (Java as an OSGi bundle). If you use Java or if you require third-party libraries, Guidewire recommends implementing your code as an OSGi plugin.

Inbound Integration Configuration XML File

In Studio, there is a configuration file for inbound integrations. In the Project window, navigate to `configuration` → `config` → `integration`, and the open `inbound-integration-config.xml`.

The file contains the following types of data:

- “Thread Pool Configuration” on page 283
- “Configuring a List of Inbound Integrations” on page 284

Thread Pool Configuration

The first section of the `inbound-integration-config.xml` file configures thread pools. Within the `<threadpools>` element, there is a list of `<threadpool>` elements, each of which look like the following

```
<threadpool name="gw_default" disabled="false">
  <gwthreadpooltype>FORKJOIN</gwthreadpooltype>
</threadpool>
```

The `name` attribute is a symbolic name that is used later in the XML file to refer uniquely to this thread pool.

The `disabled` attribute is a Boolean value that defines whether to disable that thread pool. If set to `true`, the thread pool is disabled.

Within the element, there are two supported element types for setting thread pool parameters:

- `<gwthreadpooltype>` – The thread pool type with the following supported values, which are case-sensitive:
 - Set to `FORKJOIN` for a self-managing default thread pool.
 - Set to `COMMONJ` for running WebSphere or WebLogic and defining the JNDI name of the thread pool. If you set this value, you must also set the `<workmanagerjndi>` parameter.
- `<workmanagerjndi>` – JNDI name of the thread pool. This parameter is required if the thread pool type is set to `COMMONJ`.

IMPORTANT In the default configuration, there are thread pools predefined for default WebSphere or WebLogic thread pools. These are for development but not production use. For best performance, create your own custom thread pool with a unique name and tune that thread pool for the specific work you need, such as your JMS work. Then, update the `<threadpool>` settings to include the JNDI name for your new thread pool.

Configuring a List of Inbound Integrations

The second section of the `inbound-integration-config.xml` file contains a list of inbound integrations that you want to use. For example, if you want five different JMS inbound integrations, each listening to its own JMS queue, add five elements to this section of the file.

Within the `<integrations>` element, there is a list of subelements of several pre-defined element names. For each inbound integration, define configuration parameters as subelements. For example, you must declare the name of the registered plugin implementations that corresponds to your handler code.

Some of the parameters are required, and some are optional. If you use either the built-in file processing or JMS integration, there are special parameters just for those integrations.

The following table lists the supported integration element names and where to find the complete reference for parameter names.

| Type of integration | Configuration element | For more information |
|---|---|---|
| File inbound integration | <code><file-integration></code> | "Inbound File Integration" on page 285 |
| JMS inbound integration | <code><jms-integration></code> | "Inbound JMS Integration" on page 290 |
| Custom inbound integration. Directly implement the <code>InboundIntegrationStartablePlugin</code> or <code>InboundIntegrationMessageReply</code> interface. | <code><custom-integration></code> | "Custom Inbound Integrations" on page 292 |

Your configuration XML element must have the following two attributes:

- `name` – A unique identifying name for this inbound integration in the `inbound-integration-config.xml` file.

IMPORTANT The `name` attribute must match the value of the `integrationservice` plugin parameter in the Plugins registry for all registered plugin implementations of any inbound integration interfaces. In the Plugins registry, add the plugin parameter `integrationservice` and set to the value of this unique identifying name. See "Registering a Plugin Implementation Class" on page 139 and "Using the Plugins Registry Editor" on page 109 in the *Configuration Guide*.

- `disabled` – Determines whether to disable this inbound integration. Set to `true` to disable the inbound integration. Otherwise, set to `false`.
- `env` – Sets a configurations that is valid only for a specific server environment. See “Varying Inbound Integration Settings” on page 285.

Varying Inbound Integration Settings

There are multiple ways you can vary configuration of inbound integration by system environment:

- **Within the inbound integration XML file** – Within your `inbound-integration-config.xml` file, each top-level element can have an `env` attribute. That attribute specifies that element is valid only for one value of the `env` system environment configuration setting. See “Defining the Application Server Environment” on page 14 in the *System Administration Guide*. For example, to use different JMS settings for development and for production, list two `<jms-integration>` elements. For one element, set the `env` attribute to `development`. For the other element, set the `env` attribute to `production`. For each element, set appropriate JMS configuration settings for that system environment.
- **Plugin property configuration in Plugins Registry** – In the Plugins Registry, you must set the `integrationservice` plugin property in the user interface in one or more Plugins Registry files. See “Configuring a List of Inbound Integrations” on page 284. Within a single Plugins Registry file, you can optionally define the `integrationservice` plugin property multiple times with values that vary by system environment (`env`) or server ID (`serverid`) values. See “Adding an Implementation to a Plugins Registry Item” on page 110 in the *Configuration Guide*.

You can use one or both of these techniques to vary the run time behavior of the server based on the server environment.

Note that within the elements in the `inbound-integration-config.xml` file, the `env` attribute is supported but you cannot vary the configuration by server ID. To do configuration by server ID, use the plugin property technique.

Inbound File Integration

BillingCenter includes built-in code that supports file-based input with high-performance multi-threaded processing. BillingCenter provides this code in two variants, one for processing message replies, and one as a startable plugin.

You cannot modify the plugin implementation code in Studio, but you can use one or more instances of these integrations to work with your own file data.

The processing flow for file integration is as follows:

1. In response to some inbound event, your own integration code creates a new file in a specified incoming directory on the local file system.
2. The inbound file integration code polls the *incoming directory* at a specified interval and detects any new files since the last time checked.

IMPORTANT Any files that are in the incoming directory before the plugin initializes are never processed. For example, if you restart the server, files created after the server shuts down but before initialization are never processed. To process files that already existed, wait until the plugin is initialized, then move files outside the *incoming directory*, and then back to the incoming directory again.

The order of the processing of individual files in the directory is undefined. Never rely on the order being any particular order, such as alphabetic or creation date.

3. The inbound file integration code moves all found files to the *processing directory*, which stores inbound files in progress.
 4. The inbound file integration code opens each new file using a specified character set. The default is UTF-8, but it is configurable.
 5. The inbound file integration code reads one unit of work (one chunk of data) and dispatches it to your handler code. The `processingmode` parameter in the `inbound-integration-config.xml` file defines the type of data processed in one unit of work. If that parameter has the value `line`, BillingCenter sends one line at a time to the handler as a `String` object. If that parameter has the value `file`, BillingCenter sends the entire file to the handler as a `java.nio.file.Path` object.
- If exceptions occur during processing, the plugin code moves the file to the *error directory*. The file name is changed to add a prefix that includes the time of the error, as expressed in milliseconds as returned from the Java time utilities. For example, if the file name `ABC.txt` has an error, it is renamed in the error directory with a name similar to `1864733246512.error.ABC.txt`.
6. After successfully reading and processing the complete file, the inbound file integration code moves the file to the *done directory*.
 7. If there were any other files detected in this polling interval in step 2, the inbound file integration code repeats the process at step 4. Optionally, you can set the integration to operate on the most recent batch of files in parallel. For related information, see the `ordered` parameter, mentioned later in this section.
 8. The inbound file integration waits until the next polling interval, and repeats this process at step 3.

To create an inbound file integration

1. In the Project window, navigate to `configuration` → `config` → `integration`, and the open `inbound-integration-config.xml`.
2. Configure the thread pools. See “Thread Pool Configuration” on page 283.
3. In the list of integrations, create one `<integration>` element of type `<file-integration>`. Follow the pattern in the file to set the XML element namespace. In the base configuration, type the element name as `<ci:i:file-integration>`. Set the `name` and `disabled` attributes as described in “Configuring a List of Inbound Integrations” on page 284.
4. Set configuration parameter subelements as follows:

| File integration configuration parameters | Required | Description | Example value |
|--|-----------------|--|--|
| <code>pluginhandler</code> | Required | The name in the Plugins registry for an implementation of the <code>InboundIntegrationHandlerPlugin</code> plugin interface. Note that this is the <code>.gwp</code> file name, not the implementation class name. | <code>InboundFileIntegrationExample</code> |
| <code>processingmode</code> | Required | To process one line at a time, set to <code>line</code> . In your handler class in the <code>process</code> method, you must downcast to <code>String</code> . To process one file at a time, set to <code>file</code> . In your handler class in the <code>process</code> method, you must downcast to <code>java.nio.file.Path</code> . | <code>line</code> |
| <code>threadpool</code> | Required | The unique name of a thread pool as configured earlier in the file. See “Thread Pool Configuration” on page 283. | <code>gw_default</code> |

| File integration configuration parameters | Required | Description | Example value |
|--|-----------------|--|--------------------------------------|
| osgiservice | Required | Always set to <code>true</code> . This is for internal use. This value is independent of whether you choose to register your handler class as an OSGi plugin. | <code>true</code> |
| transactional | Required | You must always set this to <code>false</code> . | <code>false</code> |
| createdirectories | Optional | If <code>true</code> , BillingCenter creates the incoming, processing, error, and done directories if they do not already exist. If errors that prevent creation of any directories, the server does not startup. If <code>false</code> , BillingCenter all of these directories must already exist. If any directories do not already exist, the server does not startup. For better security, set to <code>false</code> . The default is <code>false</code> . | <code>false</code> |
| stoponerror | Required | If <code>true</code> , BillingCenter stops the integration if an error occurs. Otherwise, BillingCenter just skips that item. Be sure to log any errors or notify an administrator. | <code>true</code> |
| incoming | Required | The full path of the configured incoming events directory | <code>/tmp/inbound/incoming</code> |
| processing | Required | The full path of the configured processing events directory | <code>/tmp/inbound/processing</code> |
| done | Required | The full path of the configured done events directory | <code>/tmp/inbound/done</code> |
| error | Required | The full path of the configured error events directory | <code>/tmp/inbound/error</code> |
| pollinginterval | Optional | The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the ordered parameter. See "Understanding the Polling Interval and Throttle Interval" on page 299. The default is 60 seconds. | 15 |

| File integration configuration parameters | Required | Description | Example value |
|---|----------|--|---------------|
| throttleinterval | Optional | The time interval in seconds after polling, though the algorithm interacts with the polling interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 299. The default is 60 seconds. | 15 |
| ordered | Optional | <p>By default, the inbound file integration handles multiple files at a time in parallel in multiple server threads. If you want files handled in a single thread sequentially, set this value to true. The default is false.</p> <p>WARNING: The order of the processing of individual files in the directory is undefined. Never rely on the order being any particular order, such as alphabetic or creation date.</p> <p>Also see “Understanding the Polling Interval and Throttle Interval” on page 299.</p> | false |

5. In Studio, within the Plugins registry, add a new .gwp file. For related information, see “Registering a Plugin Implementation Class” on page 139 and “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*.
6. Studio prompts for a plugin name and plugin interface. For the plugin name, use a name that represents the purpose of this specific inbound integration. For the **Interface** field:
 - For a message reply plugin, type `InboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type `InboundIntegrationStartablePlugin`.
7. Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
8. In the **Java class** field, type:
 - For a message reply plugin, type
`com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type
`com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationPlugin`.
9. Add a plugin parameter with the key **integrationservice**. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
10. Write your own *inbound integration handler plugin* implementation.
 - For a message reply plugin, your handler code must implement the plugin interface `gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin`.
 - For a startable plugin for non-messaging use, your handler code must implement the plugin interface `gw.plugin.integration.inbound.InboundIntegrationMessageReply`.

You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (Java as an OSGi bundle). If you use Java or if you require third-party libraries, Guidewire recommends implementing your code as an OSGi plugin.

This interface has one method called `process`, which has a single argument of type `Object`. The method returns no value. The file integration calls that method to process one chunk of data. The data type depends on the value you set for the `processingmode` parameter:

- If you set the `processingmode` parameter to `line`, downcast the `Object` to `String` before using it.
- If you set the `processingmode` parameter to `file`, downcast the `Object` to `java.nio.file.Path` before using it.

If your code throws an exception, BillingCenter moves the file to the error directory. Note the `stoponerror` parameter in the configuration file. If that value is `true`, BillingCenter stops the integration if an error occurs. Otherwise, skips that item. Be sure to log any errors or notify an administrator.

In Studio, within the Plugins registry, register your handler plugin implementation class. Add a row in the Plugins registry editor for your plugin implementation class. See “Registering a Plugin Implementation Class” on page 139 and “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*. Set the interface name to the handler interface that you implemented (see step 10).

IMPORTANT Within the Plugins registry for your handler plugin implementation, the Name field must match the `pluginhandler` parameter you use in the `inbound-integration-config.xml` file for this integration.

11. Add a plugin parameter with the key `integrationservice`. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
12. Start the server and test your new inbound integration.

Example File Integration

The following is an example file integration configuration in the `inbound-integration-config.xml` file:

```
<cii:file-integration name="exampleFileIntegration" disabled="true">
    <pluginhandler>InboundFileIntegrationHandler</pluginhandler>
    <pollinginterval>1</pollinginterval>
    <throttleinterval>5</throttleinterval>
    <threadpool>gw_default</threadpool>
    <ordered>true</ordered>
    <stoponerror>false</stoponerror>
    <transactional>false</transactional>
    <osgiservice>true</osgiservice>
    <processingmode>line</processingmode>
    <incoming>/tmp/incoming</incoming>
    <processing>/tmp/processing</processing>
    <error>/tmp/error</error>
    <done>/tmp/done</done>
    <charset>UTF-8</charset>
    <createdirectories>true</createdirectories>
```

The following is a simple handler class called `mycompany.integration.SimpleFileIntegration`, which prints the lines in the file:

```
package mycompany.integration
uses gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin
class SimpleFileIntegration implements InboundIntegrationHandlerPlugin {
    // this example assumes the inbound-integration-config.xml file
    // sets this to use "line" not "entire file" processing
    // See the <processingmode> element
    construct(){
        print("***** SimpleFileIntegration startup ");
    }
    override function process(obj: Object) {
        // downcast as needed (to String or java.nio.file.Path, depending on value of <processingmode>
        var line = obj as String
        print("***** SimpleFileIntegration processing one line of file: ${line} (!)");
    }
}
```

For this example, in the Plugins Registry there are two plugin implementations in the Plugins registry:

- `InboundFileIntegration.gwp` – Registers an implementation of `InboundFileIntegrationPlugin` with the required Java class `com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationPlugin`. The `integrationservice` plugin parameter is set to `exampleFileIntegration`.
- `InboundFileIntegrationHandler.gwp` – Registers an implementation of `InboundIntegrationHandlerPlugin` with the Gosu class `mycompany.integration.SimpleFileIntegration`. The `integrationservice` plugin parameter is set to `exampleFileIntegration`.

When you start up the server, you will see the log line in the console for startup:

```
***** SimpleFileIntegration startup
```

After the server starts, add files to the `/tmp/incoming` directory. You will see additional lines for each processed line. As mentioned earlier, do not add files to the directory until after the plugin is initialized. Files that are in the incoming directory on startup are never processed.

Inbound JMS Integration

BillingCenter includes a built-in high-performance multi-threaded integration with inbound queues of Java Message Service (JMS) messages. The inbound JMS integration supports application servers that implement the `commonj.work.WorkManager` interface specification. These currently include the IBM WebSphere and Oracle Weblogic application servers. Define your own code that processes an individual message, and the inbound JMS framework handles message dispatch and thread management.

BillingCenter can use JMS implementations on the application server but BillingCenter does not include its own JMS implementation. For additional advice on setting up or configuring an inbound JMS integration with BillingCenter, contact Guidewire Customer Support.

To create an inbound JMS integration

1. In the Project window, navigate to `configuration` → `config` → `integration`, and then open `inbound-integration-config.xml`
2. Configure the thread pools. See “Thread Pool Configuration” on page 283.
3. In the list of integrations, create one `<integration>` element of type `<jms-integration>`. Follow the pattern in the file to set the XML element namespace. In the base configuration, type the element name as `<ci:i:jms-integration>`. Set the `name` and `disabled` attributes as described in “Configuring a List of Inbound Integrations” on page 284.
4. Set configuration parameter subelements as follows:

| Plugin parameters in Plugins editor, description | Required | Description | Example value |
|--|----------|--|---|
| <code>pluginhandler</code> | Required | The name in the Plugins registry for an implementation of the <code>InboundIntegrationHandlerPlugin</code> plugin interface. Note that this is the <code>.gwp</code> file name, not the implementation class name. | <code>InboundJMSIntegrationExample</code> |
| <code>transactional</code> | Required | You must always set this to <code>true</code> . | <code>true</code> |
| <code>threadpool</code> | Required | The unique name of a thread pool as configured earlier in the file. See “Thread Pool Configuration” on page 283. | <code>gw_default</code> |

| Plugin parameters in Plugins editor, description | Required | Description | Example value |
|--|----------|---|----------------|
| ordered | Optional | <p>For typical use, set to true to maintain the processing of inbound JMS messages in order.</p> <p>The default value is false, which means the JMS integration dispatches the inbound messages in parallel with no guarantees of strict ordering.</p> <p>The behavior of the ordered flag with the polling and throttle interval works the same in the JMS integration as in the file integration. See “Understanding the Polling Interval and Throttle Interval” on page 299.</p> | true |
| batchlimit | Required | The maximum number of messages to receive in a poll interval | 2 |
| connectionfactoryjndi | Required | The application server configured JNDI connection factory | jms/gw/queueCF |
| destinationjndi | Required | The application server configured JNDI destination | jms/gw/queue |
| user | Optional | User name for authenticating on the JMS queue. | jsmith |
| password | Optional | Password for authenticating on the JMS queue. | pw123 |
| osgiservice | Required | Always set to true. This is for internal use. This value is independent of whether you choose to register your handler class as an OSGi plugin. | true |
| stoponerror | Required | If true, BillingCenter stops the integration if an error occurs. Otherwise, BillingCenter just skips that message. Be sure to log any errors or notify an administrator. | true |
| messagereceivetimeout | Optional | The maximum time in seconds to wait for an individual JMS message. The default is 15. | 15 |
| pollinginterval | Optional | The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 299. The default is 60 seconds. | 60 |
| throttleinterval | Optional | The time interval in seconds after polling, though the algorithm interacts with the polling interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 299. The default is 60 seconds. | 60 |

If you throw an exception in your code, the transaction of the message processing is rolled back. The original message is back in the queue.

5. In Studio, within the Plugins registry, add a new .gwp file. For related information, see “Registering a Plugin Implementation Class” on page 139 and “Using the Plugins Registry Editor” on page 109 in the Configuration Guide.
6. Studio prompts for a plugin name and plugin interface. For the plugin name, use a name that represents the purpose of this specific inbound integration. For the Interface field:

- For a message reply plugin, type `InboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type `InboundIntegrationStartablePlugin`.
7. Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
8. In the **Java class** field, type:
- For a message reply plugin, type
`com.guidewire.pl.integration.inbound.jms.DefaultJMSInboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type
`com.guidewire.pl.integration.inbound.jms.DefaultJMSInboundIntegrationPlugin`.
9. Add a plugin parameter with the key `integrationService`. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
10. Write your own *inbound integration handler plugin* implementation. Your handler code must implement a plugin interface
 - For a message reply plugin, implement the interface
`gw.plugin.integration.inbound.InboundIntegrationMessageReplyHandler`.
 - For a startable plugin for non-messaging use, implement the interface
`gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin`.You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (Java as an OSGi bundle). If you use Java or if you require third-party libraries, Guidewire recommends implementing your code as an OSGi plugin.
This interface has one method called `process`, which has a single argument of type `Object`. The method returns no value. The JMS integration calls that method to process one message. Downcast this `Object` to `javax.jms.Message` before using it.
If you throw an exception in your code, the behavior depends on the configuration parameter `stopOnError`. If that parameter has the value `true`, processing on that queue stops. If it has the value `false`, that message is skipped. Be sure to catch any errors in your processing code and log any issues so that an administrator can follow up later.
11. In Studio, within the Plugins registry, register your handler plugin implementation class. Add a row in the Plugins registry editor for your plugin implementation class. See “Registering a Plugin Implementation Class” on page 139 and “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*. Set the interface name to the handler interface that you implemented (see step 10).
-
- IMPORTANT** Within the Plugins registry for your handler plugin implementation, the Name field must match the `pluginHandler` parameter you use in the `inbound-integration-config.xml` file for this integration.
-
12. Add a plugin parameter with the key `integrationService`. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
13. Start the server and test your new inbound integration.

Custom Inbound Integrations

BillingCenter includes built-in inbound integrations of file-based input and JMS messages. If these built-in integrations do not serve your needs, write your own integration based on the plugin interfaces in the `gw.plugin.integration.inbound` package:

- `InboundIntegrationMessageReply` – message reply plugin
- `InboundIntegrationStartablePlugin` – startable plugin for other non-messaging contexts

You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (Java as an OSGi bundle). If you use Java or if you require third-party libraries, Guidewire recommends implementing your code as an OSGi plugin.

The `InboundIntegrationStartablePlugin` plugin interface extends several other interfaces:

- `InitializablePlugin` – This interface requires one method that passes plugin parameters. The plugin parameters are passed to the `setup` method in the `WorkAgent` interface, which your plugin must implement.
- `WorkAgent` – This *work agent* interface defines the core behavior of the inbound integration framework. This is the most complex part of writing your own custom inbound integration. See “Writing a Work Agent Implementation” on page 293.
- `IStartablePlugin` – The startable plugin interface defines methods such as `start`, `stop`, and `getState`. See “Startable Plugins Overview” on page 273. Be aware that the main `WorkAgent` interface defines `start` and `stop` methods with no arguments. The `IStartablePlugin` interface adds additional method signatures of the `start` and `stop` methods that take arguments.

The `InboundIntegrationMessageReply` plugin interface extends several other interfaces:

- `MessageReply` – The interface for classes that handle replies from messages sent by a `MessageTransport` implementation. See “Implementing Messaging Plugins” on page 347 for the relationship between the interfaces `MessageTransport`, `MessageRequest`, and `MessageReply`.
- `WorkAgent` – This *work agent* interface defines the core behavior of the inbound integration framework. This is the most complex part of writing your own custom inbound integration. See “Writing a Work Agent Implementation” on page 293.

There are no other methods on `InboundIntegrationStartablePlugin` not defined in one of those other interfaces.

After you write your own custom implementation of `InboundIntegrationStartablePlugin`, use the BillingCenter Studio Plugins Registry to register your plugin implementation with BillingCenter. See “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*.

Writing a Work Agent Implementation

The `gw.api.integration.inbound.WorkAgent` interface defines methods that coordinate and process work. You must write your own class that implements this interface. In addition to the primary functions of each method, you may also want to perform some logging to help debug your code.

To write a complete work agent implementation, you must write multiple related classes that work together. Refer to the following table for a summary of each class

| Class that you write | Implements this interface | Description |
|--|--|--|
| A work agent | <code>WorkAgent</code> | Your plugin is the top level class that coordinates work for this service. Instantiates your class that implements the interface <code>Factory</code> . |
| Finding and preparing work during each polling interval | | |
| A factory | <code>Factory</code> | A factory is a class that for each polling interval. Instantiates your class that implements the interface <code>WorkSetProcessor</code> . |
| A work set processor | <code>WorkSetProcessor</code> If your plugin supports the optional feature of being <i>transactional</i> , implement the subinterface <code>TransactionalWorkSetProcessor</code> | An object that knows how to acquire and divide resources. Instantiates your class that implements the interface <code>Inbound</code> . The main method for processing one unit of work within a <code>WorkData</code> object is the <code>process</code> method within this object. |

| Class that you write | Implements this interface | Description |
|-------------------------------------|---------------------------|--|
| Inbound | Inbound | A class that knows how to find work in its <code>findWork</code> method and return new work data sets. Instantiates your class that implements the interface <code>WorkDataSet</code> . |
| Representing the work itself | | |
| A work data set | <code>WorkDataSet</code> | Represents the set of all data found in this polling interval. Encapsulates a set of work data (<code>WorkData</code>) objects and any necessary context information to operate on the data. Instantiates your class that implements the interface <code>WorkData</code> . |
| Work data | <code>WorkData</code> | Represents one unit of work |

Setup and Teardown the Work Agent

In your `WorkAgent` implementation class, write a `setup` method that initializes your resources. BillingCenter calls the `setup` method before the `start` method. The method signature is:

```
public void setup(Map<String, Object> properties);
```

The `java.util.Map` object that is the method argument is the set of plugin parameters from the Studio Plugins Editor for your plugin implementation.

Implement the `teardown` method to release resources acquired in the `start` method. BillingCenter calls the `teardown` method before the `stop` method.

Start and Stop the Plugin

In your `WorkAgent` implementation class, implement the plugin method `start` to start the work listener and perform any necessary initialization that must happen each time you start the work agent.

Implement the plugin method `stop` and perform any necessary logic to stop your work agent.

Compare and contrast the `start` and `stop` methods with the different methods `setup` and `teardown`. See “Setup and Teardown the Work Agent” on page 294.

Handling Start and Stop in Custom Startable Plugins

If you use the startable plugin variant of in the inbound integration plugin (`InboundIntegrationStartablePlugin`), be aware that the main `WorkAgent` interface defines `start` and `stop` methods with no arguments.

The startable plugin variant also implements the interface `IStartablePlugin`. This means you must add additional method signatures of the `start` and `stop` methods that take arguments. See related topic “Startable Plugins Overview” on page 273.

For startable plugin variants of the custom inbound integration plugin, all method signatures of the `start` and `stop` method must call the appropriate method of the utility class `gw.api.integration.inbound.CustomWorkAgent`. As a method argument, pass the plugin name that you used in your `inbound-integration-config.xml` file with the `name` attribute on the `<custom-integration>` element.

- In each `start` method, call `CustomWorkAgent.startCustomWorkAgent(pluginName)`
- In each `stop` method, call `CustomWorkAgent.stopCustomWorkAgent(pluginName)`

The following Java example demonstrates this pattern:

```
import gw.api.integration.inbound.CustomWorkAgent;
import gw.api.server.Availability;
import gw.api.server.AvailabilityLevel;
import gw.api.startable.IStartablePlugin;

@Availability(AvailabilityLevel.MULTIUSER)
public class CustomStartableInboundIntegration implements IStartablePlugin {
    private String _name = "exampleCustomIntegration";
```

```
...
private void start() throws GWLifecycleException {
    CustomWorkAgent.startCustomWorkAgent( _name );
}

private void stop() {
    try {
        CustomWorkAgent.stopCustomWorkAgent( _name );
    }
    catch ( GWLifecycleException e ){
        throw new RuntimeException( e );
    }
}
```

Declare Whether Your Work Agent is Transactional

BillingCenter calls the plugin method `transactional` to determine whether your agent is *transactional*. A transactional work agent creates work items with a slightly different interface. There are additional methods that you must implement to begin work, to commit work, and to roll back transactional changes to partially finished work. To specify your work agent is transactional, return `true` from the `transactional` method. Otherwise, return `false`.

If your work agent is transactional, your implementation of `Factory.createWorkUnit()` must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface is a subinterface of (extends from) `WorkSetProcessor`. See “Get a Factory for the Work Agent” on page 295.

Get a Factory for the Work Agent

In your `WorkAgent` implementation class, implement the `factory` method. This method must return a `Factory` object, which represents an object that creates work data sets.

The `Factory` interface has only one method, which is called `createWorkProcessor`. It takes no arguments and returns an instance of your own custom class that implements the `WorkSetProcessor` interface.

If your agent is transactional, your implementation of `Factory.createWorkProcessor()` must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface is a subinterface of `WorkSetProcessor`. See “Declare Whether Your Work Agent is Transactional” on page 295. Both interfaces are in the `gw.api.integration.inbound` package.

Writing a Work Set Processor

Most of your actual work happens in code called a work set processor, which is a class that you create that implements the `WorkSetProcessor` interface or its subinterface `TransactionalWorkSetProcessor`. Both interfaces are in the `gw.api.integration.inbound` package. See “Declare Whether Your Work Agent is Transactional” on page 295

The basic `WorkSetProcessor` interface defines two methods

- `getInbound` – Gets an object that knows how to acquire and divide resources to create work items. This method returns an object of type `Inbound`, which is an interface with only one method, called `findWork`. Define your own class that implements the `Inbound` interface. The `findWork` method must get the *work data set*, which represent multiple work items. If your plugin supports unordered multi-threaded work, each work item represents work that can be done by its own thread. For example, for the inbound file integration, a work data set is a list of newly-added files. Each file is a separate work item. The `findWork` method returns the data set encapsulated in a `WorkDataSet` object. The polling process of the inbound integration framework calls the `findWork` method to do the main work of getting new data to process. See “Error Handling” on page 296. From Gosu, this method appears as a getter for the `Inbound` property rather than as a method.

- **process** – Processes one work data item within a work data set. The method takes two arguments of type `WorkContext` and `WorkData`. The `WorkData` is one work item in the work data set. You can optionally choose to use the `WorkContext` to declare a resource or other context necessary to process the data item. Your own implementation of the work data set (`WorkDataSet`) is responsible for populating this context information if you need it. For example, if your inbound integration is listening to a message queue, you might store the connection or queue information in the `WorkContext` object. Your `WorkSetProcessor` can then access this connection information in the `process` method when processing one message on the queue. See “Error Handling” on page 296.

Only if your agent is transactional, your implementation of `Factory.createWorkUnit()` must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface is a subinterface of `WorkSetProcessor`. See “Declare Whether Your Work Agent is Transactional” on page 295.

The `TransactionalWorkSetProcessor` interface defines several additional methods:

- **begin** – Begin any necessary transactional context. You are responsible for management of any transactions.
- **commit** – Commit any changes. You are responsible for management of any transactions.
- **rollback** – Rollback any changes. You are responsible for management of any transactions.

All three methods take a single argument of type `TxContext`. The `TxContext` interface is a subinterface of `WorkContext`. Use it to represent customer-specific work context information that also contains transaction-specific information. Create your own implementation of this class in your `getContext` method of your `WorkDataSet`. See “Creating a Work Data Set” on page 296.

Error Handling

Any exceptions in `WorkDataSet.findWork()` causes BillingCenter to immediately stop processing until the plugin is restarted or the server is restarted.

Any exceptions in the `WorkSetProcessor` in the `process` method are logged and the item is skipped.

WARNING It is important to catch any exceptions in the `process` method to ensure that you correctly handle error conditions. For example, you may need to notify administrators or place the work item in a special location for special handling.

Creating a Work Data Set

You must implement your own class that encapsulates knowledge about a work data set, which represents the set of all data found in this polling interval. The work data set is created by your own implementation of the `Inbound.findWork()` method. For example, an inbound file integration creates a work data set representing a list of all new files in an incoming directory. See “Get a Factory for the Work Agent” on page 295.

Create a class that implements the `gw.api.integration.inbound.WorkDataSet` interface. Your class must implement the following methods:

- **getData** – Get the next work item and move any iterator that you maintain forward one item so that the next call returns the next item after this one. Return a `WorkData` object if there are more items to process. Return `null` to indicate no more items. For example, an inbound file integration might return the next item in a list of files. The `WorkData` interface is a marker interface, so it has no methods. Write your own implementation of a class that implements this interface. Add any object variables necessary to store information to represent one work item. It is the `WorkDataSet.getData()` method that is responsible for instantiating the appropriate class that you write and populating any appropriate data fields. For example, for an inbound file integration, one `WorkData` item might represent one new file to process.

Note: From Gosu, the `getData` method appears as a getter for the `Data` property, not a method.

- `hasNext` – Return `true` if there are any unprocessed items, otherwise return `false`. In other words, if this same object's `getData` method would return a non-null value if called immediately, return `true`.
- `getContext` – Your implementation of a work data set can optionally declare a resource or other context necessary to process the data item. You are responsible for populating this context information if you need it. For example, if your inbound integration listens to a message queue, store the connection or queue information in an instance of a class that you write that implements `gw.api.integration.inbound.WorkContext`. In your code that processes each item (your `WorkSetProcessor` implementation), you can access this connection information from the `WorkSetProcessor` object's `process` method when processing each new message. If your plugin supports transactional work items (the `transactional` plugin parameter), your class must implement a subinterface called `TxContext`, which requires two additional methods:
 - `isRollback` – Returns a `boolean` value that indicates the transaction will be rolled back.
 - `setRollbackOnly` – Set your own `boolean` value to indicate that a rollback will occur.
- `close` – Close and release any resources acquired by your work data set.

Getting Parameters

Like all other plugin types, your plugin implementation can get parameter values. See “Plugin Parameters” on page 140. The `Map` argument to the `setup` method includes all parameters that you set in the `inbound-integration-config.xml` file for that integration. Save the map or the values of important parameters in private variables in your plugin implementation.

The map argument to the `setup` method also includes any arbitrary parameters that you set in the `<parameters>` configuration element. See the parameter called `parameters` in “Installing a New Custom Inbound Integration” on page 297.

Installing a New Custom Inbound Integration

To create and register a new custom inbound integration

1. Design and write all required implementation Java classes as described in “Installing a New Custom Inbound Integration” on page 297.

IMPORTANT You must write your plugin implementation for this plugin interface in Java, not Gosu.

2. In the Project window, navigate to `configuration` → `config` → `integration`, and the open `inbound-integration-config.xml`.
3. Configure the thread pools. See “Thread Pool Configuration” on page 283.
4. In the list of integrations, create one `<integration>` element of type `<custom-integration>`. Follow the pattern in the file to set the XML element namespace. In the base configuration, type the element name as `<ci:i:custom-integration>`. Set the `name` and `disabled` attributes as described in “Configuring a List of Inbound Integrations” on page 284.

5. Set configuration parameter subelements as follows:

| Plugin parameters in Plugins editor, description | Required | Description | Example |
|--|----------|---|---|
| <code>workagentimpl</code> | Required | The name of the <code>InboundIntegrationStartablePlugin</code> or <code>InboundIntegrationMessageReply</code> implementation class. If this is a Java class, list the fully-qualified name of the class with the package. If this is a Gosu class, list the fully-qualified name of the class with the package, with the suffix <code>.gs</code> . For example, <code>mycompany.integ.MyInboundPlugin.gs</code> | <code>mycompany.integ.MyInboundPlugin.gs</code> |
| <code>pluginhandler</code> | n/a | <i>This parameter is unused in custom inbound integrations. This parameter is used only for file and JMS integrations.</i> | n/a |
| <code>transactional</code> | Optional | Always set this boolean value to the same value returned by your plugin implementation's <code>transactional</code> method. See "Declare Whether Your Work Agent is Transactional" on page 295. If not set, defaults to <code>false</code> . | true |
| <code>threadpool</code> | Optional | The unique name of a thread pool as configured earlier in the file. See "Thread Pool Configuration" on page 283. | <code>gw_default</code> |
| <code>stoponerror</code> | Required | If true, BillingCenter stops the integration if an error occurs. Otherwise, BillingCenter just skips that item. Be sure to log any errors or notify an administrator. | true |
| <code>osgiservice</code> | Required | If you deploy your plugin implementation as an OSGi plugin, set this to <code>true</code> . If you deploy your plugin implementation as a Gosu plugin, set this to <code>true</code> . If you deploy your plugin implementation as a Java plugin (not using OSGi), set this to <code>false</code> . | true |
| <code>ordered</code> | Optional | Set to <code>true</code> to process in a single thread. Set to <code>false</code> to process items in multiple threads, as managed by the thread pool. The behavior of the <code>ordered</code> flag interacts with the behavior of the polling interval and throttle interval. See "Understanding the Polling Interval and Throttle Interval" on page 299. | true |
| <code>pollinginterval</code> | Optional | The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the <code>ordered</code> parameter. See "Understanding the Polling Interval and Throttle Interval" on page 299. The default is 60 seconds. | 60 |

| Plugin parameters in Plugins editor, description | Required | Description | Example |
|--|----------|--|--|
| throttleinterval | Optional | The time interval in seconds after polling, though the algorithm interacts with the polling interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 299. The default is 60 seconds. | 60 |
| parameters | Optional | Arbitrary parameters that you can define, for example to store server names and port numbers. Define subelements that alternate between key and value elements, contain key/value pairs. For example: The plugin interface has a setup method. These parameters are in the java.util.Map that BillingCenter passes to that initialization method. | <parameters> <key>key1</key> <value>myvalue</value> </parameters> |

6. In Studio, within the Plugins registry, add a new .gwp file. For related information, see “Registering a Plugin Implementation Class” on page 139 and “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*.
7. Studio prompts for a plugin name and plugin interface. For the name, use a name that represents the purpose of this specific inbound integration.
8. For the **Interface** field, type the plugin interface you implemented, either `InboundIntegrationStartablePlugin` or `InboundIntegrationMessageReply`.
9. Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
10. In the **Java class** field, type the fully-qualified name of your plugin implementation.
11. Add a plugin parameter with the key `integrationservice`. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
12. Start the server and test your new inbound integration. Add logging code as appropriate to confirm the integration.

Understanding the Polling Interval and Throttle Interval

There are two different configuration parameters for coordinating when the work begins: `pollinginterval` and `throttleinterval`.

The primary mechanism is the *polling interval* (`pollinginterval`). Additionally, there is a throttle interval (`throttleinterval`) which you can use to reduce the impact on the server load or external resources.

1. At the beginning of the polling interval, the integration polls for new work and creates new work items but not yet begin the items.
2. BillingCenter begins working on the work items.
 - If the work is ordered (the `ordered` parameter is `true`), the work happens in the same thread.
 - If the work is unordered (the `ordered` parameter is `false`), the work happens in separate additional threads with no guarantee of strict ordering.
3. After the current work is complete, the system determines how much time has transpired and compares with the two interval parameters. The behavior is slightly different for ordered and unordered work.

If the work is ordered (the `ordered` parameter is `true`), the following table describes the behavior.

| If the total time since last polling is... | Behavior |
|---|--|
| Less than the <code>pollinginterval</code> | The server waits until the remaining part of the polling interval and then waits the complete <code>throttleinterval</code> time |
| Greater than the <code>pollinginterval</code> but less than the sum of the <code>pollinginterval</code> and the <code>throttleinterval</code> | The server waits until the end of the sum of the <code>pollinginterval</code> and the <code>throttleinterval</code> times |
| Greater than the sum of the <code>pollinginterval</code> and the <code>throttleinterval</code> | The server immediately polls for new work |

If the work is unordered, the same time check occurs with some differences:

- the work proceeds in parallel
- the time check happens immediately after new work is created but does not wait for the work to be done.

part IV

Messaging

Messaging and Events

When certain actions occur in BillingCenter—such as sending an invoice or receiving a payment—the action triggers an *event* associated with that action. The event invokes optional configuration code to process or handle the event. For example, when an invoice is sent, the triggered event might run configuration code that communicates with an external invoice system, instructing the system to send the invoice. The instruction sent to the external system is called a *message*.

BillingCenter defines a large number of events of potential interest to external systems. Write rules to generate messages in response to events of interest. BillingCenter queues these messages and then dispatches them to the receiving systems.

This topic explains how BillingCenter generates messages in response to events and how to connect external systems to receive those messages.

This topic discusses *plugins*, which are software modules that BillingCenter calls to perform an action or calculate a result. For information on plugins, see “Plugin Overview” on page 135. For the complete list of all BillingCenter plugins, see “Summary of All BillingCenter Plugins” on page 153.

Register new messaging plugins in Studio. As you register plugins in Studio, Studio prompts you for a plugin interface name (in a picker) and, in some cases, for a plugin name. Use that plugin *name* as you configure the messaging destination in the Messaging editor in Studio.

This topic includes:

- “Messaging Overview” on page 304
- “Message Destination Overview” on page 315
- “List of Messaging Events in BillingCenter” on page 323
- “Generating New Messages in Event Fired Rules” on page 331
- “Message Ordering and Multi-Threaded Sending” on page 337
- “Late Binding Data in Your Payload” on page 342
- “Reporting Acknowledgements and Errors” on page 343
- “Tracking a Specific Entity With a Message” on page 347
- “Implementing Messaging Plugins” on page 347

- “Resynchronizing Messages for a Primary Object” on page 353
- “Message Payload Mapping Utility for Java Plugins” on page 356
- “Message Status Code Reference” on page 357
- “Monitoring Messages and Handling Errors” on page 358
- “Messaging Tools Web Service” on page 360
- “Batch Mode Integration” on page 362
- “Included Messaging Transports” on page 363

Messaging Overview

To understand this topic, it might help to get a high-level overview of terminology and an overview of events in BillingCenter. The following subtopics summarizes important messaging concepts.

Event

An event is an abstract notification of a change in BillingCenter that might be interesting to an external system. An event most often represents a user action to application data, or an API that changed application data. For some (but not all) entities in the data model configuration files, if a user action or API adds, changes, or removes an entity instance, the system triggers an event. Each event has a name, which is a `String` value.

For example, in BillingCenter adding a new payment triggers an event. The event name is `"PaymentMoneyReceivedAdded"`.

One user action or API call might trigger multiple events for different objects in one database transaction. In some cases, the object might have multiple events occur in one database transaction.

Triggering an event triggers one call to the Event Fired rule set for each external system that is interested in that event.

Event Fired rules for an event name run only if you tell BillingCenter that at least one external system is interested in that event. To request that BillingCenter run Event Fired rules for that event and that destination ID, see “Message Destination Overview” on page 315.

Additionally, only entity types defined with the `<events>` tag by default generate added, changed, or deleted events. For more information, see “`<events>`” on page 185 in the *Configuration Guide*.

Message

A message is information to send to an external system in response to an event. BillingCenter can ignore the event or send one or more messages to each external system that cares about that event. In addition to an ID and some status information, each message has a *message payload*. The payload is the main data content of the message. The payload is `String` in the `Message.Payload` property.

Your code creates messages in the Event Fired rule set. For more information, see “Generating New Messages in Event Fired Rules” on page 331. Because message creation impacts user response times, avoid unnecessarily large or complex messages if possible.

Message History

After a message is sent, the application converts a message (`Message`) object to a message history (`MessageHistory`) object. The message history object has the same properties as a message object.

You can use the message history database table to understand the messaging history to your external systems. This can be useful to help understand problems.

Additionally, message history objects are important when detecting duplicate messages. To report a duplicate message, you must find the message history object. Call the `reportDuplicate` method on the message history object, which represents the original message. See “[Reporting Acknowledgements and Errors](#)” on page 343.

Messaging Destinations

A messaging destination is an external system to which to send messages. Generally speaking, a messaging destination represents one external system. Register one destination for each external system, even if that external system is used for multiple types of data.

Register your messaging destinations in Studio. In the **Messaging** editor, specify which classes implement the messaging plugin interfaces for that messaging destination. The most important message plugin interface is `MessageTransport`. The `MessageTransport` plugin interface is responsible for actually dispatching the message (most importantly, its payload) to your external system.

Each destination registers a *list of event names* for which it wants notifications. Each destination may listen for different events compared to other destinations. The Event Fired rule set runs once for every combination of an event name and a destination interested in that event. To request that BillingCenter run Event Fired rules for that event and that destination ID, see “[Message Destination Overview](#)” on page 315.

If more than one messaging destination requests an event name, BillingCenter runs the Event Fired rule set once for every destination that requested it. To determine which destination this event trigger is for, your Event Fired rules must check the `DestinationID` property of the message context object at run time.

In the **Messaging** editor there are other important settings for each destination, including:

- **Alternative Primary Entity** – See “[Primary Entity and Primary Object](#)” on page 305
- **Message Without Primary** – See “[How Destination Settings Affects Ordering](#)” on page 341

Root Object

A root object for an event is the entity instance that is most associated with the event. This might be the primary entity that is the top of a hierarchy of objects, or it might a small subobject.

Separate from the concept of a root object for an event, each message has a root object. By default, the message’s root object is the same as the root object for the event that triggered the Event Fired rules within which you created the message. This default makes sense in most cases. You can override this default for a message if desired. See “[Setting a Message Root Object or Primary Object](#)” on page 335.

Primary Entity and Primary Object

A *primary entity* represents a type of high-level object that a Guidewire application uses to group and sort related messages. A *primary object* is a specific instance of a primary entity. Each Guidewire application specifies a default *primary entity* type for the application, or no default primary entity type.

Additionally, messaging destinations can override the primary entity type, and that setting applies just to that messaging destination. Only specific entity types are supported for each Guidewire application.

IMPORTANT Determining which primary object, if any, applies for a messaging destination is critical to understanding how BillingCenter orders messages. See “[Safe Ordering](#)” on page 306 and “[Message Ordering and Multi-Threaded Sending](#)” on page 337.

In BillingCenter:

- There is no default primary entity in BillingCenter.
- A messaging destination can specify another entity as an alternative primary object, and that setting applies just to that messaging destination. The supported alternative primary entities are as follows:

- Account – for ordering account-related messages
- Producer – for ordering producer-related messages
- PolicyPeriod – for ordering policy-period-related messages
- Contact – for ordering contact-related messages. This choice supports the integration with contact systems. In particular, this supports the integration with ContactManager, which is licensed with the optional Client Data Management module of the PolicyCenter product.

IMPORTANT To support safe ordering messages, after you create a message you must set properties on Message object to point to the primary object. See “Setting a Message Root Object or Primary Object” on page 335. Additionally, you need to set the alternative primary entity configuration on each destination. See “Messaging Destinations” on page 305.

- No other entity types can be alternative primary entities for a messaging destination.

Some objects are not associated with any primary object. For example, User objects are not associated with a single account, producer, or contact. Messages associated with such objects are called *non-safe-ordered messages*. See “Safe Ordering” on page 306 and “Message Ordering and Multi-Threaded Sending” on page 337.

Do not confuse the *root object* for a message with the *primary object* associated with a message. The root object is the generally the object that triggered the event. The primary object is the highest-level object related to the root object.

To configure how a message is associated with a primary entity, you must manually set the primary entity properties for a message. See “Setting a Message Root Object or Primary Object” on page 335.

Acknowledgement (ACK and NAK)

An acknowledgement is a formal response from an external system back to a BillingCenter that declares how successfully the system processed the message:

- A *positive acknowledgement* (ACK) means that the external system processed the message successfully.
- A *negative acknowledgement* (NAK) means the external system failed to handle the message due to errors.

It is very important for integration programmers to understand that BillingCenter distinguishes between the following errors:

- An error that throws a Gosu or Java exception during initial sending in the `MessageTransport` method called `send`. Errors during this phase typically indicate network errors or other automatically retryable errors. If the `send` method throws an exception, BillingCenter automatically retries sending the message multiple times.
- An error reported from the external system, which are also called a NAK.

For more information about error handling, see “Error Handling in Messaging Plugins” on page 352.

Safe Ordering

Safe ordering is a messaging feature that causes related messages to send in the same order they were sent. This is called *safe order*. For each messaging destination, the application groups messages by their associated *primary object*. For important definitions relevant to this topic, see “Primary Entity and Primary Object” on page 305.

Any messages associated with a primary object for that destination are called *safe-ordered messages*. The application waits for an acknowledgement before processing the next safe-ordered message for that same primary object for that destination. Any delays or errors for that destination always block further sending of messages for that same destination for that same primary object.

In BillingCenter, there is no default primary entity. However, every messaging destination can set its own *alternative primary entity* to one of several acceptable values in the `Messaging` editor:

- If the alternative primary entity is **Account**, messages associated with an account send grouped by account. Within each group, BillingCenter sends messages for that messaging destination ordered by creation time.
- If the alternative primary entity is **Producer**, messages associated with an producer send grouped by producer. Within each group, BillingCenter sends messages for that messaging destination ordered by creation time.
- If the alternative primary entity is **PolicyPeriod**, messages associated with an policy period send grouped by policy period. Within each group, BillingCenter sends messages for that messaging destination ordered by creation time.
- If the alternative primary entity is **Contact**, messages associated with an contact send grouped by contact. Within each group, BillingCenter sends messages for that messaging destination ordered by creation time.

Not all messages are associated with a primary object. The logic for how and when to send these *non-safe-ordered messages* is different from the logic for safe-ordered messages. Additionally, the behavior varies based on the messaging destination configuration setting called **Strict Mode**. For details, see “Message Ordering and Multi-Threaded Sending” on page 337.

Transport Neutrality

BillingCenter does not assume any specific type of transport or message formatting.

Destinations deliver the message any way they want, including but not limited to the following:

- **Submit the message by using remote API calls** – Use a web service (SOAP) interface or a Java-specific interface to send a message to an external system.
- **Submit the message to a guaranteed-delivery messaging system** – For instance, a message queue.
- **Save to special files in the file system** – For large-scale batch handling, you could send a message by implementing writing data to local text files that are read by nightly batch processes. If you do this, remember to make your plugins thread-safe when writing to the files.
- **Send e-mails** – The destination might send e-mails, which might not guarantee delivery or order, depending on the type of mail system. This approach is acceptable for simple administrative notifications but is inappropriate for systems that rely on guaranteed delivery and message order, which includes most real-world installations.

Messaging Flow Overview

The high-level steps of event and message generation and processing are as follows.

1. **Application startup** – At application startup, BillingCenter checks its configuration information and constructs messaging destinations. Each destination registers for specific events for which it wants notifications.
2. **Users and APIs trigger messaging events** – Events trigger after data changes. For example, a change to data in the user interface, or an outside system calls a web service that changes data. The messaging event represents the changes to application data as the entity commits to the database.
3. **Event Fired rules create messages** – BillingCenter runs the Event Fired rule set for each event name that triggers. BillingCenter runs the rule set multiple times for an event name if multiple destinations listen for it. Your rules can choose to generate new messages. Messages have a text-based *message payload*.
In BillingCenter, for example you might write rules that check if the event name is `PaymentMoneyReceivedAdded`, and if certain other conditions occur. If so, the rules might generate a new message with an XML payload. The rules might also link entities to the message for later use.
4. **BillingCenter sends message to a destination** – Messages are put in a queue and handed one-by-one to the *messaging destination*.

In BillingCenter, you might generate an XML payload and submit the message to an external message queue to notify the external system of a payment.

5. BillingCenter waits for an acknowledgement – The external system replies with an acknowledgement to the destination after it processes the message, and the destination’s messaging plugins process this information. If the message successfully sent, the messaging plugins submit an ACK and BillingCenter sends the next message. For details of the messaging ordering system, see “Message Ordering and Multi-Threaded Sending” on page 337. The code that submits the acknowledgement might also make changes to application data, such as updating a messaging-specific property on a persisted object or advancing a workflow. If you change data, see the warnings in the section “Messaging Flow Details” on page 309. Remember that you can also submit an error (a negative acknowledgement, a NAK) instead of a positive acknowledgement.

Most messaging integration code that you write for a typical deployment is writing your Event Fired rules that create message payloads and writing `MessageTransport` plugin implementations to send messages. There are two other optional messaging plugin interfaces that are discussed later.

Once you write messaging plugin implementation classes, you register your new messaging plugin classes in Studio. Register your messaging plugin implementations first in the `Plugins` editor and then in the `Messaging` editor. For plugin interfaces that can have multiple implementations, which includes all messaging plugin interfaces, Studio asks you to *name* your plugin implementation when registering your plugin class. Use that *plugin implementation class name* when you configure the messaging destination in the `Messaging` editor.

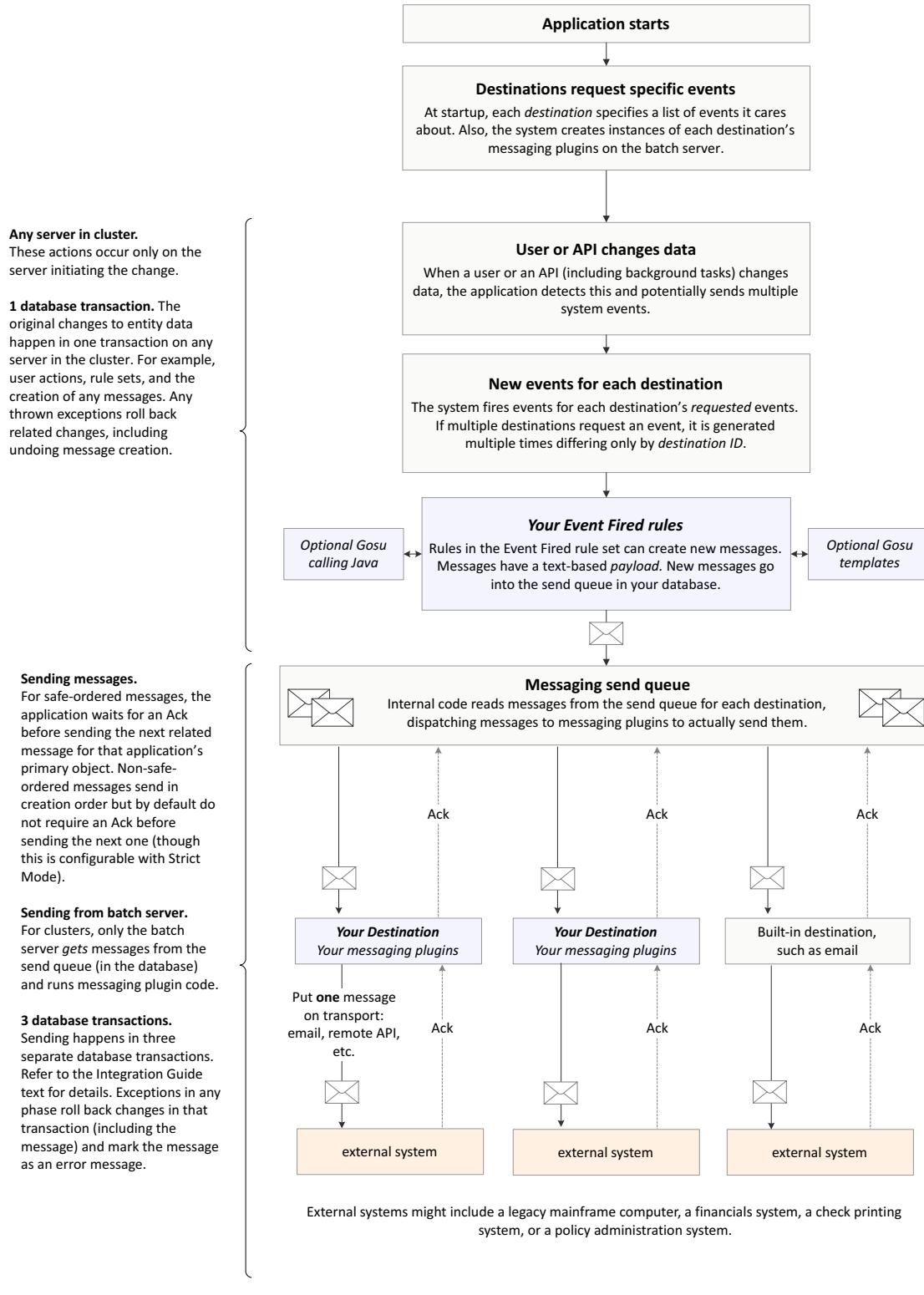
See Also

- “Using the Messaging Editor” on page 131 in the *Configuration Guide*
- “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*
- “Plugin Overview” on page 135
- “Message Destination Overview” on page 315

Messaging Flow Details

The following diagram illustrates the chronological flow of events and messaging, starting from the top of the diagram. Refer to the section following the diagram for a detailed explanation of each step.

Messaging Overview



The following list describes a detailed chronological flow of event-related actions:

- 1. Destination initialization at system startup** – After the BillingCenter application server starts, the application initializes all destinations. BillingCenter saves a list of events for which each destination requested notifications. Because this happens at system startup, if you change the list of events or destination, you must restart BillingCenter. Each destination encapsulates all the necessary behavior for that external system, but uses three different plugin interfaces to implement the destination. Each plugin handles different parts of what a destination does. The message request plugin handles message pre-processing. The message transport plugin handles message transport, and the message reply plugin handles message replies.

Register new messaging plugins in Studio first in the Plugins editor. When you create a new implementation, Studio prompts you for a plugin interface name (in a picker) and, in some cases, for a plugin name. Use that plugin name in the Messaging editor in Studio to register each destination. Remember that you need to register your plugin in two different editors in Studio.

- 2. A user or an API changes something** – A user action, API, or a batch process changed data in BillingCenter.

For example, BillingCenter triggers an event if you add an account or change a note.

It is critical to understand that the change does not fully commit to the database until step 4. Any exceptions that occur before step 4 undo the change that triggered the event. In other words, unless all follow-up actions to the change succeed, the database transaction rolls back. For related information, see “Messaging Database Transactions During Sending” on page 322.

BillingCenter generates messaging events – The same action, such as a single change to the BillingCenter database, might trigger *more* than one event. BillingCenter checks whether each destination has listed each relevant event in its messaging configuration. For each messaging destination that listens for that event, BillingCenter calls your Event Fired business rules. If multiple destinations want notifications for a specific event, BillingCenter duplicates the event for each destination that wants that event. To the business rules, these duplicates look the same except with different *destination ID* properties. It is critical to understand that a change to BillingCenter data might generate events for one destination but not for another destination. To change this list for destination, review the **Messaging** editor in Studio and select the row for your destination. Additionally, only entity types defined with the `<events>` tag by default generate added, changed, or deleted events. For more information, see “`<events>`” on page 185 in the *Configuration Guide*.

- 3. BillingCenter invokes Event Fired rules (and they generate messages)** – BillingCenter calls the Event Fired rule set for each destination-event pair. Event Fired rules must rules check the event name and the messaging destination ID to determine whether to send a message for that event. Your Event Fired rules generate messages using the Gosu method `messageContext.createMessage(payload)`. The rule actions choose whether to generate a message, the order of multiple messages, and the text-based message *payload*. For more information, see “Generating New Messages in Event Fired Rules” on page 331. Your rules can use the following techniques:

- Optionally export an entity to XML using generated XSDs** – Studio includes a tool that helps you export business data entities (and other types like Gosu classes) to XML. You can select which properties are required or optional for each integration point. You can export an XSD to describe the data interchange format you selected. Then, you can edit your Event Fired rules to generate a payload for the entity that conforms to your custom XSD. For more details, see “Creating XML Payloads Using Guidewire XML (GX) Models” on page 336.

The Guidewire XML (GX) modeler is a powerful tool for integrations. Create custom XML models that contain only the subset of entity object data that is appropriate for each integration point. It can output a custom XSD that your external system can use. You can also use GX models to parse (import) XML data into in-memory objects that describe the XML structure. See “Creating XML Payloads Using Guidewire XML (GX) Models” on page 336.

- Optionally use Gosu templates to generate the payload** – Rules can optionally use templates with embedded Gosu to generate message content.

- **Optionally use Java classes (called from Gosu) to generate the payload** – Rules can optionally use Java classes to generate the message content from Gosu business rules. See “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.
- **Optional late binding** – You can use a technique called *late binding* to include parameters in a message payload at message creation time but evaluate them immediately before sending. See “Late Binding Data in Your Payload” on page 342 for details.

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Restrictions on Entity Data in Messaging Rules and Messaging Plugins” on page 313.

4. New messages are added to the send queue – After all rules run, BillingCenter adds any new messages to the send queue in the database. The submission of messages to the send queue is part of the same database transaction that triggered the event to preserve atomicity. If the transaction succeeds, all related messages successfully enter the send queue. If the transaction *fails*, the change rolls back including all messages added during that transaction. Messages might wait in this queue for a while, depending on the state of acknowledgements and the status of safe ordering of other messages (see step 5).

5. On the batch server only, BillingCenter dispatches messages to messaging destinations – BillingCenter retrieves messages from the send queue and dispatches messages to messaging plugins for each destination. For details of how BillingCenter retrieves messages and orders them for sending, see “Safe Ordering” on page 306 and “Message Ordering and Multi-Threaded Sending” on page 337.

To send a message, BillingCenter finds the messaging destination’s transport plugin and calls its `send` method. The message transport plugin sends the message in whatever native transport layer is appropriate. See “Transport Neutrality” on page 307. If the `send` method throws an exception, BillingCenter automatically retries the message.

6. Acknowledging messages – Some destination implementations know success or failure immediately during sending. For example, a messaging transport plugin might call a synchronous remote procedure call on a destination system before returning from its `send` method.

In contrast, a messaging destination might need to wait for an *asynchronous*, time-delayed reply from the destination to confirm that it processed the message successfully. For example, it might need to wait for an incoming message on an external messaging queue that confirms the system processed that message.

In either case, the messaging destination code or the external system must confirm that the message arrived safely by submitting an *acknowledgement* (an ACK) to BillingCenter for that specific message. Alternatively, it can submit an error, also called a negative acknowledgement or NAK. You can submit an ACK or NAK in several places. For synchronous sending, submit it in your `MessageTransport` plugin during the `send` method. For asynchronous sending, submit it in your `MessageReply` plugin. For asynchronous sending, an external system could optionally use a SOAP API to submit an acknowledgement or error.

If using messaging plugins to submit the ACK, you can also make changes to data during the ACK, such as updating properties on entities.

Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Restrictions on Entity Data in Messaging Rules and Messaging Plugins” on page 313.

7. After an ACK or NAK for a safe-ordered message, BillingCenter dispatches the next related message – An ACK for a safe-ordered message affects what messages are now sendable. If there are other messages for that destination in the send queue for the same primary object, BillingCenter soon sends the next message for that primary object. For details of how messages are retrieved and ordered, see “Safe Ordering” on page 306 and “Message Ordering and Multi-Threaded Sending” on page 337. Read that section for details of the setting called Strict Mode. Strict Mode affects whether BillingCenter waits for acknowledgements for non-safe-ordered messages.

Restrictions on Entity Data in Messaging Rules and Messaging Plugins

Event Fired rules and messaging plugin implementations have limitations about changing entity instance data. Messaging code in these locations must perform only the minimal data changes necessary for integration on the message entity.

WARNING For data integrity and server reliability, you must carefully follow restrictions regarding what data you can change in Event Fired rules and messaging plugin implementations.

Event Fired Rule Set Restrictions for Entity Data Changes

WARNING Entity changes in Event Fired must be very limited. Changes do not trigger validation rules, or pre-update rules. Read this topic carefully for additional restrictions.

The following important restrictions apply to code within the Event Fired rule set:

- In general, perform any data updates in your preupdate rules, not your Event Fired rules.
- A property is safe to change in Event Fired rules only if it is messaging-specific and users can never modify it from the user interface, even indirectly. Carefully consider the role of every property you might modify.
- The only object safe to add is a new message using the `createMessage` method on the message context object. Never create new objects of any other types, even indirectly through other APIs.
- Never delete objects.
- Never call business logic APIs that might change entity data, even in edge cases.
- Never rely on any entity data changes triggering these common rule sets:
 - Pre-update rule set
 - Validation rule set
 - Event Fired rule set
- These restrictions apply to all entity types, including custom entity types.

Design your messaging rules carefully around these restrictions to avoid data corruption and logical errors that are difficult to diagnose.

Messaging Plugin Restrictions for Entity Data Changes

WARNING Entity changes in messaging plugin code must be very limited. Changes do not trigger validation rules, pre-update rules, or concurrent data change exceptions. Read this topic carefully for additional restrictions.

The following important restrictions apply to code triggered by a `MessageTransport` or `MessageReply` plugin implementation:

- A property is safe to change only if it is messaging-specific and users can never modify it from the user interface, even indirectly. Carefully consider the role of every property you might modify.
- You cannot create any new objects in general. In the default configuration, only the following objects are safe to create within a messaging plugin:
 - activities
 - notes
 - workflows and workflow items
 - work queues

You cannot rely on preupdate rules or validation rules running for those objects.

All other object types are dangerous and unsupported to add from within messaging plugins.

If you modify the data model such that there are additional foreign keys on these objects, even these objects explicitly listed may be unsafe to add. For advice on specific changes, contact Guidewire Customer Support.

- Never delete objects.
 - You must not rely on any entity data changes eventually triggering these common rule sets:
 - Pre-update rule set
 - Validation rule set
 - Event Fired rule set for standard events `ENTITYAdded`, `ENTITYChanged`, and `ENTITYRemoved`.
- However, Event Fired rule set still triggers for events that you explicitly add. You can use the API `entity.addEvent("messageName")` to add events. BillingCenter calls the Event Fired rule set once for each custom event for each messaging destination that listens for it. Your Event Fired code must conform to all restrictions in “Event Fired Rule Set Restrictions for Entity Data Changes” on page 313.
- Never call business logic APIs that might change entity data, even in edge cases.
 - From messaging plugin code, entity instance changes do not trigger concurrent data exceptions except in special rare cases. To avoid data integrity issues with concurrent changes, avoid changing data in these code locations. See later in this topic for additional guidance.
 - In some cases, consider adding or advancing a workflow as an alternative to direct data modifications from messaging code. The workflow can asynchronously perform code changes in a separate bundle outside your messaging-specific code.
 - If you must update messaging-specific data, consider how absence of detecting concurrent data changes from messaging plugins might affect your extensions to the data model. For example, suppose you intend to modify an entity type to add a property with simple data. Instead, you could add a property with a foreign key to an instance of a custom entity type. First, create the instance of your custom entity type at an early part of the lifecycle of your main objects long before messaging code runs. As mentioned earlier, it is unsupported to create a entity instance in Event Fired rules or in messaging plugins. This restriction applies to all entity types, including custom entity types. In Event Fired rules or in messaging plugins, modify the messaging-specific entity instance. With this design, there is less chance of concurrent data change conflicts from a simple change on the main business entity instance from within the user interface.

In a `MessageRequest` plugin implementation, do no data changes at all.

Separate from the concurrent data exception differences mentioned earlier, there is an optional feature to *lock* related objects during messaging actions. If you use optional locking and code tries to access locked data (the primary entity instance or the message), the calling code waits for it to be unlocked.

IMPORTANT For maximum data integrity, enable optional entity locking during messaging. See “Messaging Database Transactions During Sending” on page 322. For maximum performance, disable optional entity locking during messaging.

Design your messaging code carefully around these restrictions to avoid data corruption and logical errors that are difficult to diagnose.

Database Transactions When Creating Messages

All steps up to and including adding messages to the send queue occur in one database transaction. In the list earlier in “Messaging Flow Details” on page 309, this is step 2 through step 4. All changes commit in the same transaction. This is always the same database transaction that triggers the initial event.

The database transaction rolls back if any of the following occur:

- Exceptions in rule sets that run before message creation
- Exceptions in Event Fired rules (where you create your messages)

- Exceptions in rule sets that run after Event Fired rules but before committing the bundle to the database
- Errors committing the bundle to the database, and remember that this bundle includes new Message objects

If any of these errors occur, BillingCenter rolls back all messages added to the send queue in that transaction.

There are special rules about database transactions during message sending at the destination level. See “[Messaging Database Transactions During Sending](#)” on page 322.

Messaging in BillingCenter Clusters

If you run BillingCenter in a cluster, be aware that step 1 through step 4 can occur on any BillingCenter server within the cluster. BillingCenter processes events on the same server as the user action or API call that triggered the event.

Once a message is in the send queue (step 5 through step 7), any further action with the message occurs *only* on the batch server.

Consequently, the batch server is the only server on which messaging plugins run. Configure your batch server (and any backup batch servers) to communicate with your destination external systems. For example, remember to configure your firewalls accordingly including your backup batch servers.

For BillingCenter clusters, only the batch server actually uses your messaging plugins.

Messaging Plugins Must Not Call SOAP APIs on the Same Server

In general, avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine. Be careful about any SOAP calls to the same server. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data.

Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance. If you have questions about how to convert some particular use, locally-hosted SOAP APIs from plugins or rules, contact Customer Support.

This is true for all types of local loopback SOAP calls to the same server.

Those limitations are true for all plugin code. In addition, there are messaging-specific limitations with this approach. Specifically, BillingCenter locks the root entity for the message in the database. Any attempts to modify this entity from outside your messaging plugin (and SOAP APIs are included) result in concurrent data exceptions.

WARNING Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine. There are various problems if you call SOAP APIs that modify entities that are currently in use, including but not limited to APIs that might change the message root entity.

Message Destination Overview

To represent each external system that receives a message, you must define a *messaging destination*. Typically, a destination represents a distinct remote system. However, you could use destinations to represent different remote APIs or different message types to send from BillingCenter business rules.

Carefully choose how to structure your messaging code. For example, if there are several related remote systems or APIs, you must choose whether they are logically one messaging destination or multiple destinations. Your choice affects messaging ordering. The BillingCenter messaging system ensures there is no more than one in-flight message per primary object per destination. This means that the definition of a destination is important for message ordering and multithreading. See “[Message Ordering and Multi-Threaded Sending](#)” on page 337.

Each destination specifies a list of events for which it wants notifications and various other configuration information. Additionally, a destination encapsulates a list of your plugins that perform its main destination functions:

- **Message preparation** – (Optional) If you need message preparation before sending, write a plugin that implements the message request (`MessageRequest`) plugin interface. For example, this plugin might take simple name/value pairs stored in the message payload and construct an XML message in the format required by a transport or final message recipient. If the destination requires no special message preparation, omit the request plugin entirely for the destination. For implementation details and optional post-send processing, see “Implementing a Message Request Plugin” on page 348.
- **Message transport** – (Required) The main task of a destination is to send messages to an external system. Its underlying protocol might be an external message queue, web service request to external systems, FTP, or a proprietary legacy protocol. You actually send your messages in your own implementation of the `MessageTransport` plugin interface. Every destination must provide a message transport plugin implementation. Multiple messaging destinations might use the same messaging transport plugin implementation if they really do use the underlying transport code. For implementation details, see “Implementing a Message Transport Plugin” on page 348.
- **Message reply handling** – (Required only if asynchronous) If a destination requires an asynchronous callback for acknowledgements, implement the `MessageReply` plugin. If the destination requires no asynchronous acknowledgement, omit the reply plugin. For implementation details, see “Implementing a Message Reply Plugin” on page 350.

Note: BillingCenter includes multi-threaded inbound integration APIs that you can optionally use in conjunction with `MessageReply` plugins. For example, listen for JMS messages or process text files that represent message replies. If you want to use input data other than JMS messages or text files for message replies, write a custom integration by implementing the `InboundIntegrationMessageReply` plugin. `InboundIntegrationMessageReply` is a subinterface of `MessageReply`. See “Multi-threaded Inbound Integration Overview” on page 281 and “Custom Inbound Integrations” on page 292.

After you write code that implements your messaging plugins, register them in Studio. When registering an implementation of new messaging plugin, Studio prompts you for a plugin name. The plugin name is different from the implementation class name. The plugin name is a short arbitrary name that identifies a plugin implementation. Studio only prompts you for a plugin name for plugin interfaces that support more than one implementation.

Use that plugin name to configure the messaging destination in the Messaging editor in Studio to register each destination. For details and examples of this editor, see “Using the Messaging Editor” on page 131 in the *Configuration Guide*.

For more information about plugins, see “Plugin Overview” on page 135.

To create new destinations, configure the messaging registry to specify the list of destinations, each of which includes the following information:

- The *plugin name* class for a `MessageTransport` plugin in Java or Gosu.
- Optionally, the *plugin name* for the implementation of a `MessageRequest` plugin.
- Optionally, the *plugin name* for the implementation of a `MessageReply` plugin.
- **Retry Interval** – The amount of time in milliseconds (`initialretryinterval`) after a retryable error to retry a sending a message.
- **Max Retries** – The number of automatic retries (`maxretries`) to attempt before suspending the messaging destination.
- **Backoff Multiplier** – The amount to increase the time between retries, specified as a multiplier of the time previously attempted. For example, if the last retry time attempted was 5 minutes, and the multiplier (`retrybackoffmultiplier`) is set to 2, BillingCenter attempts the next retry in 10 minutes.
- **Event Names** – A list of events to listen for, by name. Each event triggers the Event Fired rule set for that destination. To specify that the destination wants to listen for all events, use the special event name string “`(\w)*`”.

- **Destination ID**, which typically you use in your Event Fired rules to check the intended messaging destination for the event notification. If five different destinations request an event that fires, the Event Fired rule set triggers five times for that event. They differ only in the *destination ID* property (`destID`) within each message context object. Each messaging plugin implementation must have a `setDestinationID` method. This method allows your destination to get its own destination ID to store it in a private variable. Your code can use the stored value for logging messages or send it to integration systems so that they can programmatically suspend/resume the destination if necessary.

The valid range for your destination IDs is 0 through 63, inclusive. Guidewire reserves all other destination IDs for built-in destinations such as the email transport destination.

- **Alternative Primary Entity** – See “Primary Entity and Primary Object” on page 305
- **Strict Mode** – See “How Destination Settings Affects Ordering” on page 341
- **Poll interval** – Each messaging destination pulls messages from the database (from the send queue) in batches of messages on the batch server. The application does not query again until this amount of time passes. Use this field to set the value of the polling interval to wait. After the current round of sending, the messaging destination sleeps for the remainder of the poll interval. If the current round of sending takes longer than the poll interval, then the thread does not sleep at all and continues to the next round of querying and sending. See “Message Ordering and Multi-Threaded Sending” on page 337 for details on how the polling interval works. If your performance issues primarily relate to many messages per primary object per destination, then the polling interval is the most important messaging performance setting.
- **Sender Threads** – To send messages associated with a primary object, BillingCenter can create multiple sender threads for each messaging destination to distribute the workload. These are threads that actually call the messaging plugins to send the messages. Use this field to configure the number of sender threads for safe-ordered messages. This setting is ignored for non-safe-ordered messages, since those are always handled by one thread for each destination. If your performance issues primarily relate to many messages but few messages per claim for each destination, then this is the most important messaging performance setting. For more information, see “Message Ordering and Multi-Threaded Sending” on page 337.
- **Shutdown timeout** – Messaging plugins have methods to handle the administrative commands suspend, resume, and preparing for the messaging system to shutdown. The shutdown timeout value is the length of time to wait before attempting to shutdown the messaging subsystem. For more information about suspend and shutdown actions, see “Message Destination Overview” on page 315.

Manage these settings in Guidewire Studio in the Messaging editor. See “Using the Messaging Editor” on page 131 in the *Configuration Guide*.

IMPORTANT If you register a messaging plugin, you must register it in two places. First, register it in the plugin registry in the plugin editor; see “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*. Remember the plugin implementation name that you use. You need it to configure the messaging destination in the Messaging editor in Studio to register each destination. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 131 in the *Configuration Guide*.

To write your messaging plugins, see “Implementing Messaging Plugins” on page 347.

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Restrictions on Entity Data in Messaging Rules and Messaging Plugins” on page 313.

Sharing Plugin Classes Across Multiple Destinations

It is common to implement messaging plugin classes that multiple destinations share. For example, a transport plugin might manage the transport layer for multiple destinations that use that physical protocol. In such cases, be aware of the following things:

- The class instantiates once for each destination. The instance is not shared across destinations.
- Each messaging plugin instance distinguishes itself from other instances by implementing the `setDestinationID` method and saving the destination ID in a private class variable. Use this later for logging, exception handling, or notification e-mails. For more information, see “Saving the Destination ID for Logging or Errors” on page 353.

Handling Acknowledgements

Due to differences in external systems and transports, there are two basic approaches for handling replies. BillingCenter supports synchronous and asynchronous acknowledgements, although in different ways:

1. **Synchronous acknowledgement at the transport layer** – For some transports and message types, acknowledging that a message was successfully sent can happen synchronously. For example, some systems can accept messages through an HTTP request or web service API call. For such a situation, use the synchronous acknowledgement approach. The synchronous approach requires that your transport plugin `send` method actually send the message and immediately submit the acknowledgement with the message method `reportAck`. For error and duplicate handling:
 - In most cases, including most network errors, throw an exception in the `send` method. This triggers automatic retries of the message sending using the default schedule for that messaging destination.
 - For other errors or flagging duplicate messages, call the `reportError` or `reportDuplicate` methods. See “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 344.
2. **Asynchronous acknowledgement** – Some transports might finish an initial process such as submitting a message on an external message queue. However in some cases, the transport must wait for a delayed reply before it can determine if the external system successfully processed the message. The transport can wait using polling or through some other type of callback. Finally, submit the acknowledgement as successful or an error. External systems that send status messages back through a message reply queue fit this category. There are several ways to handle asynchronous acknowledgements, as described later.

For asynchronous acknowledgement, the messaging system and code path is much more complex. In this case, the message transport plugin does not acknowledge the message during its main `send` method.

The typical way to handle asynchronous replies is through a separate plugin called the message reply plugin. The message reply plugin uses a callback function that acknowledges the message at a later time. For example, suppose the destination needed to wait for a message on a special incoming messaging queue to confirm receipt of the message. The destination’s message reply plugin registers with the queue. After it receives the remote acknowledgement, the destination reports to BillingCenter that the message successfully sent.

One special step in asynchronous acknowledgement with a message reply plugin is setting up the callback routine’s database *transaction* information appropriately. Your code must retrieve message objects safely and commit any updated objects (the ACK itself and or additional property updates) to the BillingCenter database.

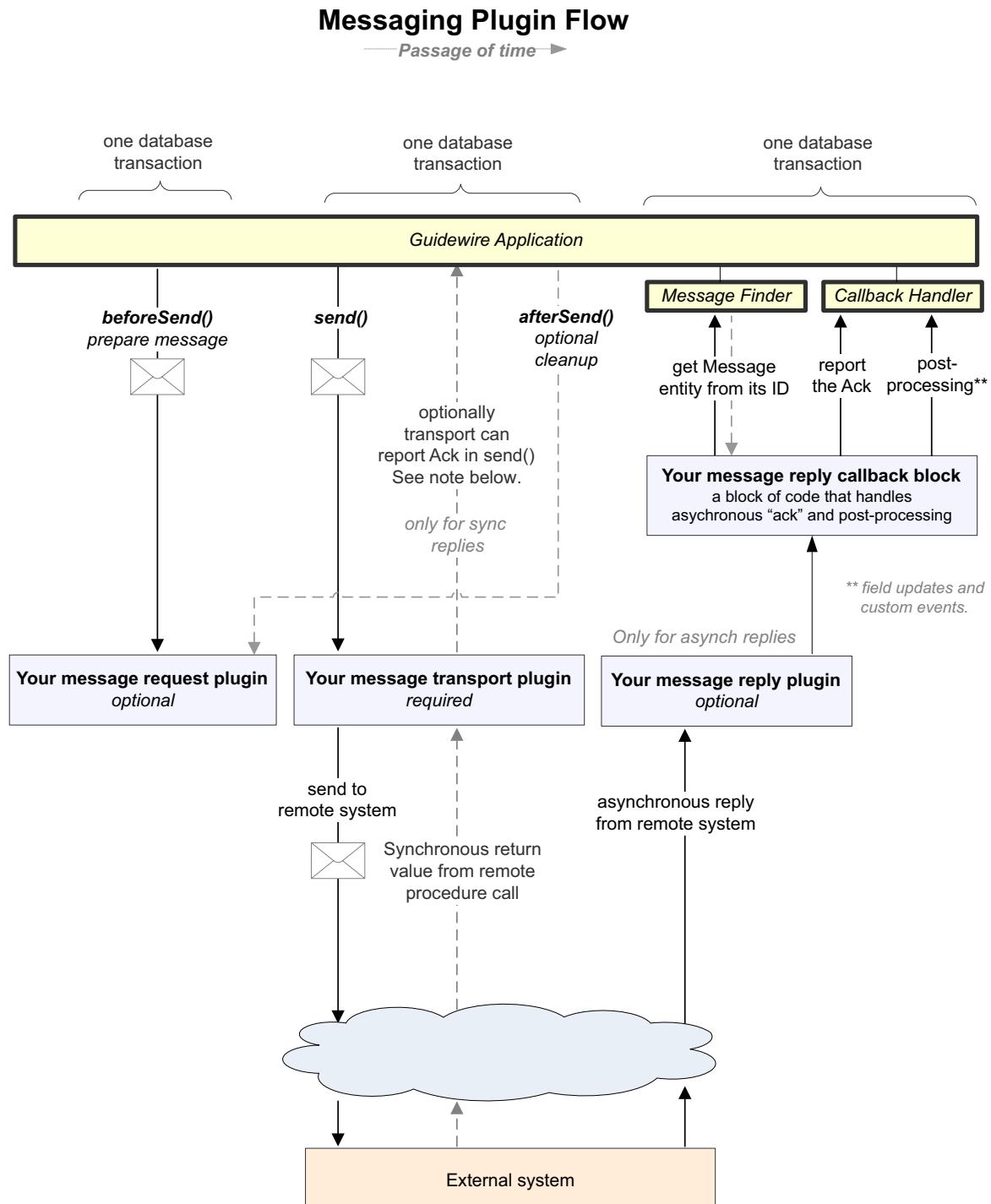
To set up the callbacks properly, Guidewire provides several interfaces, including:

- **A message finder** – A class that returns a `Message` object from its message ID (the `MessageID` property) or from its sender reference ID and destination ID (`SenderRefID` and `DestinationID`). BillingCenter provides an instance of this class to a message reply plugin upon initialization in its `initTools` method.
- **A plugin callback handler** – A class that can execute the message reply callback block in a way that ensures that any changes commit. BillingCenter provides an instance of this class to a message reply plugin upon initialization in its `initTools` method.
- **Message reply callback block interface** – The actual code that the callback handler executes is a block of code that you provide called a *message reply callback block*. This code block is written to a very simple interface with a single `run` method. This code can acknowledge a message and perform post-processing such as property updates or triggering custom events.

For more information about these objects and how to implement them, see “Implementing a Message Reply Plugin” on page 350.

An alternative to this approach is for the external system to call a BillingCenter web service (SOAP) API to acknowledge the message. BillingCenter publishes a web service called `MessagingToolsAPI` that has an `ackMessage` method. Set up an `Acknowledgement` object with the `MessageID` set to the message ID as a `String`. If it was an error, set the `Error` property to `true`. Pass the `Acknowledgement` to the `ackMessage` method.

The following diagram illustrates the destination-related plugins, associated components, along with the chronological flow of actions between elements in the system.



Guidewire code
Your Code
External system

Note: A message transport can acknowledge the message from within the transport's `send()` method if possible. In that case, post-processing such as field updates would be done during the `send()` method. For destinations where the reply must be asynchronous (delayed), that destination must define a *message reply plugin* and its *callback block* to listen for and report the acknowledgement.

Rule Sets Must Never Call Message Methods for ACK, Error, or Skip

From within rule set code, you must never call any message acknowledgment or skipping methods such as the `Message` methods `reportAck`, `reportError`, or `skip`. Use those `Message` methods only within messaging plugins.

This prohibition also applies to Event Fired rules.

Messaging Database Transactions During Sending

As mentioned in “Database Transactions When Creating Messages” on page 314, all steps up to and including adding the message to the send queue occurs in one database transaction. This is the same database transaction that triggered the event.

Additionally, there are special rules about database transactions during message sending at the destination level.

1. BillingCenter calls `MessageRequest.beforeSend(...)` in one database transaction and commits changes assuming no exceptions occurred.
2. BillingCenter calls `MessageTransport.send(...)` and then `MessageRequest.afterSend(...)` in one database transaction and commits changes assuming no exceptions occurred.
3. The `MessageReply` plugin, which optionally handles asynchronous acknowledgements to messages, does its work in a separate database transaction and commits changes assuming no exceptions occurred.

Message and Entity Locking

At the start of each transaction, BillingCenter locks the message object itself at a database level. This ensures that the application does not try to acknowledge a skipped message or similar conditions. If some code tries to access the message while it is locked, the calling code waits for the current lock holder to unlock the message. See the diagram earlier in this section for the timing of messaging database transactions.

BillingCenter can optionally lock the primary entity instance for a message if there is an associated safe ordered object for the message. The entity instance locking only affects the one entity instance, not any subobjects.

Similar to message locking, if some code tries to access the message while it is locked, the calling code waits for the current lock holder to unlock the message. BillingCenter checks the `config.xml` parameter

`LockPrimaryEntityDuringMessageHandling`. If it is set to `true`, BillingCenter locks the primary entity instance for that destination during the following operations:

- during send
- during message reply handling
- while marking a message as skipped

For example, if the message is associated with a account, during all of these operations you can request that BillingCenter optionally lock the associated `Account` object at the database level. This can reduce problems in edge cases in which other threads try to modify objects associated with this account.

WARNING This entity instance (and message) lock handling is separate from the concurrent data exception system. Be aware that concurrent data exception checking is disabled during messaging access. For important related information, see “Restrictions on Entity Data in Messaging Rules and Messaging Plugins” on page 313.

Built-In Destinations

Your rule sets can send standard e-mails and optionally attach the email to the account as a document. The built-in email APIs use the built-in email destination. For more information about email configuration and API to send emails, see “Sending Emails” on page 79 in the *Rules Guide*. BillingCenter always creates the built-in email destination, independent of your configuration settings.

List of Messaging Events in BillingCenter

BillingCenter generates events associated with a specific entity instance as the root object for the event. For more information about root objects, see “Root Object” on page 305. Also, contrast this with the concept of a primary object, see “Primary Entity and Primary Object” on page 305.

Sometimes an event root object is a higher-level object such as account. In other cases, the event is on a subobject, and your Event Fired rules must do some work to determine what high-level object it is about. For example, the Address subobject is common and changing it is common. However, what larger object contains this address? You might need this additional context to do something useful with the event.

For example, was the address an account holder’s address? Is it part of some other BillingCenter object?

In BillingCenter, most subobjects have properties that link to related objects:

- Most account objects have a property that points to the account. Sometimes this property is called Account, but it might have other names, for example Issuance.IssuanceAccount.
- Some objects have a PolicyPeriod object that points to a PolicyPeriod.
- Some objects have a Producer object that points to a Producer.

BillingCenter triggers events for many objects if an entity is added, removed (or retired), or if a property changes on the object. For example, selecting a different account type on any account generates a *changed* event on the account.

The changes are about the change to that database row itself, not on subobjects. For example, BillingCenter reports a change to an object if a foreign key reference to a subobject changes but not if properties on the subobject changes.

There are exceptions to this rule.

For example, switching to a different account contact on an account is a change to the account.

The following table describes the events that BillingCenter raises. In this table, *standard* events refer to the *added*, *changed*, and *removed* events for entities that generate events. For example, Account entity would generate events whenever code adds, changes, or removes entities of that type in the database. In those cases, the Event Fired business rules would see AccountAdded, AccountChanged, and AccountRemoved events if one or more destinations registered for those event names.

Following this table is another section called “Important BillingCenter Events and Related APIs” on page 327. That section describes special BillingCenter events and related APIs that technically are not messaging events.

| Entity | Events | Description |
|----------------------------|---|---|
| Account | AccountAdded AccountChanged AccountRemoved | Standard events for root entity Account. |
| | ResyncAccount | Resync the account. See “Resynchronizing Messages for a Primary Object” on page 353 |
| Activity | ActivityAdded ActivityChanged ActivityRemoved | Standard events for root entity Activity. The ActivityChanged event triggers after something changes an activity, including if a user marks it as completed or skipped. |
| AgencyBillWorkflow | AgencyBillWorkflowAdded AgencyBillWorkflowChanged AgencyBillWorkflowRemoved | Standard events for root entity AgencyBillWorkflow. |
| AuthorityLimitProfileAdded | AuthorityLimitProfileAddedAdded AuthorityLimitProfileAddedChanged AuthorityLimitProfileAddedRemoved | Standard events for root entity AuthorityLimitProfileAdded. |

| Entity | Events | Description |
|-------------------------|---|---|
| Charge | ChargeAdded ChargeChanged ChargeRemoved | Standard events for root entity Charge. |
| ChargeWrittenOffTxn | WriteoffReversed | Reversal of charge write-off |
| CmsnPayableReduction | CommissionWriteoffReversed | Reversal of commission write-off |
| CmsnReceivableReduction | CommissionWriteoffReversed | Reversal of commission write-off |
| CollectionAgency | CollectionAgencyAdded CollectionAgencyChanged CollectionAgencyRemoved | Standard events for root entity CollectionAgency. |
| DelinquencyProcess | DelinquencyProcessAdded DelinquencyProcessChanged DelinquencyProcessRemoved | Standard events for root entity DelinquencyProcess. |
| | ConnectCollectionAgency | Triggers if the collection agency on this delinquency process needs to connect to the account. |
| De1ProcessWorkflow | De1ProcessWorkflowAdded De1ProcessWorkflowChanged De1ProcessWorkflowRemoved | Standard events for root entity De1ProcessWorkflow (delinquency process workflow). |
| DirectBillMoneyRcvd | PaymentMoneyReceivedAdded PaymentMoneyReceivedChanged PaymentMoneyReceivedRemoved | Standard events for root entity DirectBillMoneyRcvd. |
| Document | DocumentAdded DocumentChanged DocumentRemoved | Standard events for root entity Document. Some implementations do not let users remove documents once they have been added, so the remove event may not be called. |
| Invoice | InvoiceAdded InvoiceChanged InvoiceRemoved | Standard events for root entity Invoice. |
| | InvoiceResent | This event triggers to resend an invoice. This does not trigger at the time the invoice originally sends. Users can <i>resend</i> an invoice from the BillingCenter user interface. BillingCenter increments the invoice's NumResends property and then triggers the InvoiceResent event. Nothing about the invoice changes as a result of a resend. However, the NumResends property and this event provide a hook for external systems to reprint invoices. You determine the logic for what to do with a resend in your Event Fired rule set message-generation code for this event. |
| InvoiceItem | InvoiceItemAdded InvoiceItemChanged InvoiceItemRemoved | Standard events for root entity InvoiceItem. |
| LineItem | LineItemAdded LineItemChanged LineItemRemoved | Standard events for root entity LineItem. |
| Note | NoteAdded NoteChanged NoteRemoved | Standard events for root entity Note. |
| OutgoingPayment | OutgoingPaymentAdded OutgoingPaymentChanged OutgoingPaymentRemoved | Standard events for root entity OutgoingPayment. |

| Entity | Events | Description |
|----------------------|---|--|
| | OutgoingPaymentStatusChanged | <p>Something created a payment or changed the payment status for an outgoing payment. The account.PaymentStatus property changed to another value of the PaymentStatus typelist, which includes:</p> <ul style="list-style-type: none"> accountpaid - an account was paid but funds have not been distributed. distributed - funds have been distributed open - account has not received any funds yet. |
| | | |
| PaymentMoneyReceived | PaymentMoneyReceivedAdded PaymentMoneyReceivedChanged PaymentMoneyReceivedRemoved | Standard events for root entity PaymentMoneyReceived. |
| PaymentRequest | PaymentRequestAdded PaymentRequestChanged PaymentRequestRemoved | Standard events for root entity PaymentRequest. |
| Plan | PlanAdded PlanChanged PlanRemoved | Standard events for root entity Plan. |
| Policy | PolicyAdded PolicyChanged PolicyRemoved | Standard events for root entity Policy. |
| PolicyPeriod | PolicyPeriodAdded PolicyPeriodChanged PolicyPeriodRemoved | Standard events for root entity PolicyPeriod. |
| | ResyncPolicyPeriod | Resync the policy period. See “Resynchronizing Messages for a Primary Object” on page 353 |
| Producer | ProducerAdded ProducerChanged ProducerRemoved | Standard events for root entity Producer. |
| | ResyncProducer | Resync the producer. See “Resynchronizing Messages for a Primary Object” on page 353 |
| ProducerStatement | ProducerStatementAdded ProducerStatementChanged ProducerStatementRemoved | Standard events for root entity ProducerStatement. |
| SuspensePayment | SuspensePaymentAdded SuspensePaymentChanged SuspensePaymentRemoved | Standard events for root entity SuspensePayment. |
| TAccount | TAccountAdded TAccountChanged TAccountRemoved | Standard events for root entity TAccount. |
| Transaction | TransactionAdded TransactionChanged TransactionRemoved | Standard events for root entity Transaction. |
| | | IMPORTANT: These events for Transaction trigger for all instances of transaction subtypes, such as PolicyWriteoffTxn and AbstractChargePaidTxn. |

| Entity | Events | Description |
|---|---|--|
| General-purpose events | | |
| Workflow | WorkflowAdded WorkflowChanged WorkflowRemoved | Standard events for Workflow entities. |
| ProcessHistory | ProcessHistoryAdded ProcessHistoryChanged ProcessHistoryRemoved | Some integrations can be done as a batch process. For example, use the event/messaging system to write records to a batch file (or rows to a database table) as each transaction processes. You submit the batch data to some downstream system. Write a batch process that starts manually or on a timer. To coordinate your software modules, use the ProcessHistory entity. After a batch process starts, a ProcessHistoryAdded event triggers. In your Event Fired code that processes the ProcessHistoryChanged event, check the <code>processHistory.CompletionTime</code> property for the date stamp in a <code>datetime</code> object. For more information about batch processes, see "Batch Processing" on page 107 in the <i>System Administration Guide</i> . |
| SOAPCallHistory | SOAPCallHistoryAdded SOAPCallHistoryChanged SOAPCallHistoryRemoved | Standard events for SOAPCallHistory entities. The application creates one for each incoming web service call. |
| StartablePluginHistory | StartablePluginHistoryAdded StartablePluginHistoryChanged StartablePluginHistoryRemoved | Standard events for StartablePluginHistory entities. The application creates these to track when a startable plugin runs. |
| InboundHistory | InboundHistoryAdded InboundHistoryChanged InboundHistoryRemoved | Standard events for root entity Invoice. |
| Administration events | | |
| Group | GroupAdded GroupChanged GroupRemoved | Standard events for root entity Group. |
| GroupUser | GroupUserAdded GroupUserChanged GroupUserRemoved | Standard events for root entity GroupUser. |
| Role | RoleAdded RoleChanged RoleRemoved | Standard events for root entity Role. |
| User | UserAdded UserChanged UserRemoved | Standard events for root entity User. The UserChanged event triggers only for changes made directly to the user entity, not for changes to roles or group memberships. A change to the user's contact record, for example a phone number, trigger a ContactChanged event. |
| UserRoleAssignment | UserRoleAssignmentAdded UserRoleAssignmentChanged UserRoleAssignmentRemoved | Standard events for root entity UserRoleAssignment. |
| Contacts and address book events | | |
| Adjudicator | ContactAdded ContactChanged ContactRemoved | Standard events for root entity Contact. |

| Entity | Events | Description |
|-------------------------|--|--|
| Company | ContactAdded ContactChanged ContactRemoved | Standard events for root entity Contact. |
| CompanyVendor | ContactAdded ContactChanged ContactRemoved | Standard events for root entity Contact. |
| Contact | ContactAdded ContactChanged ContactRemoved | Contact entities only exist as subtypes of Contact, such as Person. Those subtypes generate standard events for root entity Contact. |
| ContactAutoSyncWorkItem | ContactAutoSyncWorkItemAdded ContactAutoSyncWorkItemChanged ContactAutoSyncWorkItemRemoved | Standard events for root entity ContactAutoSyncWorkItem. |
| LegalVenue | ContactAdded ContactChanged ContactRemoved | Standard events for root entity Contact. |
| PersonVendor | ContactAdded ContactChanged ContactRemoved | Standard events for root entity Contact. |
| Place | ContactAdded ContactChanged ContactRemoved | Standard events for root entity Contact. |
| UserContact | ContactAdded ContactChanged ContactRemoved | Standard events for root entity Contact. |

At What Time Does a Remove-related Event Trigger?

The *EntityNameRemoved* events trigger after either of the following occurs:

- some code deletes an entity from BillingCenter
- some code marks the entity as *retired*. Retiring means a *logical delete* but leaves the entity record in the database. In other words, the row remains in the database and the *Retired* property changes to indicate that the entity data in that row is inactive. Only some entities in the data model are retrievable. Refer to the *Data Dictionary* for details.

Important BillingCenter Events and Related APIs

The following table lists APIs that seem like messaging events, including some messaging events and others implemented in a way other than messaging events.

| Type of data | To detect this change | Implement this API |
|--------------|-------------------------------------|---|
| Charges | A new charge | An Event Fired rule that listens for the ChargeAdded event. Remember to register a messaging destination to listen for that event. |
| Policies | A new policy | An Event Fired rule that listens for the PolicyAdded event. Remember to register a messaging destination to listen for that event. |
| | A policy transfers to a new account | An Event Fired rule that listens for the PolicyChanged event and check to see if the <i>Policy.Account</i> foreign key changed. Remember to register a messaging destination to listen for that event. |
| | A policy is canceled | An Event Fired rule that listens for the PolicyPeriodChanged event and check if <i>Policy.CancelStatus</i> changed value to either pendingcancelation to canceled. Remember to register a messaging destination to listen for that event. |

| Type of data | To detect this change | Implement this API |
|----------------|---|---|
| | A policy is delinquent | An Event Fired rule that listens for the <code>DelinquencyProcessPolicyPeriodAdded</code> event. Remember to register a messaging destination to listen for that event. |
| | A policy is written off | An Event Fired rule that listens for the <code>PolicyWriteoffTxnAdded</code> event. Remember to register a messaging destination to listen for that event. |
| Policy periods | A new policy period | An Event Fired rule that listens for the <code>PolicyPeriodAdded</code> event. Remember to register a messaging destination to listen for that event. |
| | Before a policy period closes | Implement the <code>IEventHandler</code> plugin interface method <code>beforePolicyPeriodClosed</code> . For more information, see “BillingCenter Application Event Customization Plugin” on page 170. |
| | A policy period closes | An Event Fired rule that listens for the <code>PolicyPeriodChanged</code> event. This rule checks if <code>PolicyPeriod.ClosureStatus</code> went from the value <code>open</code> to <code>closed</code> . Remember to register a messaging destination to listen for that event. |
| | A policy period transfers to a new producer | Implement the <code>IEventHandler</code> plugin interface method <code>policyPeriodTransferredToNewProducer</code> . For more information, see “BillingCenter Application Event Customization Plugin” on page 170. This does not use messaging events because changes from transfers of a policy period between producer codes are too complex to represent as property/array changes in a ruleset. |
| | A policy period's payment schedule changes | An Event Fired rule that listens for the <code>PolicyPeriodChanged</code> event. This rule checks if <code>PolicyPeriod.PaymentPlan</code> changed. To see if an individual invoice item changed, listen for the <code>InvoiceItemChanged</code> event instead. However, invoice item events can be numerous if a payment plan refactors as part of a <code>PaymentPlan</code> change. To isolate manual invoice item refactors, check that the <code>PaymentPlan</code> did not change. Remember to register a messaging destination to listen for these events. |
| | A policy period's charges were paid | An Event Fired rule that listens for the <code>AbstractChargePaidTxnAdded</code> event, and check if <code>AbstractChargePaidTxn.TAccountOwner</code> is a <code>PolicyPeriod</code> . Remember to register a messaging destination to listen for that event. |
| Invoices | An invoice is sent | In Event Fired rules that listen for the <code>InvoiceChanged</code> event, check if <code>Invoice.Status</code> changes from value <code>planned</code> to <code>billed</code> . If the <code>Status</code> property changes from <code>planned</code> to <code>carriedforward</code> , the invoice's items carry forward. If the <code>Status</code> property changes from <code>planned</code> to <code>writtenoff</code> , the invoice's items were written off. Remember to register a messaging destination to listen for this event. |
| | An invoice is due / overdue | In Event Fired rules that listen for the <code>InvoiceChanged</code> event, check if <code>Invoice.Status</code> changes from value <code>planned</code> to <code>billed</code> . If the <code>Status</code> property changes from <code>billed</code> to <code>due</code> , the invoice is due or overdue. Remember to register a messaging destination to listen for this event. |
| | An invoice's due date changes | In Event Fired rules that listen for the <code>InvoiceChanged</code> event, check if <code>Invoice.PaymentDueDate</code> changes. Remember to register a messaging destination to listen for this event. |
| Invoice items | An invoice item changes | See “A policy period's payment schedule changes.” earlier. |
| Accounts | A new account | An Event Fired rule that listens for the <code>AccountAdded</code> event. Remember to register a messaging destination to listen for that event. |

| Type of data | To detect this change | Implement this API |
|---------------|------------------------------------|---|
| | An account closes | An Event Fired rule that listens for the AccountAdded event and check if Account.Closed went from false to true. Remember to register a messaging destination to listen for that event. |
| | Charges paid for an account | An Event Fired rule that listens for the AbstractChargePaidTxnAdded event, and check if AbstractChargePaidTxn.TAccountOwner is an Account. Remember to register a messaging destination to listen for that event. |
| Payments | A payment was received | An Event Fired rule that listens for the PaymentStatusChanged event. Remember to register a messaging destination to listen for that event. |
| | A payment was distributed | An Event Fired rule that listens for the PaymentChanged event and check if the Payment.Status property changed from accountpaid to distributed. Remember to register a messaging destination to listen for that event. |
| Producers | A producer was paid | An Event Fired rule that listens for the ProducerPaidAdded event. Remember to register a messaging destination to listen for that event. |
| Checks | A new check | An Event Fired rule that listens for the CheckAdded event, although a CheckStatusChanged event also triggers for new checks. Remember to register a messaging destination to listen for that event. |
| | A check status changed | An Event Fired rule that listens for the CheckStatusChanged event. Remember to register a messaging destination to listen for that event. |
| Transactions | A new transaction | An Event Fired rule that listens for the TransactionAdded event. Remember to register a messaging destination to listen for that event. |
| Delinquency | An account becomes past due | An Event Fired rule that listens for the DelinquencyProcessAdded event. An account becoming past due is equivalent to a delinquency process being started for that account. Remember to register a messaging destination to listen for that event. |
| | A delinquency grace period expires | A grace period is a configurable, optional part of the delinquency process, so there is no explicit event generated if a grace period expires. Code that relates to grace period exist mostly within workflow process. Thus, one way you can handle this event is to listen for the WorkflowChanged event. In your event-handling code, check if the Workflow.currentStep property changed from the grace period step to a new step. Remember to register a messaging destination to listen for that event. |
| Disbursements | A disbursement (refund) is paid | An Event Fired rule that listens for the events OutgoingPaymentAdded and OutgoingPaymentStatusChanged. Remember to register a messaging destination to listen for each event. |
| Commissions | Before commission payment | Implement the IEventHandler plugin interface method beforeCommissionPayable. For more information, see "BillingCenter Application Event Customization Plugin" on page 170. |

| Type of data | To detect this change | Implement this API |
|---|---|---|
| Insufficient funds | Insufficient funds notification was processed | An Event Fired rule that listens for the PaymentChanged event. Remember to register a messaging destination to listen for that event. |
| Write-off reversals for charges and commissions | Reversals of an object of type ChargeWrittenOffTxn, CmsnPayableReduction, or CmsnReceivableReduction. | An Event Fired rule that listens for the events WriteoffReversed (for charges) and CommissionWriteoffReversed (for commissions). Remember to register a messaging destination to listen for each event. |

For integration points based on plugin interfaces, see “Plugin Overview” on page 135. That topic contains an overview of plugin interfaces, including links to appropriate documentation sections.

Triggering Custom Event Names

Business rules can trigger custom events using the `addEvent` method of account entities and most other entities. This method can also be called from Java code that uses the Java API libraries, specifically from Java plugins or from Java classes called from Gosu.

You might choose to create custom events in response to actions within the BillingCenter application user interface or using data changes originally initiated from the web service APIs. Triggering custom events is useful also if you want to use event-based rules to encapsulate code that logically represents one important action that could generate events. You could handle the event in your Event Fired rules but trigger it from another rule set such as validation, or from PCF pages.

To raise custom events within Gosu, use the `addEvent` method of the relevant object. In other words, call `myEntity.addEvent(eventname)`. In this case, the event name is whatever `String` you pass as the parameter. The entity whose `addEvent` method you call is the root object for the event.

You might also want to trigger custom events during message acknowledgement. If acknowledging the message from a messaging plugin, use the entity `addEvent` method described earlier.

Custom Events From SOAP Acknowledgements

Integrations that use SOAP API to acknowledge a message can use a separate mechanism for triggering custom events as part of the message acknowledgement.

First, your web service API client code in Java creates a new SOAP entity `Acknowledgement`. Next, call its `setCustomEvents` method to store a list of custom events to trigger as part of the acknowledgement:

```
String[] myCustomEvents = {"ExternalABC", "ExternalDEF", "ExternalGHI"};
myAcknowledgement.setCustomEvents(myCustomEvents);
```

Then, submit the acknowledgement using the SOAP APIs:

```
messagingToolsAPI.acknowledgeMessage(myAcknowledgement);
```

How Custom Events Affect Pre-Update and Validation

Be aware that pre-update rules do not run solely because of a triggered event. An entity’s pre-update rules run only if actual entity data changes. In cases where triggered events do not correspond to entities with modified properties, the event firing alone does not trigger pre-update rules. This does not affect most events, since almost all events correspond to entity data changes.

However, this affects for custom event firing through the `addEvent` entity method if no other entity properties actually changed. If you require pre-update rules to run as part of custom events, you must modify some property on the entity, or else those rule sets do not run.

No Events from Import Tool

The web services interface `ImportToolsAPI` and the corresponding `import_tools` command line tool are a generic mechanism for loading system data or sample data into the system. Events do not trigger in response to data added or updated using this interface. Be very careful about using this interface for loading important business data where events might be expected for integration purposes. You must use some other system to ensure your external systems are up to date with this newly-loaded data.

Generating New Messages in Event Fired Rules

Each time a system event triggers a messaging event, BillingCenter calls the Event Fired rule set. The application calls this rule set once for each event/destination pair for destinations that are interested in this event. Remember that destinations signal which events they care about in the Messaging editor in Studio, which specifies your messaging plugins by name. The *plugin name* is the name for which Studio prompts you when you register a plugin in the Plugins Editor in Studio. Your Event Fired rules must decide what to do in response to the event. Most importantly, decide whether you want to create a message in response to the event. Because message creation impacts user response times, avoid unnecessarily large or complex messages if possible.

The most important object your Event Fired rules use is a *message context object*, which you can access using the `messageContext` variable. This object contains information such as the event name and destination ID. Typically your rule set generates one or more messages, although the logic can omit creating messages as appropriate. The following sections explain how to use business rules to analyze the event and generate messages.

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Restrictions on Entity Data in Messaging Rules and Messaging Plugins” on page 313.

Studio includes a tool that helps you export business data entities (and other types like Gosu classes) to XML. You can select which properties are required or optional for each integration point. You can export an XSD to describe the data interchange format you selected. Then, you can edit your Event Fired rules to generate a payload for the entity that conforms to your custom XSD. For more details, see “Creating XML Payloads Using Guidewire XML (GX) Models” on page 336.

Rule Set Structure

If you look at the sample Event Fired rule set, you can see a suggested hierarchy for your rules. The top level creates a different branch of the tree for each destination. You can determine which destination this event applies to by using the `messageContext` variable accessible from Event Fired rule sets. For example, to check the destination ID number, use code like the following:

```
// If this is for destination #1  
messageContext.DestID == 1
```

At the next level in the rules hierarchy, it determines for what root object an event triggered:

```
messageContext.Root typeis Account // If the root object is a Account...
```

Finally, at the third level there is a rule for handling each event of interest:

```
messageContext.EventName == "AccountChanged"
```

In this way, it is easy to organize the rules and keep the logic for handling any single event separate. Of course, if you have shared logic that would be useful to processing multiple events, create a Gosu class that encapsulates that logic. Your messaging code can call the shared logic from each rule that needs it.

Simple Message Payload

There are multiple steps in creating a message. First, you must convert (*cast*) the root object of the event to a variable of known type:

```
var account = messageContext.Root as Account
```

Once the Rule Engine recognizes the root object as a Account, it allows you to access properties and methods on the account to parameterize the payload of your message.

Next, create a message with a String payload:

```
var msg = messageContext.createMessage("The account number is " + account.AccountNumber +
" and event name is " + messageContext.EventName)
```

Message Payloads and Setting Message Root or Primary Entities

If you want to use the safe ordering feature of BillingCenter, you may need additional lines of code.

For example, suppose you want to use the default message root for an event that operates on a PolicyPeriod object. However, you want to safe order the message on a destination with Account as the primary entity. Your message creation code in Event Fired rules looks like the following:

```
var policyPeriod = messageContext.Root as PolicyPeriod
var message = messageContext.createMessage(policyPeriod.DisplayName) // generate your payload somehow
message.Account = policyPeriod.Account // set a primary entity property
```

Now suppose you want to set the message root to a specific PolicyPeriod object that is not the object root object. As with the previous example, assume the message is for a destination with Account as the primary entity. Your message creation code in Event Fired rules looks like the following:

```
var message = messageContext.createMessage(policyPeriod.DisplayName) // generate your payload somehow
message.MessageRoot = policyPeriod
message.Account = policyPeriod.Account
```

For more information about these concepts, see:

- “Root Object” on page 305
- “Primary Entity and Primary Object” on page 305
- “Setting a Message Root Object or Primary Object” on page 335

Multiple Messages for One Event

The Event Fired rule set runs once for each event/destination pair. Therefore, if you need to send multiple messages, create multiple messages in the desired order in your Event Fired rules. For example:

```
var msg1 = messageContext.createMessage("Message 1 for account " + account.PublicID +
" and event name " + messageContext.EventName)
var msg2 = messageContext.createMessage("Message 2 for account " + account.PublicID +
" and event name " + messageContext.EventName)
```

You can also use loops or queries as needed. For example, suppose that if a account-related event occurs, you want to send a message for the account and then a message for each note on the account. The rule might look like the following:

```
var account = messageContext.Root as Account
var msg = messageContext.createMessage("message for account with public ID " + account.PublicID)

for (note in account.Notes) {
    msg = messageContext.createMessage(note.Body)
}
```

This creates one message for the account and also one message for *each* note on the account.

If you create multiple messages for one event like this, you can share information easily across all of the messages. For example, you could determine the username of the person who made the change, store that in a variable, and then include it in the message payload for all messages.

Remember that if multiple destinations requested notification for a specific event name, your Event Fired rule set runs once for each destination, varying only in the messageContext.DestID.

You might need to share information across multiple runs of the Event Fired rule set for the same event or different events. If so, see “Saving Intermediate Values Across Rule Set Executions” on page 333.

Determining What Changed

In addition to normal access to the `messageContext`'s root object, there is a way to find out what has changed. Your Gosu business rule logic can determine which user made the change, the timestamp, and the original value of changed properties. This information is available only at the time you originally generate the message (*early binding*).

Note: You cannot use the `messageContext` object during processing of late bound properties, which are properties that employ *late binding* immediately before sending. For more information about late binding, see “Late Binding Data in Your Payload” on page 342.

At the beginning of your code, use `isFieldChanged` method test whether the property changed. If the field changed (and only if the property changed), call the `getOriginalValue` function to get the original value of that property. To get the new (changed) value, simply access the property directly on an entity. The new value has not yet been committed to the database. There are additional functions similar to `isFieldChanged` and `getOriginalValue` that are useful for array properties and other situations. Refer to “Determining What Data Changed in a Bundle” on page 348 in the *Gosu Reference Guide* for the complete list.

For example, the following Event Fired rule code checks if a desired property changed, and checks its original value also:

```
Var usr = User.util.getCurrentUser() as User  
  
Var msg = "Current user is " + usr.Credential.UserName + ".  
msg = msg + " current account display name is " + account.DisplayName  
  
if (account.isFieldChanged("DisplayName")) {  
    msg = msg + " old value is " + (account.getOriginalValue("DisplayName"))  
}
```

For a complete list of Gosu methods related to finding what data changed, see “Determining What Data Changed in a Bundle” on page 348 in the *Gosu Reference Guide*.

Rule Sets Must Never Call Message Methods for ACK, Error, or Skip

From within rule sets, you must never call any message acknowledgment or skipping methods such as the `Message` methods `reportAck`, `reportError`, or `skip`. Use those `Message` methods only within messaging plugins.

This prohibition also applies to Event Fired rules.

Saving Intermediate Values Across Rule Set Executions

A single action in the user interface can generate multiple events that share some of the same information. Imagine that you do some calculation to determine the user's ID in the destination system and want to send this in all messages. You cannot save that in a variable in a rule and use it in another rule. The built-in scope of variables within the rule engine is a single rule. You cannot use the information later if the rule set runs again for another event caused by the same user interface action. BillingCenter solves this problem by providing a `HashMap` that you can access across multiple rule set executions for the same action that triggered the system event.

There are two versions of this API, which are methods on the object returned by the Gosu expression `messageContext.SessionMarker`. Both APIs create a hash map that exists for multiple Event Fired rules executing in a single database transaction triggered by the same system event. However, there is an important difference:

- To write to a hash map that exists for the lifetime of all rules for all destinations, call the method `addToSessionMap(key, value)`. To read from the hash map, call the `getFromSessionMap(key)` method.

- To write to a hash map that exists for the lifetime of all rules for the current messaging destination only, call the method `addToTempMap(key, value)`. To read from the hash map, call the `getFromTempMap(key)` method.

For example, suppose that in a single action, an activity completes and it creates a new note. This change causes two different events and hence two separate executions of the `EventFired` rule set. As BillingCenter executes rules for completing the activity, your rule logic could save the subject of the activity by adding it to the *temporary map* using the `SessionMarker.addToTempMap` method. Later, if the rule set executes for the new note, your code checks if the subject is in the `HashMap`. If it is in the map, your code adds the subject of the activity to the message for the note.

Code to save the activity's information would look like the following:

```
var session = messageContext.SessionMarker // get the sessionmarker
var act = messageContext.Root as Activity // get the activity

// Store the subject in the "temporary map" for later retrieval!
session.addToTempMap("related_activity_subject", act.Subject)
```

Later, to retrieve stored information from the `HashMap`, your code would look like the following:

```
var session = messageContext.SessionMarker // get the sessionmarker

// Get the subject line from the "temporary map" stored earlier!
var subject = session.getFromTempMap("related_activity_subject") as String
```

If you need to add an entity instance to the bundle, explicitly add it to the bundle. Get the correct bundle using the Gosu expression `messageContext.Bundle`. Add entities to the bundle before adding it to the hash map. For example:

```
var findResult = mycompany.QueryUtils.findRelatedObject().AtMostOneRow /* your own database query */
var resultToAdd = (findResult == null) ? null : messageContext.Bundle.add(findResult)
messageContext.SessionMarker.addToTempMap("MyKey", resultToAdd)
```

Creating a Payload Using Gosu Templates

You can use Gosu code in business rules to generate Gosu strings using concatenation to design *message payloads*, which are the text body of a message. Generating your message payloads directly in Gosu offers more control over the logic flow for what messages you need and for using shared logic in Gosu classes.

However, sometimes it is simpler to use a text-based template to generate the message payload text. This is particularly true if the template contains far more static content than code to generate content. Also, templates are easier to write than constructing a long string using concatenation with linefeed characters. Particularly for long templates, templates expose static message content in simple text files. People who might not be trained in Guidewire Studio or Gosu coding can easily edit these files.

You can use Gosu templates from within business rules to create some or all of your message payload.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. Your fully-qualified name of the template is `mycompany.templates.NotifyAdminTemplate`.

Use the following code to generate (run) a template and pass a parameter:

```
var myAccount = messageContext.Root as Account;

// generate the template and pass a parameter to the template
var x = mycompany.templates.NotifyAdminTemplate.renderToString(myAccount)

// create the message
var msg = messageContext.createMessage("Test my template content: " + x)
```

This code assumes the template supports parameter passing. For example, something like this:

```
<%@ params(myAccountParameter : Account) %>
The Account Number is <%= myAccountParameter.AccountNumber %>
```

There are a couple of steps. First, select the template. Then, you let templates use objects from the template's Gosu context using the `template.addSymbol` method. Finally, you execute the template and get a `String` result you could use as the message payload, or as part of the message payload.

The `addSymbol` method takes the *symbol name* that is available from within the template's Gosu code, an object type, and the actual object to pass to the template. The object type could be any intrinsic type, including BillingCenter entities such as Account or even a Java class.

For more information about using templates, see "Gosu Templates" on page 359 in the *Gosu Reference Guide*.

Setting a Message Root Object or Primary Object

From the Gosu environment, the `messageContext.Root` property specifies the *root object* for an event. Typically this same object also the root object for the message generated for that event. Because that is the most common case, by default any new message gets the same root object as the event object root. The message root indicates which object this message is about. For further definition of a root object, see "Root Object" on page 305. Contrast this with the definition of primary object on "Primary Entity and Primary Object" on page 305.

You can override the message root object to be a different object. For example, suppose you added a subobject and caught the related event in your Event Fired rules and added a message. You might want the message root to be the subobject's parent object instead of the default behavior. To override the message root, set the message's `MessageRoot` (not `Root`) property in your Event Fired rules. For example:

```
message.MessageRoot = myObject
```

It is important to note that the *primary object* of a message is different from the *message root*, however. It is actually the primary object of a message that defines the message ordering algorithm. See "Safe Ordering" on page 306 and "Message Ordering and Performance Tuning Details" on page 339.

Unlike the message root, there is not a single property that implements the primary object data for a message. Instead, there are multiple properties on a `Message` object that correspond to each type of data that could be a primary object for that application. As mentioned in "Primary Entity and Primary Object" on page 305, each application can define a default primary entity. Each messaging destination can choose from a small set of primary entities. The properties on the `Message` object for primary entity are strongly typed to the type of the primary entity. In other words, there is a different property on `Message` for each primary entity type.

In BillingCenter, there are multiple properties on `Message` for the primary entity:

- `message.Account` – The account, if any, associated with this `Message` object.
- `message.PolicyPeriod` – The policy period, if any, associated with this `Message` object.
- `message.Producer` – The producer, if any, associated with this `Message` object.
- `message.Contact` – The contact, if any, associated with this `Message` object.

In BillingCenter, if you set the `message.MessageRoot` property, the following behavior occurs automatically as a side-effect of setting this property from Gosu or Java:

- If the entity type is `Account`, BillingCenter sets the `message.Account` property to the account.
- If the entity type is `Contact`, BillingCenter sets the `message.Contact` property to the contact.
- If the entity type is `PolicyPeriod`, BillingCenter sets the `message.PolicyPeriod` property to the policy period.
- If the entity type is `Producer`, BillingCenter sets the `message.Producer` property to the producer.

You only need to set the primary entity properties manually if the automatic behaviors described in this topic did not set them already. Note that you do not need to set unused primary entity properties to `null`. The only primary entity property used in the `Message` object is the one that matches the primary entity for the destination.

To configure custom behavior of the message ordering system (safe ordering) by manually overriding properties that reference a primary object:

- Set the alternative primary entity in the messaging destination .

- In your Event Fired rules, manually set the primary entity property on the `Message` object that matches the primary entity for that destination. You can set a primary entity property to `null` instead of an object. If the property for the primary entity for that destination is `null`, then BillingCenter treats the message as a *non-safe-ordered* message. BillingCenter orders safe-ordered and non-safe-ordered messages differently. For details, see “Message Ordering and Multi-Threaded Sending” on page 337.

For example, suppose you want to use the default message root for an event that operates on a `PolicyPeriod` object. However, you want to safe order the message on a destination with `Account` as the primary entity. Your message creation code in Event Fired rules looks like the following:

```
var policyPeriod = messageContext.Root as PolicyPeriod  
var message = messageContext.createMessage(policyPeriod.DisplayName) // generate your payload somehow  
message.Account = policyPeriod.Account // set a primary entity property
```

Now suppose you want to set the message root to a specific `PolicyPeriod` object that is not the object root object. As with the previous example, assume the message is for a destination with `Account` as the primary entity. Your message creation code in Event Fired rules looks like the following:

```
var message = messageContext.createMessage(policyPeriod.DisplayName) // generate your payload somehow  
message.MessageRoot = policyPeriod  
message.Account = policyPeriod.Account
```

WARNING Be careful with setting message properties that store a reference to a primary object.

BillingCenter uses that information to implement safe ordering of messages by primary object.

Creating XML Payloads Using Guidewire XML (GX) Models

Studio includes a tool that helps you export business data entities (and other types like Gosu classes) to XML. You can select which properties are required or optional for each integration point. You can export an XSD to describe the data interchange format you selected. Then, you can use this model to export XML or import XML in your integrations. For example, your messaging plugins or your Event Fired rules could send XML to external systems. You could also write web services that take XML data its payload from an external system or return XML as its result.

The output XML only includes the properties specified in your custom XSD. It is best to create a custom XSD for each integration. Part of this is to ensure you send only the data you need for each integration point. For example, a check printing system probably needs a smaller subset of object properties than a external legacy financials system might need.

The first step is to create a new XML model in Studio. In Studio, navigate in the resource tree to the package hierarchy in which you want to store your XML model. Next, right-click on the package and from the contextual menu choose **New → Guidewire XML Model**.

For instructions on using the GX modeler, see “The Guidewire XML (GX) Modeler” on page 310 in the *Gosu Reference Guide*.

Using Java Code to Generate Messages

Business rules, including message-generation rules, can optionally call out to Java modules to generate the message payload string. See “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.

Saving Attributes of the Message

As part of creating a message, you can save a *message code* property within the message to help categorize the types of messages that you send. Optionally, you can use this information to help your messaging plugins know how to handle the message. Alternatively, your destination could report on how many messages of each type were processed by BillingCenter (for example, for reconciliation).

If you need additional properties on the `Message` entity for messaging-specific data, extend the data model with new properties. Only do this for messaging-specific data.

During the `send` method of your message transport plugin, you could test any of these properties to determine how to handle the message. As you acknowledge the message, you could compare values on these properties to values returned from the remote system to detect possible mismatches.

BillingCenter also lets you save *entities by name* (saving references to objects) with the message to update BillingCenter entities as you process acknowledgements. For example, to save a `Note` entity by the name `note1` to update a property on it later, use code similar to the following:

```
msg.putEntityByName("note1", note)
```

For more information about using `putEntityByName`, see “[Reporting Acknowledgements and Errors](#)” on page 343.

These methods are especially helpful to handle special actions in acknowledgements. For example, to update properties on an entity, use these methods to authoritatively find the original entity. These methods work even if public IDs or other properties on the entity change. This approach is particularly useful if public ID values could change between the time Event Fired rules create the message and the time you messaging plugins acknowledge the message. The `getEntityByName` method always returns the correct reference to the original entity.

Maximum Message Size

In this version of BillingCenter, messages can contain up to one billion characters.

Message Ordering and Multi-Threaded Sending

This section explains in detail how BillingCenter orders messages and in some cases uses multiple threads.

For an overview of safe ordering, see the following topics:

- “Primary Entity and Primary Object” on page 305
- “Safe Ordering” on page 306
- “Setting a Message Root Object or Primary Object” on page 335

The application waits for an acknowledgement before processing the next safe-ordered message for that primary object for each destination. This allows BillingCenter to send messages as soon as possible and yet prevent prevents errors that might occur if related messages send out of order.

For example, suppose some external system must process an initial new message for a parent object before receiving any messages for subobjects or notes that relate to the parent object. If the external system rejects the new message for the parent object, it is not safe to send further messages to that system. However, it is likely safe to send messages about unrelated objects, or messages about the same object but to a different destination.

Even if the transport layer guarantees delivery order, it is unsafe for BillingCenter to send the second message before confirming that the first safe-ordered message succeeds. Doing otherwise could cause difficult error recovery problems.

Safe ordering has large implications for messaging performance. For a destination configured with `Account` as the primary entity, suppose the send queue contains 10 messages associated with different accounts, BillingCenter can send these 10 safe-ordered messages immediately. However, if the send queue contains 10 safe-ordered messages for the same account, BillingCenter can only send one. BillingCenter must wait for the message acknowledgement, and then at that point can send the next (only one) message for that account. Another way of thinking about is that only one message can be in flight for each account/destination pair.

If you license the ContactManager application for use with BillingCenter, note that ContactManager also supports safe ordering of messages associated with each ABContact entity. This means that ContactManager only allows one message for each combination of ABContact and destination pair at any given time. For more ContactManager integration information, see “ContactManager Integration Reference” on page 227 in the *Contact Management Guide*.

Ordering Non-safe-ordered Messages

Some messages are not associated with the primary object for a destination. The potential primary entities for BillingCenter are Account, Producer, PolicyPeriod, or Contact. Any messages for a destination that do not have an associated primary entity are called *non-safe-ordered messages*.

These messages are sometimes also called Messages Without Primary.

See the beginning of this topic for links to topics that define primary entities and safe ordering.

By default, non-safe-ordered messages send as soon as possible, before any queued safe-ordered messages, and send in the order that Event Fired rules generated them. By default BillingCenter does not wait for acknowledgements for non-safe-ordered messages before sending the next -safe-ordered message.

For example, BillingCenter sends a message about a new User immediately. BillingCenter never waits for acknowledgements for other messages before sending this message.

However, if you enable the Strict Mode feature for a destination in the Messaging editor in Studio, BillingCenter waits for the acknowledgement for every non-safe-ordered message. There are also destination options for multi-threaded sending of non-safe-ordered messages. For details, see “How Destination Settings Affects Ordering” on page 341.

For BillingCenter, messages for the following events are by default non-safe-ordered messages:

- Group events
- User events

If Multiple Events Fire, Which Message Sends First?

For each destination, the Event Fired rules run in the order that each destination’s configuration specifies in the Messaging editor in Studio. In general, messages send in the order of message creation in Event Fired rules. BillingCenter runs the Event Fired for one event name before the rules run again for the next listed event name. For messages with both the same event name and same destination, the message order is the order that your Event Fired rules create the messages. For more information about destination setup, see “Message Destination Overview” on page 315.

If you register a messaging plugin, you must register it in two places. First, register it in the plugin registry in the plugin editor. See “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 131 in the *Configuration Guide*.

In typical deployments, this means that the event name order in the destination setup is very important. Carefully choose the order of the event names in the destination setup. It is important to remember that changes to the ordering of the event names change the order of the events in Event Fired rules. Such changes can produce radical effects in the behavior of Event Fired rules if they assume a certain event order. For example, typical downstream systems want information about a parent object before information about the child objects.

The event name order in the destination setup is critical. Carefully choose the order of the event names in the destination setup. Be extremely careful about any changes to ordering event names in the destination setup. Changes in the event name order could change message order, and that can force major changes in your Event Fired rules logic.

Because BillingCenter supports *safe-ordering* of messages related to a primary object, the actual ordering algorithm is more complex. Refer to “Message Ordering and Multi-Threaded Sending” on page 337 for details.

Message Ordering and Performance Tuning Details

BillingCenter pulls messages from the database (the send queue) in batches of messages on the batch server only, and then waits for a polling interval before querying again.

You can configure the number of messages that the messaging subsystem retrieves in each round of sending. This is called the chunk size. Configure the chunk size in the messaging destination configuration editor in the **Chunk Size** field. By default, this value is set to 100,000, which typically includes all sendable messages currently in the send queue. You can also change the polling interval in messaging destination configuration editor in the **Polling Interval** field.

You must understand the difference between message readers and message sending threads:

- *Message readers* are threads that query the database for messages. Message readers use the *message send order* (typically this is equivalent to creation order). The message reader never loads more than the maximum number of messages in the chunk size setting at one time.
- *Message sender threads* are threads that actually call the messaging plugins to send the messages. The application supports multiple sender threads per messaging destination for safe-ordered messages. You can configure the number of sender threads for safe-ordered messages in the messaging destination configuration editor in the **Number Sender Threads** field.

The messaging ordering and sending architecture works as follows by default (see “How Destination Settings Affects Ordering” on page 341):

1. Each messaging destination has a worker thread that queries the database for messages for that destination only. In other words, *each destination has its own message reader*. Each message reader thread (each worker thread) acts independently.
2. The destination’s message reader queries the database for one batch of messages, where the maximum size is defined by the chunk size. BillingCenter does not use the chunk size value in the query itself. Instead, the message reader orders the results by send order and stops iterating across the query results after it retrieves that many messages from the database.

BillingCenter performs two separate queries:

- a. First, BillingCenter queries for messages associated with a primary object for that destination, also known as *safe-ordered messages*. The maximum number of messages returned for this query is also the chunk size. The database query itself ensures BillingCenter that no more than one message per primary object exists in this list. If another message for a primary object is sent but unacknowledged, no messages for that object appears in this list. This enforces the rule that no more than one message can be in-flight for each primary object per destination.

The query ensures that no two messages are in flight (sent but not yet acknowledged) for the same destination for the same primary object. So, if there are 100 messages for one primary object, the query only reads and dispatches one of those messages (out of a possible 100) to the destination subthreads.

- b. Next, BillingCenter queries for all messages not associated with the primary object for that destination, also known as *non-safe-ordered messages*. If the chunk size is not set high enough, the returned set is not the full set of non-claim-specific messages. Be aware the chunk size affects each query. It is not cumulative for safe-ordered and non-safe-ordered messages.

By default, the chunk size is set to 100,000, which is usually sufficient for customers. Be sure not to lower the chunk size too much. Typically there are dependencies between safe-ordered and non-safe-ordered messages on that destination. If the chunk size is too low, the non-safe-ordered message query might not retrieve all the non-safe-ordered messages that your safe-ordered messages rely upon. The downstream system might need to receive the message that applies to many primary objects before any other messages reference that information.

3. For each destination, the worker thread iterates through all non-safe-ordered messages for that destination, sending to the messaging plugins. Settings in the Messaging editor in Studio for each destination affect how non-safe-ordered messages are sent. The choices are single thread, multi-thread, or strict mode. Those settings affect how many threads send messages, and how the application handles errors. See “How Destination Settings Affects Ordering” on page 341.

After each worker thread finishes sending non-safe-ordered messages, it creates subthreads to send safe-ordered messages for that destination. Configure the number of threads in the messaging destination configuration editor in the **Number Sender Threads** field. Each worker thread distributes the list of safe-ordered messages to send to the subthreads.

Assigning the number of sender subthreads for a destination by default affects only the safe-ordered messages for that destination. For non-safe-ordered messages (also called Messages Without Primary), the rules depend on destination settings. See “How Destination Settings Affects Ordering” on page 341.

If the message is associated with a primary entity, during messaging operations you can optionally lock the primary entity at the database level. This can reduce some problems in edge cases in which other threads (including worker threads) try to modify objects associated with this same object.

For example, if using the Account primary entity, the system locks the Account during messaging.

BillingCenter checks the `config.xml` parameter `LockPrimaryEntityDuringMessageHandling`. If it is set to true, BillingCenter locks the primary entity during message send, during all parts of message reply handling, and while marking a message as skipped.

4. The message reader thread waits until all destination threads send all messages in the queues for each subthread.
5. BillingCenter checks how much time passed since the beginning of this round of sending (since the beginning of step 2) and sleeps the remainder of the polling interval. Configure the polling interval in the messaging destination configuration editor in the **Polling Interval** field. If the amount of time since the last beginning of the polling interval is `TIME_PASSED` milliseconds, and the polling interval is `POLLING_INTERVAL` milliseconds. If the polling interval since the last query has not elapsed, the reader sleeps for `(POLLING_INTERVAL - TIME_PASSED)` milliseconds. If the time passed is greater than the polling interval, the thread does not sleep before re-querying.

The message reader reads the next batch of messages. Begin this procedure again at step 2.

The polling interval setting critically affects messaging performance. If the value is low, the message reader thread sleeps little time or even suppresses sleeping between rounds of querying the database for more messages.

To illustrate how this works, compare the following situations.

Suppose there are two messaging destinations, and both destinations use Account as the primary entity. Also suppose the send queue contains 10 messages for each destination. For each destination, assume that there is no more than one message for each account. In other words, for each destination, there are 10 total messages related to 10 different accounts:

- Assuming the number of messages does not exceed the chunk size, each destination gets only one message for the account for that destination from the database. In this case, every message can be sent immediately because each message for that destination is independent because they are for different accounts. If the destination’s **Number Sender Threads** setting is greater than 1, BillingCenter distributes all account-specific messages to multiple subthreads. The length of the destination queue never exceeds the number of messages queried in each round of sending. The message reader waits until all sending is complete before repeating.

In contrast, suppose the messages for one destination includes 10 account-specific messages for the same account and 5 non-account-specific messages:

- Assuming the number of messages does not exceed the chunk size, BillingCenter reads all 5 non-safe-ordered messages and sends them. However, BillingCenter only gets one message for the account for that destination from the database. If the destination's **Number Sender Threads** setting is greater than 1, BillingCenter distributes all account-specific messages in multiple threads per destination. In this case there is only message that is sendable. Compared to the previous example, BillingCenter handles fewer messages for each polling interval.

To improve performance, particularly cases like the second example, change the following settings in the Messaging editor for your destinations:

- Lower the **Polling Interval**. The value is in milliseconds. Experiment with lower values perhaps as low as 1000 (which means 1 second) or even lower. Test any changes to see the real-world effects on your messaging performance. If your performance issues primarily relate to many messages per primary entity per destination, then the polling interval is the most important messaging performance setting.
- Increase the value for **Number Sender Threads**. This permits more worker threads to operate in parallel on the batch server only for sending safe-ordered messages. Again, test any changes to see the real-world effects on your messaging performance. If your performance issues primarily relate to many messages but few messages per primary entity for each destination, then the sending threads number is the most important messaging performance setting.

To get maximum performance from multiple threads, keep your message transport plugin implementation's `send` method as quick as possible, with little variation in duration. The application does not load the next chunk of messages until the application passes all messages in the chunk to the message transport `send` method. If your `send` method sometimes takes a long time, convert your code to asynchronous sending. With asynchronous sending, the `send` method completes quickly and the reply is handled later. See "Message Destination Overview" on page 315.

Thread-Safe Plugins

You must write all messaging plugin implementation code as *thread-safe code*. This means that you must be extremely careful about static variables and any other shared memory structures that multiple threads might access running the same (or related) code.

For more information, see "Concurrency" on page 381 in the *Gosu Reference Guide*.

You must write your messaging plugin code as thread-safe even if the **Number Sender Threads** setting is set to 1.

How Destination Settings Affects Ordering

For messages without a primary object (sometimes called non-safe-ordered messages), there are settings in the Messaging editor for each messaging destination that configures how to order them.

The **Message Without Primary** setting has three choices:

- Single thread** – Messages without a primary object send in a single thread, and do not wait for an acknowledgement before proceeding to other messages.
- Multi thread** – Messages without a primary object send in multiple threads, and do not wait for an acknowledgement before proceeding to other messages. The precise order of sending of messages without a primary object is non-deterministic.
- Strict Mode** – If Strict Mode is enabled, messages without a primary object send in a strict order, and wait for an acknowledgement before proceeding to other messages.

Carefully choose the value for each messaging destination.

WARNING If you use either **Single thread** or **Multi thread** options, errors for messages without a primary object do not hold up other messages. This means that errors in such messages may cause errors at the destination if the system is unprepared for this situation.

If Strict Mode is enabled, if errors occur other messages for that destination stop until an administrator resolves the problem. For example, an administrative user can resync that message. Resyncing the messages allows other messages to send for that primary object for that destination after resynchronizing that primary object. For more information about resynchronizing, see “Resynchronizing Messages for a Primary Object” on page 353.

With the value to **Single thread** or **Multi thread**, which means Strict Mode is disabled:

- Each destination sends non-safe-ordered messages in one or more threads, depending on whether you choose **Single thread** or **Multi thread**. Next, the application sends safe-ordered messages.
Note: Optionally, sending safe-ordered messages is multi-threaded. For more, see “Message Ordering and Multi-Threaded Sending” on page 337.
- Non-safe-ordered messages send in order but never wait for acknowledgement. Non-safe-ordered message errors never block other messages.

With Strict Mode on, the behavior is:

- Non-safe-ordered messages require an acknowledgement before sending future next message (of either type) to that destination. This option reduces throughput of non-safe-ordered messages.
- Delay sending safe-ordered messages until all non-safe-ordered messages are sent.
- Errors block all future messages for that destination, independent of the message type.

For each messaging destination, carefully consider the data integrity, error handling, and performance needs for your system before deciding on the value of the Message without Primary option.

Late Binding Data in Your Payload

In your Event Fired messages, in general it is best to use the current state of entity data to create the message payload. In other words, generate the entire payload when the Event Fired rule set runs. For example, for **AccountChanged** events, messages typically contain the latest information for the account as of the time the messaging event triggers. This includes any changes in this database transaction (entity instance additions, removals, or changes). Generally this is the best approach for messaging. If you waited until messaging sending time (rather than creation time), the data might be partially different or even data removed from the database. This could disrupt the series of messages to a downstream system. Downstream systems typically need messages that match with data model changes as they happen. Creating the entire payload at Event Fired time is the standard recommended approach. This is called *early binding*.

However, this prevents *later* changes to an object appearing in an *earlier* message about that object. This is relevant if there is a large delay in sending the message for whatever reason. Sometimes you need the latest possible value on an entity as the message leaves the send queue on its way to the destination. For example, imagine sending a new account to an external system. As part of the acknowledgement, the external system might send back its ID for the new account. You can set the *public ID* in BillingCenter to that external system’s ID for the account.

Suppose the next message separately sends information related to that account to the external system. In this message, you want to include the account’s new public ID received from the external system in the acknowledgement. The external system knows which account belongs with this second message. If the public ID were merely set during the original processing of the event, that message does not contain the new value from the external system. There would be no way to tell the external system which account this second set of information goes with.

BillingCenter solves this problem by permitting *late binding* of properties in the message payload. You can designate certain properties for late binding so you re-calculate values immediately before the messaging transport sends the message.

IMPORTANT Guidewire recommends exporting all data in the Event Fired rules into a message payload (early-bound) unless you have a specific reason why late binding is critical for that situation.

For newly-created entity instances, some customers send the entity instance's *public ID property* as a late bound property. A message acknowledgement or external system (using web service APIs) could change the public ID between creating (submitting) the message and sending it.

For other properties, decide whether early binding or late binding is most appropriate.

At message creation time in Event Fired rules, add your own marker text within the message, for example "<AAAAAA>". Your MessageRequest plugin code or MessageTransport plugin code can directly find the message root object or primary object and substitute the current value. If BillingCenter calls your MessageRequest plugin, the current value of the property is a late bound value and you can replace the marker with the new value.

For example, a Gosu implementation of the MessageRequest plugin interface might do this using the following code in the `beforeSend` method. In this example, the transport assumes the message root object is a Account and replaces the special marker in the payload with the value of extension property `SomeProperty`. This example assumes that the message contains the string "<AAAAAA>" as a special marker in the message text:

```
function beforeSend(m : Message) {  
    var c = m.MessageRoot as Account  
    var s = org.apache.commons.lang.StringUtils.replace(m.getPayload(), "<AAAAAA>", c.SomeProperty)  
    return s  
}
```

See also

- “Primary Entity and Primary Object” on page 305
- “Setting a Message Root Object or Primary Object” on page 335

Reporting Acknowledgements and Errors

BillingCenter expects to receive acknowledgements back from the destination. In most cases, the destination submits a *positive acknowledgement* to indicate success. However, errors can also occur, and you must tell BillingCenter about the issue.

Message Sending Error Behaviors

Sometimes something goes wrong while sending a message. Errors can happen at two different times. Errors can occur during the *send* attempt as BillingCenter calls the message sync transport plugin's `send` method with the message. For asynchronous replies, errors can also occur in negative acknowledgements.

Error conditions during the destination `send` process:

- **Exceptions during `send()` causes automatic retries** – 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, BillingCenter retries after a delay time, and continues to retry multiple times. The delay time is an exponential wait time (*backoff time*) up to a wait limit specified by each destination. For safe-ordered messages, BillingCenter halts sending messages all messages for that combination of primary object and destination during that retry delay. After the delay reaches the wait limit, the retryable error becomes a non-automatic-retry error.

- **For errors during send() and you do not want automatic retry** – If the destination has an error that the application does not expect to be temporary, do not throw an exception. Throwing an exception triggers automatic retry. Instead, call the message’s `reportError` method with no arguments. See “Submitting Ack, Errors, and Duplicates from Messaging Plugins” on page 344.

The destination suspends sending for all messages for that destination until one of the following is true:

- An administrator retries the sending manually, and it succeeds this time.
- An administrator removes the message.
- It is a safe-ordered BillingCenter message and an administrator resynchronizes the account, producer, or policy period.

Note: Although Contact can be a primary entity for a BillingCenter messaging destination, BillingCenter does not support resynchronizing a contact.

Errors conditions that occur later:

- **A negative acknowledgement (NAK)** – A destination might get an error reported from the external system (database error, file system error, and delivery failure) and human intervention might be necessary. For safe-ordered messages, BillingCenter stops sending messages for this combination of primary object and destination until the error clears through the administration console or automated tools.
 - **No acknowledgement for a long period** – BillingCenter does not automatically time out and resend the message because of delays. If the transport layer guarantees delivery, delay is acceptable whereas resending results in message duplicates. External system may not be able to properly detect and handle duplicates.
- For safe-ordered messages, BillingCenter does not send more messages for the combination of primary object and destination until it receives an acknowledgement (ACK) or some sort of error (NAK).

For BillingCenter administrative tools that monitor and recover from messaging errors, see “The Administration User Interface” on page 360.

Submitting Ack, Errors, and Duplicates from Messaging Plugins

To submit an acknowledgement or a negative acknowledgement from a messaging plugin implementation class, use the following APIs. Refer to the following table based on the success status, the type of failure, and the location of your code.

| Situation | Do this | Description |
|---|---|---|
| Report acknowledgement | | |
| Success | <code>message.reportAck()</code> | Submits an ACK for this message, which may permit other messages to be sent. For detailed logic of message ordering, see “Message Ordering and Multi-Threaded Sending” on page 337. |
| Errors within the send method of your <code>MessageTransport</code> plugin implementation, and the error is presumed temporary. | Throw an exception within the send method, which triggers automatic retries potentially multiple times. | <p>Automatically retries the message potentially multiple times, including the backoff timeout and maximum tries. After the maximum retries, the application ceases to automatically retry it and suspends the messaging destination.</p> <p>An administrator can retry the message from the Administration tab. Select the message and click Retry. For details of automatic retry, see “Message Sending Error Behaviors” on page 343.</p> |

| Situation | Do this | Description |
|--|--|--|
| Report error | | |
| Errors in any messaging plugins and no automatic retry is needed | <code>message.reportError()</code> | The no-argument version of the <code>reportError</code> method reports the error and omits automatic retries. An administrator can retry the message from the Administration tab. Select the message and click Retry . For more information about the retry schedule, see “ Reporting Acknowledgements and Errors ” on page 343. |
| Errors in any messaging plugins and scheduled retry is needed | <code>message.reportError(date)</code> | Reports the error and schedules a retry at a specific date and time. An administrator can retry the message from the user interface before this date. This is equivalent to using the application user interface in the Administration tab at that specified time, and select the message and click Retry . You can call this method from the <code>MessageTransport</code> or when handling errors in your <code>MessageReply</code> plugin implementation. |
| Errors in any messaging plugins and scheduled retry is needed | <code>message.reportError(category)</code> | Reports the error and assigns an error category from the <code>ErrorCategory</code> typelist. The Administration tab uses this error category to identify the type of error in the user interface. You can extend this typelist to add your own meaningful values. In the base configuration of BillingCenter, this typelist is empty. You can call this method from the <code>MessageTransport</code> or when handling errors in your <code>MessageReply</code> plugin implementation. |

| Situation | Do this | Description |
|-------------------------|--|--|
| Report duplicate | | |
| Message is a duplicate | <code>msgHist.reportDuplicate()</code> | <p>Reports a duplicate. This is a method on the message history object, not the message object. A message history object is what a message becomes after successful sending. A message history (<code>MessageHistory</code>) object has the same properties as a message (<code>Message</code>) object but has different methods.</p> <p>If your duplicate detection code runs in the <code>MessageTransport</code> plugin (typical only for synchronous sending), use standard database query builder APIs to find the original message. Query the <code>MessageHistory</code> table.</p> <ul style="list-style-type: none"> For asynchronous sending with the <code>MessageReply</code> plugin implementation, The <code>MessageFinder</code> interface has methods that a reply plugin uses to find message history entities. The methods use either the original message ID or the combination of sender reference ID and destination ID: <code>findHistoryByOriginalMessageID(originalMessageId)</code> - find message history entity by original message ID <code>findHistoryBySenderRefID(senderRefID, destinationID)</code> - find message history entity by sender reference ID <p>After you find the original message in the message history table, report the duplicate message by calling <code>reportDuplicate</code> on the original message. For related information about asynchronous replies, see "Implementing a Message Reply Plugin" on page 350</p> |

Using Web Services to Submit Ackcs and Errors From External Systems

If you want to acknowledge the message directly from an external system, use the web service method `IMessagingTools.acknowledgeMessage(ack)`. First, create an `Acknowledgement` SOAP object.

If there are no problems with the message (it is successful), pass the object as is.

If you detect errors with the message, set the following properties:

- `Error` – Set the `Acknowledgement.error` property to `true`
- `Retryable` – For all errors other than duplicates, always set the `Acknowledgement.Retryable` property to `true` and `Acknowledgement.Error` to `true`. Set this property to `false` (the default) only if there is no error.
- `Duplicate` – If you detect the message is a duplicate, set the `Duplicate` and `Error` properties to `true`.

Using Web Services to Retry Messages from External Systems

The `MessagingToolsAPI` web service contains methods to retry messages.

Review the documentation for the `MessagingToolsAPI` methods `retryMessage` and `retryRetryableErrorMessages`. The method `retryRetryableErrorMessages` optionally limits retry attempts to a specified destination. You can only use the `MessagingToolsAPI` interface if the server's run mode is set to `multiuser`. Otherwise, all these methods throw an exception.

As part of an acknowledgement, the destination can update properties on related objects. For example, the destination could set the `PublicID` property based on an ID in the external system.

Tracking a Specific Entity With a Message

If desired, you can track a specific entity at message creation time in your Event Fired rules. You can use this entity in your messaging plugins during sending or while handling message acknowledgements. To attach an entity to a message in Event Fired rules, use the `Message` method `putEntityByName`. to attach an entity to this message and associate it with a custom ID called a *name*. Later, as you process an acknowledgement, use the `Message` method `getEntityByName` to find that entity attached to this message.

The `putEntityByName` and `getEntityByName` methods are helpful for handling special actions in an acknowledgements. For example, if you want to update properties on a certain entity, these methods authoritatively find the *original* entity that triggered the event. These methods work even if the entity's public ID or other properties changed. This is particularly useful if the public IDs on some objects changed between the time you create the message and the time the messaging code acknowledges the messaged. In such cases, `getEntityByName` always returns the correct entity.

For example, Event Fired rules could store a reference to an object with the name “abc:expo1”. In the acknowledgement, the destination would set the `publicID` property. For example, set the public ID of object abc:expo1 to the value “abc:123-45-4756:01” to indicate how the external system thinks of this object.

Saving this name is convenient because in some cases, the external system's name for the object in the response is known in advance. You do not need to store the object type and public ID in the message to refer back to the object in the acknowledgement.

Implementing Messaging Plugins

There are three types of messaging plugins. See the following sections for details:

- “[Implementing a Message Request Plugin](#)” on page 348
- “[Implementing a Message Transport Plugin](#)” on page 348
- “[Implementing a Message Reply Plugin](#)” on page 350

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “[Restrictions on Entity Data in Messaging Rules and Messaging Plugins](#)” on page 313.

Getting Message Transport Parameters from the Plugin Registry

It may be useful in some cases to get parameters from the plugin registry in Studio. The benefit of setting parameters for messaging transports is that you can separate out variable or environment-specific data from your code in your plugin.

For example, you could use the Plugins editor in Studio for each messaging plugin to specify the following types of data for the transport:

- external system's server name
- external system's port number
- a timeout value

If you want to get parameters from the plugin registry, your messaging plugin must explicitly implement `InitializablePlugin` in the class definition. This interface tells the application that you support the `setParameters` method to get parameters from the plugin registry. Your code gets the parameters as name/value pairs of `String` values in a `java.util.Map` object.

For example, suppose your plugin implementation's first line looks like this:

```
class MyTransport implements MessageTransport {
```

Change it to this:

```
class MyTransport implements MessageTransport, InitializablePlugin {
```

To conform to this new interface, add a `setParameters` method with the following signature:

```
function setParameters(map: java.util.Map) { // this is part of InitializablePlugin
    // access values in the MAP to get parameters defined in plugin registry in Studio
    var myValueFromTheMap = map["servername"]
}
```

Implementing a Message Request Plugin

A destination can optionally define a message request (`MessageRequest`) plugin to prepare a `Message` object before a message is sent to the message transport. It may not be necessary to do this step. For example, if textual message payload contains strings or codes that must be translated for a specific remote system. Or perhaps a textual message payload contains simple name/value pairs that must be translated into a XML before sending to the message transport. Or, perhaps you need to set data model extension properties on `Message`.

To prepare a message before sending, implement the `MessageRequest` method `beforeSend`. BillingCenter calls this method with the message entity (a `Message`).

The main task for this method is to generate a modified payload and return it from the method. Generally speaking, do not modify the payload directly in the `Message` entity. The result from this method passes to the messaging transport plugin to send in its `transformedPayload` parameter. This transformed payload parameter is separate from the `Message` entity that the message transport plugin gets as a parameter.

You can modify properties within the `Message` or within the message's root object such as a `Account` object as needed. However, as mentioned before, to transform the payload just return a `String` value from the method rather than modifying the `Message`.

You can also use the `MessageRequest` plugin to perform post-sending processing on the message. To perform this type of action, implement the `MessageRequest` method `afterSend`. BillingCenter calls the method with the message (a `Message` object) as a parameter. BillingCenter calls this method *immediately* after the transport plugin's `send` method completes. If you implement asynchronous callbacks with a message reply plugin, be sure to understand that the server calls `afterSend` in the same thread executing the plugin's `send` method. However, it might be a separate thread from any asynchronous callback code.

Also, if the `send` method acknowledges the message with any result (successful ACK or an error), then be aware that the ACK does not affect whether the application calls `afterSend`. Assuming no exceptions trigger during calls to `beforeSend` or `send`, BillingCenter calls the `afterSend` method.

If you want to get parameters from the plugin registry, your messaging plugin must explicitly implement `InitializablePlugin` in the class definition. This interface tells the application that you support the `setParameters` method to get parameters from the plugin registry. Your code gets the parameters as name/value pairs of `String` values in a `java.util.Map` object.

Implementing a Message Transport Plugin

A destination must define a message transport (`MessageTransport`) plugin to send a `Message` object over some physical or abstract transport. This might involve submitting a message to a message queue, calling a remote web service API, or might implement a complex proprietary protocol specific to some remote system. The message transport plugin is the only required plugin interface for a destination.

To send a message, implement the `send` method, which BillingCenter calls with the message (a `Message` object). That method has another argument, which is the *transformed payload* if that destination implemented a message request plugin. See “[Implementing a Message Request Plugin](#)” on page 348.

In the message transport plugin's simplest form, this method does its work synchronously entirely in the `send` method. For example, call a single remote API call such as an outgoing web service request on a legacy computer. Your `send` method optionally can immediately acknowledge the message with the code `message.reportAck(...)`. If there are errors, instead use the message method `reportError`. To report a duplicate message, use `reportDuplicate`, which is a method not on the current message but on the `MessageHistory` entity that represents the original message.

If you must acknowledge the message synchronously, your `send` method may optionally update properties on BillingCenter objects such as `Account`. If you desire this, you can get the message root object by getting the property `theMessage.MessageRoot`. Changes to the message and any other modified entities persist to the database after the `send` method completes. Changes also persist after the `MessageRequest` plugin completes work in its `afterSend` method. To visualize how these elements interact, see the diagram in “Message Destination Overview” on page 315.

If your message transport plugin `send` method does not synchronously acknowledge the message before returning, then this destination must also implement the message reply plugin. The message reply plugin to handle the asynchronous reply. See the “Implementing a Message Reply Plugin” on page 350 for details.

You must handle the possibility of receiving duplicate message notifications from your external systems. Usually, the receiving system detects duplicate messages by tracking the message ID, and returns an appropriate status message. The plugin code that receives the reply messages can call the `message.reportDuplicate()` method. Depending on the implementation, the code that receives the reply would be either the message transport plugin or the message reply plugin. Your code that detects the duplicate must skip further processing or acknowledgement for that message.

Your implementations of messaging plugins must explicitly implement `InitializablePlugin` in the class definition. This interface tells BillingCenter that you support the `setParameters` method to get parameters from the plugin registry. Even if you do not need parameters, your plugin implementation must implement `InitializablePlugin` or the application does not initialize your messaging plugin.

Your implementation of this plugin must explicitly implement `InitializablePlugin` in the class definition and then also add a `setParameters` method to get parameters. Implement this interface even if you do not need the parameters from the plugin registry.

The following example in Java demonstrates a basic message transport.

Example Basic Message Transport For Testing

```
uses java.util.Map;
uses java.plugin;

class MyTransport implements MessageTransport, InitializablePlugin {

    function setParameters(map: java.util.Map) { // this is part of InitializablePlugin
        // access values in the MAP to get parameters defined in plugin registry in Studio
    }

    // NEXT, define all your other methods required by the MAIN interface you are implementing...
    function suspend() {}

    function shutdown() {}

    function setDestinationID(id:int) {}

    function resume() {}

    function send(message:entity.Message, transformedPayload:String) {
        print("====")
        print(message)
        message.reportAck()
    }
}
```

More Examples

Several example message transport plugins ship with BillingCenter in the product in the following directory:

`BillingCenter/java-api/examples/src/examples/p1/plugins/messaging`

Exception Handling

For details of exceptions and handling suspect/shutdown in messaging plugins, see “Error Handling in Messaging Plugins” on page 352.

Implementing a Message Reply Plugin

A destination can optionally define a message reply (`MessageReply`) plugin to asynchronously acknowledge a Message. For instance, this plugin might implement a trigger from an external system that notifies BillingCenter that the message send succeeded or failed.

BillingCenter requires a special step in this process to setup the BillingCenter database transaction information appropriately so that any entity changes commit to the BillingCenter database. To do this properly, Guidewire provides the types `MessageFinder`, `PluginCallbackHandler`, and inner interface `PluginCallbackHandler.Block`.

A message finder (`MessageFinder`) object is built-in object that returns a `Message` entity instance from its `messageID` or `senderRefID`, using its `findById` or `findBySendRefId` method. During the message reply plugin initialization phase, BillingCenter calls the message reply plugin `initTools` method with an instance of `MessageFinder`. Save this instance of `MessageFinder` in a private variable within your plugin instance.

A plugin callback handler (`PluginCallbackHandler`) is an object provided to your message reply plugin. The callback handler safely executes the message reply callback code block. The callback handler sets up the callback’s server thread and the database transaction information so entity changes safely and consistently save to the database.

If you want to get parameters from the plugin registry, your messaging plugin implementation must explicitly implement `InitializablePlugin` in its class declaration. The `InitializablePlugin` interface declares that the class supports the `setParameters` method to get parameters from the plugin registry. The application calls the plugin method `setParameters` and passes the parameters as name-value pairs of `String` values in a `java.util.Map` object.

For important information about using message finders to submit errors and duplicates, see “Error Handling in Messaging Plugins” on page 352.

IMPORTANT BillingCenter includes multi-threaded inbound integration APIs that you can optionally use in conjunction with `MessageReply` plugins. For example, listen for JMS messages or process text files that represent message replies. If you want to use input data other than JMS messages or text files for message replies, write a custom integration by implementing the `InboundIntegrationMessageReply` plugin. `InboundIntegrationMessageReply` is a subinterface of `MessageReply`. See “Multi-threaded Inbound Integration Overview” on page 281 and “Custom Inbound Integrations” on page 292.

Message Reply Plugin Initialization

During initialization, the message reply plugin must get and store a reference to the message finder (`MessageFinder`) and callback handler (`PluginCallbackHandler`) objects, since it needs to use them later.

Implement the simple `MessageReply` interface and include the `initTools` method, which gets these objects as parameters. Your plugin implementation must store these values in private properties, for example in private properties `_pluginCallbackHandler` and `_messageFinder`.

Message Reply Callbacks

To acknowledge the message asynchronously, your message reply plugin uses its reference to the `PluginCallbackHandler`. To run your code, you plugin must pass a specially-prepared block of Java code to the `PluginCallbackHandler` method called `execute`.

The `execute` method takes a *message reply callback block*, which is an instance of `PluginCallbackHandler.Block`. The `Block` is a simple private interface to encapsulate the block of code that you write.

The `PluginCallbackHandler` object has an `add` method that marks a BillingCenter entity as modified so the application commits changes to the database with the message acknowledgement. Call the `add` method and pass an entity instance. This method adds the entity to the correct *bundle*. For more about bundles, see “Bundles and Database Transactions” on page 341 in the *Gosu Reference Guide*. This process ensures that changes to the entity instance commit to the database in the correct database transaction with related changes.

Your code in `PluginCallbackHandler.Block` must perform the following steps:

1. Find a `Message` object. Use methods on the plugin’s `MessageFinder` instance, described earlier in this section.

2. Call methods on the `Message` to signal acknowledgement or errors:

- To report success, call the `reportAck` method on the message.
- To report errors, call the `reportError` method on the message. There are multiple method signatures to accommodate automatic retries and error categories.
- To report duplicates, call the `reportDuplicate` method on the message history (`MessageHistory`) entity instance that corresponds to the original message.

For details, see “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 344.

3. Optional post-processing such as property updates or triggering custom events. If any objects must be modified other than the objects originally attached to the message, the callback block must call `PluginCallbackHandler.add(entityReference)`. This call to the `add` method ensures all changes on the other objects properly commit to the database as part of that database transaction. You must use the return result of the `add` method and only modify that return result. The return result is a clone of the object that is now writable. Do not continue to hold a reference to the original object you passed to the `add` method. This is equivalent to the `bundle.add(entityReference)` method, which adds an entity instance to a writable bundle. For more information, see “Adding Entity Instances to Bundles” on page 345 in the *Gosu Reference Guide*.

For example, the following example demonstrates creating a `PluginCallbackHandler.Block` object and executing the callback block.

```
...
PluginCallbackHandler.Block block = new PluginCallbackHandler.Block() {
    public void run() throws Throwable {
        Message message = _messageFinder.findById(messageID);
        message.reportAck();
    }
};

\PluginCallbackHandler.execute(block);
...
```

You can call `EntityFactory` as necessary in your callback handler block to create or find entities. The application properly sets up the thread context so that it supports `EntityFactory`. For more information about `EntityFactory`, see “Accessing Entity and Typecode Data in Java” on page 455.

For details of exceptions and handling suspend/shutdown in messaging plugins, see “Error Handling in Messaging Plugins” on page 352.

Error Handling in Messaging Plugins

For a summary of message error APIs, see “Reporting Acknowledgements and Errors” on page 343.

Several methods on messaging plugins execute in a strict order for a message. Consult the following list to design your messaging code, particularly error-handling code:

1. BillingCenter selects a message from the send queue on the batch server.
2. If this destination defines a `MessageRequest` plugin, BillingCenter calls `MessageRequest.beforeSend`. This method uses the text payload in `Message.payload` and transforms it and returns the transformed payload. The message transport method uses this transformed message payload later.
3. If `MessageRequest.beforeSend` made changes to the `Message` entity or other entities, BillingCenter commits those changes to the database, assuming that method threw no exceptions. The following special rules apply:
 - Committing entity changes is important if your integration code must choose among multiple pooled outgoing messaging queues. If errors occur, to avoid duplicates the message must always resend to the same queue each time. If you require this approach, add a data model extension property to the `Message` entity to store the queue name. In `beforeSend` method, choose a queue and set your extension property to the queue name. Then, your main messaging plugin (`MessageTransport`) uses this property to send the message to the correct queue.
 - If exceptions occur, the application rolls back changes to the database and sets the message to retry later. There is no special exception type that sets the message to retry (an un-retryable error).
 - In all cases, the application never explicitly commits the transformed payload to the database.
4. BillingCenter calls `MessageTransport.send(message, transformedPayload)`. The `Message` entity can be changed, but the transformed payload parameter is read-only and effectively ephemeral. If you throw exceptions from this method, BillingCenter triggers automatic retries potentially multiple times. This is the only way to get the automatic retries including the backoff multiplier and maximum retries detection.
5. If this destination defines a `MessageRequest` plugin, BillingCenter calls `MessageRequest.afterSend`. This method can change the `Message` if desired.
6. Any changes from `send` and `afterSend` methods (step 4 and step 5) commit if and only if no exceptions occurred yet. Any exceptions roll back changes to the database and set the message to retry. Be aware there is no special exception class that sets the message not to retry.
7. If this destination defines a `MessageReply` plugin, its callback handler code executes separately to handle asynchronous replies. Any changes to the `Message` entity or other entities commit to the database after the code completes, assuming the callback throws no exceptions.

If there are problems with a message, you do not necessarily need to throw an exception in all cases. For example, depending on the business logic of the application, it might be appropriate to *skip* the message and notify an administrator. If you need to resume a destination later, you can do that using the web services APIs.

All Gosu exceptions or Java exceptions during the methods `send`, `beforeSend`, or `afterSend` methods imply retryable errors. However, the distinction between retryable errors and non-retryable errors still exists if submitting errors later in the message’s life cycle. For example, you can mark non-retryable errors while acknowledging messages with web services or in asynchronous replies implemented with the `MessageReply` plugin.

Submitting Errors

If there is an error for the message, call the `reportError` method on the message. There is an optional method signature to support automatic retries. See “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 344.

Handling Duplicates

Your code must handle the possibility of receiving duplicate messages at the plugin layer or at the external system. Typically the receiving system detects duplicate messages by tracking the message ID and returns an appropriate status message. The plugin code that receives the reply messages can report the duplicate with `message.reportDuplicate()` and skip further processing.

Depending on the implementation, your code that receives the reply messages is either within the `MessageTransport.send()` method or in your `MessageReply` plugin. For more information about acknowledgements and errors, see “Submitting Acknowledgements and Errors” on page 344.

Saving the Destination ID for Logging or Errors

Each messaging plugin implementation must have a `setDestinationID` method, which allows the plugin to find out its own destination ID and store it in a private variable. The destination can use the destination ID within code such as:

- Logging code to record the destination ID.
- Exception-handling code that works differently for each destination.
- Other integrations, such as sending destination ID to external systems so that they can suspend/resume the destination if necessary.

Handling Messaging Destination Suspend, Resume, Shutdown

The standard messaging plugins have methods that perform special actions for the administrative commands for destination suspend, destination resume, and before messaging system shutdown.

Typically, suspend and resume is the result of an administrator using the BillingCenter Administration tab in the user interface to suspend or resume a messaging destination. Alternatively, suspend and resume could be the result of a web service API call to suspend or resume destinations.

Messaging shut down occurs during server shutdown or if the configuration is about to be reread. After message sending resumes again, BillingCenter reuses the same *instance* of the plugin. BillingCenter does not destroy the existing messaging plugin instance (or recreate it) as part of shutdown.

To trap these actions, implement the `suspend`, `shutdown`, and `resume` methods of your messaging plugin.

You can implement these methods just for logging and notification, if nothing else. For example, a message transport plugin’s `suspend` or `shutdown` methods could log the action and send notifications as appropriate. For example:

```
public void suspend() {  
    ...  
  
    if(_logger.isDebugEnabled()) {  
        _logger.debug("Suspending message transport plugin.")  
    }  
  
    MyEmailHelperClass.sendEmail(".....")  
}
```

During the `suspend`, `shutdown`, and `resume` methods of the plugin, the plugin must not call any BillingCenter `IMessageToolsAPI` web service APIs that suspend or resume messaging destinations. Doing so creates circular application logic, so such actions are forbidden.

It is unsupported for a messaging plugin `suspend`, `shutdown`, and `resume` method to use messaging web service APIs that suspend or resume messaging destinations.

Resynchronizing Messages for a Primary Object

BillingCenter implementations can use the messaging system to synchronize data with an external system.

In rare cases some messaging integration condition might fail, such as failure to enforce an external validation requirement properly. If this happens, an external system might process one or more BillingCenter messages incorrectly or incompletely. If the destination detects the problem, the external system returns an error. The error must be fixed or there may be synchronization errors with the external system.

However, suppose the administrator fixes the data in BillingCenter and improves any related code. It still might be that the external system has incorrect or incomplete data. BillingCenter provides a programming hook called a *resync event* (a resynchronization event) to recover from such messaging failures.

To trigger this manually, an administrator navigates to the Administration tab in BillingCenter, views any unsent messages, clicks on a row, and clicks **Resync**.

You can trigger resync from the administration user interface or programmatically using the **MessagingToolsAPI** web service. Refer to the following table for resync methods.

| To resync this entity | MessagingToolsAPI method | Description |
|-----------------------|---------------------------------|---|
| Account | <code>resyncAccount</code> | Resyncs an account |
| Producer | <code>resyncProducer</code> | Resyncs a producer |
| PolicyPeriod | <code>resyncPolicyPeriod</code> | Resyncs a policy period |
| Contact | n/a | Although Contact can be a primary entity for a BillingCenter messaging destination, BillingCenter does not support resynchronizing a contact. |
| other entity types | n/a | You can only resynchronize entity types that can be primary entities in the BillingCenter messaging system. |

As a result of a resync request, BillingCenter triggers the resync event listed in the table. Configure your messaging destination to listen for this event. Then, implement Event Fired business rules that handle that event.

Afterwards, BillingCenter marks all messages that were pending as of the resync as *skipped*. You must implement Guidewire Studio rules that examine the and generate necessary messages. You must bring the external system into sync with the current state of the primary object related to those messages.

Design your Gosu resync Event Fired rules to how your particular external systems recover from such errors.

There are two different basic approaches for generating the resync messages.

In the first approach, your Gosu rules traverse all claim data and generate messages for the entire primary object and its subobjects that might be out-of-sync with the external system.

Depending on how your external system works, it might be sufficient to overwrite the external system's claim with the BillingCenter version of this data. In this case, resend the entire series of messages. To help the external system track its synchronization state, it may be necessary to add custom extension properties to various objects with the synchronization state. If you can somehow determine that you only need to resend a subset of messages, only send that minimal amount of information. However, one of the benefits of resync is the opportunity to send all information might conceivably be out of sync. Think carefully about how much data is appropriate to send to the external system during resync.

Your Event Fired rules that handle the resync can examine the failed message and all queued and unsent messages for the claim for a specific destination. Your rules use that information to determine which messages to recreate. Instead of examining the entire claim's history you could consider only the failed and unsent messages. Because a message with an error prevents sending subsequent messages for that claim, there may be many unsent pending messages. To help with this process, BillingCenter includes properties and methods in the rules context on `messageContext` and `Message`.

From within your Event Fired rules, your Gosu code can access the `messageContext` object. It contains information to help you copy pending Message objects. To get the list of pending messages from a rule that handles the resync event, use the read-only property `messageContext.PendingMessages`. That property returns an array of pending messages. After your code runs, the application skips all these original pending messages. This means that the application permanently removes the messages from the send queue after the resync event rules complete. If there are no pending messages at resync time, this array has length zero.

WARNING If you create new messages, the new messages send in creation order independent of the order of original messages. This might be a different order than the original messages. Think carefully about how this may or not affect edge cases in the external system.

There are various properties within any message you can get in pending messages or set in new messages:

- `payload` - A string containing the text-based message body of the message.
- `user` - The user who created the message. If you create the message without cloning the old message, the user by default is the user who triggered the resync. If you create the message by cloning a pending message, the new message inherits the original user who creates the original message. In either case, you can choose to set the `user` property to override the default behavior. However, in general Guidewire recommends setting the `user` to the original user. For financial transactions, set the `user` to the user who created the transaction.

There are also read-only properties in pending messages returned from `messageContext.PendingMessages`:

- `EventName` - A string that contains the event that triggered this message. For example, “`AccountAdded`”.
- `Status` - The message status, as an enumeration. Only some values are valid during resync. The utility class `gw.pl.messaging.MessageStatus` contains static properties and static methods that you can use for easy to understand code. For a complete reference, see “`Message Status Code Reference`” on page 357.
- `ErrorDescription` - A string that contains the description of errors, if any. This may or may not be present. This is set within a negative acknowledgement (NAK).
- `SenderRefID` - A sender reference ID set by the destination to uniquely identify the message. Your destination can optionally set the `message.senderRefID` field in any of your messaging plugins during original sending of the message. Only the *first* pending message has this value set due to *safe ordering*. You only need to use the sender reference ID if it is useful for that external system. The `SenderRefID` property is read-only from resync rules. This value is `null` unless this message is the first pending message and it was already sent (or pending send) and it did not yet successfully send. As long as the `message.status` property does not indicate that it is *pending send*, the message could have the sender reference ID property populated by the destination.

Cloning New Messages From Pending Messages

As mentioned earlier, during resync you can clone a new message from a pending message that you got from `messageContext.PendingMessages`. To clone a new a new message from the old message, pass the old message as a parameter to the `createMessage` method:

```
messageContext.createMessage(message)
```

This alternative method signature (in contrast to passing a `String`) is an API to copy a message into a new message and returns the new message. If desired, modify the new message’s properties within your resync rules. All new messages (whether standard or cloned) submit together to the send queue as part of one database transaction after the resync rules complete.

The cloned message is identical to the original message, with the following exceptions:

- The new message has a different message ID.
- The new message has status of pending send (`status = PENDING_SEND`).
- The new message has cleared properties for ACK count and code (`ackCount = 0; ackCode = null`).
- The new message has cleared property for retry count (`retryCount = 0`).
- The new message has cleared property for sender reference ID (`senderRefID = null`).

- The new message has cleared property for error description (`errorDescription = null`).

BillingCenter marks all pending messages as skipped (no longer queued) after the resync rules complete. Because of this, resync rules must either send new messages that include that information, or manually clone new messages from pending messages, as discussed earlier.

IMPORTANT All pending messages skip after the Event Fired rules for the resync event complete. You must create equivalent new messages for all pending messages.

How Resync Affects Pre-Update and Validation

Be aware that pre-update and validation rule sets do not run solely because of a triggered event. A account's pre-update and validation rules run only if actual entity data changes. In cases where triggered events do not correspond to already-changed entities, the event firing alone does not trigger account pre-update and validation rules.

This does not affect most events because almost all events correspond to entity data changes. However, for the events related to resync, no entity data inherently changes due to this event.

This also can affect other custom event firing through the `addEvent` entity method.

Resync in ContactManager

If you license Guidewire ContactManager for use with BillingCenter, be aware that ContactManager supports *resync* features for the `ABContact` entity.

To detect resynchronization of an `ABContact` entity, set your destination to listen for the `ABContactResync` event.

Your Event Fired rules can detect that event firing and then resend any important messages. Your rules generate messages to external systems for this entity that synchronize ContactManager with the external system. For more information, see “ContactManager Messaging Events by Entity” on page 238 in the *Contact Management Guide*.

Message Payload Mapping Utility for Java Plugins

A messaging plugin may need to convert items in the payload of a message before sending it on to the final destination. A common reason for this is mapping typecodes. For many properties governed by typelists, the typecode might have the same meaning in both systems. However, this does not work for all situations. Instead, you might need to map codes from one system to another. For example, convert code `A1` in BillingCenter to the code `XYZ` for the external system.

If you implement your plugin in Java, you can use a utility class included in the BillingCenter Java API libraries that map the message payload using text substitution. This class scans a message payload to find any strings surrounded by delimiters that you define and then substitutes a new value. The class is `com.guidewire.util.StringSubstitution`. Refer to the *Java API Documentation* for reference.

To use the String substitution Java class from Java plugin code

1. Choose start and end delimiters for the text to replace. For example, you can use the 2-character string “**” as the start delimiter and end delimiter.
2. Put these delimiters around the Gosu template text to map and replace. For example, a Gosu template might include “`Injury=**exposure.InjuryCode**`”. This might generate text such as “`Injury=**A1**`” in the message payload.

3. Implement a class that implements the inner interface called `StringSubstitution.Exchanger`. This exchanger class might use its own look-up table or look in a properties file and substitute new values. The Exchanger interface has one method called `exchange` that translates the token. This method takes a `String` object (the token) and translates it and returns a new `String` object.
4. Instantiate your class that implements Exchanger, and then instantiate the `StringSubstitution` class with the constructor containing your delimiters and your Exchanger instance.

```
MyExchanger myExchangerInstance = new MyExchanger()
StringSubstitution mySub = new StringSubstitution("/**", "/**", myExchangerInstance)
```
5. Call the `substitute` method on the `StringSubstitution` instance to convert the message payload string.

Message Status Code Reference

The following table describes the message status values and the contexts they can appear for `Message` and `MessageHistory` objects. The second column indicates the static property you can use on the `gw.pl.messaging.MessageStatus` class to make your code easier to understand.

The status code information is particularly valuable for Gosu code that implements resynchronization (*resync*). See “Resynchronizing Messages for a Primary Object” on page 353.

| MessageStatus static property | Message status value | Meaning | Valid in Message | Valid in MessageHistory | Can appear during resync |
|----------------------------------|----------------------------|--|---------------------|----------------------------|--------------------------------|
| PENDING_SEND | 1 | Pending send. this is the initial state for messages. | ● | | ● |
| PENDING_ACK | 2 | Messages set to this state once the <code>MessageTransport.send()</code> method completes but the transport has not acknowledged the message. The status remains in this state until acknowledged, an error is received, or it is retried This state is sometimes referred to as <i>in flight</i> . | ● | | ● |
| ERROR | 3 | <i>Legacy non-retryable error. Provided for legacy use and is no longer used.</i> | | | |
| RETRYABLE_ERROR | 4 | Message has been acknowledged with a retryable error | ● | | ● |
| ACKED | 10 | Message has been successfully acknowledged | | ● | |
| ERROR_CLEARED | 11 | The message was in RETRYABLE_ERROR state but the administrator skipped this message. | | ● | |
| ERROR_RETRYED | 12 | Error retried. This is the original message as represented as a <code>MessageHistory</code> object. Another message in the Message table represents the clone of this message. | | ● | |
| SKIPPED | 13 | The administrator skipped this message. | ● | | |

Some static properties on the `MessageStatus` class contains properties that contain arrays of message status values. You can use these to check values with easy to read code.

The class also has static methods that take a status (state) value and return true if the status is in a list of relevant values. For example, to test if a message was ever acknowledged (including error values), type the Gosu code:

```
gw.pl.messaging.MessageStatus.isAcked(Message.Status)
```

The following table lists additional `MessageStatus` static properties and the static methods.

| MessageStatus static property or static method | Description |
|---|--|
| Properties | |
| ALL_STATES | An array of all states. |
| ACKED_STATES | An array of acknowledged states, including error states. |
| ACTIVE_STATES | An array of all active states. |
| BLOCKING_STATES | An array of active states for messages blocking sends of subsequent messages. |
| ERROR_STATES | An array of error states. |
| INACTIVE_STATES | An array of final message states for messages that no longer require processing. |
| PENDING_STATES | An array of all pending states. |
| RETRYABLE_STATES | An array of retryable states, PENDING_ACK or RETRYABLE_ERROR. |
| Methods | |
| isActive(state) | Returns true if the state is in the array ACTIVE_STATES. |
| isInFlight(state) | Returns true if the status indicates a message in flight (PENDING_ACK). |
| isRetryableError(state) | Returns true if the status is RETRYABLE_ERROR. |
| isRetryable(state) | Returns true if the status is in the array RETRYABLE_STATES |
| isPending(state) | Returns true if the status is in the array PENDING_STATES. |
| isPendingSend(state) | Returns true if the status is PENDING_SEND. |
| isAcked(state) | Returns true if the status is in the array ACKED_STATES. |
| isError(state) | Returns true if the status is in the array ERROR_STATES. |

Monitoring Messages and Handling Errors

BillingCenter provides tools for handling errors that occur with sending messages:

- Automatic retries of sending errors
- Ability for the destination to request retrying or skipping messages in error
- User interface screens for viewing and taking action on errors
- Web service APIs for taking action on errors
- A command line tool for taking action on errors

Error Handling Concepts

Before describing the tools, it is useful to think about the kinds of errors that can occur and the actions that can be taken on these errors:

- **Pending send** – The message has not been sent yet:
 - The destination may be *suspended*, which means it is not processing messages

- The destination may be simply not fast enough to keep up with how quickly the application generates messages. BillingCenter can generate messages very quickly. For more information about how BillingCenter retrieves messages from the send queue, see “Message Ordering and Multi-Threaded Sending” on page 337.
- **Errors during the Send method** – BillingCenter attempted to send the message but the destination threw an exception. If the exception was *retryable*, BillingCenter automatically attempts to send the message again some number of times before turning it into a failure. If it is a *failure*, BillingCenter suspends the destination automatically until an administrator restarts it. Failures include a retryable send error reaching its retry limit, or unexpected exceptions during the send method.

The destination also resumes (un-suspends) if the administrator removes the message or resynchronizes (resyncs) the message’s primary object.

- **In-flight** – BillingCenter waits for an acknowledgement for message it sent. If errors occur such that the external system does not receive or properly acknowledge the message, BillingCenter waits indefinitely. If the message has a related primary object for that destination, it is *safe-ordered*. This type of error blocks sending other messages for that primary object for that destination. For more information, see “Safe Ordering” on page 306.

In this case, you can intervene to process the message (*skip* the unfinished message) or retry the message. Be very careful about issuing retry or skip instructions:

- A *retry* could cause the destination to receive a message twice.
- A *skip* could cause the destination to never get the intended information.

In general, you must understand the actual status of the destination to make an informed decision about which correction to make.

- **Error** – The destination indicates that the message did not process successfully. Again, the error blocks sending subsequent messages. In some cases, the error message indicates that the error condition may be temporary and the error is retryable. In other cases, the message indicates that the message itself is in error (for example, bad data) and resending does not work. In either case, BillingCenter does not automatically try to send again.
- **Positively Acknowledged (ACK)** – The message successfully processed. These messages stay in the system until an administrator purges them. However, since the number of messages is likely to be very large, Guidewire recommends that you purge completed messages on a periodic basis.
- **Negatively Acknowledged (NAK)** – Message sending for that message failed at the external system (not a network error) either because the message had an error or was a duplicate.

If BillingCenter retries a failed message, it marks the original message as failed/retried and creates a new copy of that message (with a new message ID) to send. The assumption behind this behavior is that a destination tracks messages received and does not accept duplicate messages. To do this, the retry must have a new message ID. If BillingCenter retries an in-flight message because it never got an ACK, then it resends the original message with the same ID. If the destination never got the message, then there is no problem with duplicate message IDs. If the destination received the message but BillingCenter never got an acknowledgement, then this prevents processing the message twice. The destination can send back an error (mark it as a duplicate) or send back another acknowledgement.

If BillingCenter receives an error, it holds up subsequent messages until the error clears. If the destination sends back duplicate errors, you can filter out duplicates and warn the administrator about them. However, you can choose to simply issue a positive acknowledgement back to BillingCenter.

BillingCenter could become sufficiently out of sync with an external system such that simply skipping or retrying an individual message is insufficient to get both systems in sync. In such cases, you may need special administrative intervention and problem solving. Review your sever logs to determine the root cause of the problem.

For more information about acknowledgements, see “Reporting Acknowledgements and Errors” on page 343.

The Administration User Interface

BillingCenter provides a simple user interface to view the event messaging status for accounts. This helps administrators understand what is happening, and might give some insight to integration problems and the source of differences between BillingCenter and a downstream system.

For example, if you add one account in BillingCenter but it does not appear in the external system, you need to know the following information:

- As far as BillingCenter knows, have all messages been processed? (“Green light”) If the systems are out of sync, then there is a problem in the integration logic, not an error in any specific message.
- Are messages pending, so you simply need to wait for the update to occur. (“Yellow light”)
- Is there an error that needs to be corrected? (“Red light”) If it is a retryable error, you can request a retry. This might make sense if the external system caused the temporary error. For example, perhaps a user in the external system temporarily locked the account by viewing it on that system’s screen. In many cases, you can simply note the error and report it to an administrator.

Administrators have extensive errors across the system. In the Administration section of BillingCenter, you can select the **Event Messages** console from the Administration page. There are three levels of detail for viewing events and messaging status:

- **Destinations List** – This shows a list of destinations, its sending status (for example, started or suspended), and counts of failed or in-process messages. At this level an administrator can only suspend or resume the entire destination. If suspended, BillingCenter just holds all newly generated messages until the destination is resumes.
- **Destination Status** – This provides a list of accounts that have failed messages or messages in-process for a single destination. Different filters can help you find different kinds of problems. You can also search for a particular account. This would then open the detail view for that policy.

From the **Administration** tab, you can force BillingCenter to stop and restart the messaging sending queue without restarting the entire server. In rare situations, this is useful if you suspect that the sending queue became out of sync with messages waiting in the database. For example, perhaps your a network interruption in the cluster might cause such an issue. If the administrator restarts the messaging system, BillingCenter shuts down each destination, then restarts the queue process, then reinitializes each destination.

Messaging tool actions such as `suspend`, `resume`, `retry`, and others can trigger from the Administration page and also from the `MessagingToolsAPI` web service. These messaging tools require the server’s run level to be `multiuser`.

Web Services for Handling Messaging Errors

In addition to monitoring and responding to errors with the administration user interface, BillingCenter provides some other interfaces for dealing with errors. This section describes these tools.

First, BillingCenter provides a web service called `MessagingToolsAPI`, which lets an external system remotely control the messaging system. See “[Messaging Tools Web Service](#)” on page 360.

Messaging Tools Web Service

BillingCenter provides a web service called `MessagingToolsAPI`, which lets an external system remotely control the messaging system.

These API methods (except for `getAccountMessageStatistics`) are available using the `messaging_tools` command line tool. See “[Messaging Tools Command](#)” on page 189 in the *System Administration Guide*. Within the administrative environment, the following command shows the syntax of each command:

```
messaging_tools -help
```

Retry a Message

To retry a message, call the `retryMessage` method. The behavior is the same as in the BillingCenter user interface.

Skip a Message

To skip a message, call the `skipMessage` method. The behavior is the same as in the BillingCenter user interface.

Retry Messages

To retry all *retryable* messages for a destination, call the `retryRetryableErrorMessages` method.

For example, call this method if the destination was temporarily unavailable and is now back on-line.

You can specify a maximum number of times to retry each message. This maximum prevents messages from retrying forever.

Purge Completed Messages

To purge completed messages, call the `purgeCompletedMessages` method. This method deletes completed messages in the messaging history table (`MessageHistory`) from the BillingCenter database for all messages older than the given date.

Since the number and size of messages may be very large, Guidewire recommends you use this method periodically to purge old messages. Purging messages in the message history table prevents the database from growing unnecessarily large.

Always purge completed (inactive) messages before upgrading to a new version of BillingCenter. Purging completed messages reduces the complexity of your upgrade.

Additionally, periodically use this command to purge old messages to avoid the database from growing unnecessarily.

Suspend Destination

To suspend a destination, call the `suspendDestination` method. Suspending a destination means that BillingCenter stops sending messages to a destination.

BillingCenter suspends the destination so that it can also release any resources such as a message batch file. Use this method to shut down the destination system to halt sending during processing of a daily batch file.

Resume Destination

To resume a destination, call the `resumeDestination` method. Resuming a destination means that BillingCenter starts trying to send messages to the destination again.

If a previous suspend action released any resources, resuming the destination reclaims those resources. For example, the destination might reconnect to a message queue.

Get Messaging Statistics for a Safe Ordered Object

To get messaging statistics for a safe-ordered object, call the `getMessageStatisticsForSafeOrderedObject` object.

This method returns information that is similar to the user interface for an account or policy:

- the number of messages failed
- retryable messages
- in-flight messages
- unsent messages

Messaging tool actions such as `suspend`, `resume`, `retry`, and others can be triggered from the Administrator interface and also from the web services `MessagingToolsAPI` interface. These messaging tools can only be used if the server run mode is `multiuser`.

Change Messaging Destination Configuration Parameters

To change messaging destination configuration parameters on a running server, call the `configureDestination` method. This restarts the destination with the change to the configuration settings. The command waits for the destination to stop for the configured stop time. The method returns nothing,

The arguments are:

- `destID` – The destination ID of the destination to suspend
- `maxretries` – maximum retries
- `initialretryinterval` – initial retry interval
- `retrybackoffmultiplier` – additional retry backoff
- `pollinterval` – how often to poll, from start to start
- `numsenderthreads` – number of sender threads for multi-threaded sends
- `chunksize` – number of messages to read in a chunk
- `timeToWaitInSec` – the number of seconds to wait for the shutdown before forcing it

Get Configuration Information from a Destination

To get some configuration information from a messaging destination, call the `getConfiguration` method. This information is read from files on disk during server startup, however can be modified by web services and command line tools. See earlier in this topic for the `configureDestination` method.

The `getConfiguration` method takes only one argument, the destination ID. The method returns a `ExternalDestinationConfig` object, which contains properties matching the properties in the `getConfiguration` method, such as the polling interval and the chunk size. See the `getConfiguration` method documentation for details of each property.

Batch Mode Integration

Most integrations using the messaging system are real-time integrations between BillingCenter and one or more destination systems. However, an external system might only be able to support batch updates. For example, some external server might need regular data from BillingCenter and retrieves it only at night because it is resource intensive. In this case, BillingCenter generates system events in real-time but sends updates in one batch to the external system.

There are essentially two approaches to handle batch messaging:

- **Approach 1: suspend a destination, then resume it later** – Using the SOAP API or command line tools, you can suspend sending messages to a destination during most of the day. If it is time to generate the batch file, you can resume (un-suspend) the destination so that BillingCenter drains its queue of messages to generate the batch file. If there are no more messages (or after some period of time), you can suspend the destination again while you process the batch file or until a pre-defined time. This is a very safe method because it uses the messaging transaction and acknowledgement model to track each message.
- **Approach 2: append messages to a batch file and send all later** – BillingCenter sends messages to a destination, which appends the messages to a batch file and immediately acknowledges the message. Periodically, the batch file is sent to the destination and processed. For example, after a certain number of messages are in the queue, then the message transport plugin can send them. Or, a separate process on the server can send these batch files. This approach allows you to send more than one message in a single remote call to the external system.

With both approaches, the message acknowledgement would come from the message transport plugin (or the message reply plugin) immediately, and not from the external system. With both approaches, this means that the messaging system built-in retry logic and ordered message sending cannot be used to deal with errors as gracefully as with a real-time integration. If any errors are found after sending an acknowledgement to BillingCenter, your integration code must deal with these issues outside of BillingCenter.

Included Messaging Transports

The Built-in Email Transport

BillingCenter includes a built-in transport that can send standard SMTP emails.

By default, the `emailMessageTransport` plugin uses the *system user* to retrieve a document from the external system. You can choose to retrieve the document on behalf of the user who generated the email message. To do this, set the `useMessageCreatorAsUser` property in the `emailMessageTransport` plugin. In Studio, navigate to **Configuration** → **config** → **Plugins** → **registry** → `emailMessageTransport`. In the parameters area of the pane, click the **Add** button. Add the parameter `useMessageCreatorAsUser` and set it to `true`.

Enabling the Built-in Console Transport

BillingCenter includes a built-in console message transport example, which is an extremely simple messaging transport that writes the message text payload to the BillingCenter console window. Enable this transport in the plugin registry in Studio to debug integration code that creates and sends messages. For more detail of how to do this, see the Studio Guide.

If you register a messaging plugin, you must register it in two places. First, register it in the plugin registry in the plugin editor; see “Using the Plugins Registry Editor” on page 109 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 131 in the *Configuration Guide*.

In the plugin editor, use the plugin name `consoleTransport`, the interface name `MessageTransport`, the Java class `examples.plugins.messaging.ConsoleMessageTransport` and the plugin directory `messaging`.

In the messaging editor, use the following settings:

- set the plugin name to "Console Message Logger"
- set transport plugin name to `consoleTransport`
- set initial retry interval set to 100
- set max retries set to 3
- set `retrybackoffmultiplier` to 2
- set event name to "\w*", which means the destination wants notification of all events

This tells BillingCenter to send all events to this destination, and trigger Event Fired rules accordingly. Write Event Fired rules that create messages for this destination

After redeploying the server, watch the console window for messages.

Importing Billing and Account Data

Importing from Database Staging Tables

BillingCenter supports high-volume bulk data import by using database staging tables. Staging tables typically are used for large-scale data conversions, such as migrating accounts from a legacy system to BillingCenter.

This topic includes:

- “Introduction to Database Staging Table Import” on page 367
- “Overview of a Typical Database Staging Table Import” on page 371
- “Database Import Performance and Statistics” on page 377
- “Table Import Tools” on page 377
- “Populating the Staging Tables” on page 379
- “Loading BillingCenter-specific Entities” on page 382
- “Data Integrity Checks” on page 389
- “Table Import Tips and Troubleshooting” on page 390
- “Staging Table Import of Encrypted Properties” on page 391

Introduction to Database Staging Table Import

Database staging table import provides significantly higher performance than importing individual records with entity-specific web services APIs. Staging table import avoids intermediate data formats such as XML and avoids the need to parse and transform data into internal Java objects.

Database staging table import further improves performance by using bulk SQL Insert and Select statements that operate on entire tables at one time. Staging table import does not operate on single rows individually to import data, unlike entity-specific web services APIs that insert one row at a time.

This topic includes:

- “Staging Tables” on page 368

- “Zone Data” on page 368
- “Your Conversion Tool” on page 369
- “Integrity Checks” on page 369
- “Logical Units of Work, LUW, and LUWIDs” on page 370
- “Load Error Tables” on page 370
- “Exclusion Table” on page 370
- “Load History Tables” on page 370
- “Load Commands and Loadable Entities” on page 370

Staging Tables

Staging tables are database tables that replicate almost completely the columns of corresponding operational tables. You prepare legacy or other data for bulk import directly into BillingCenter operational tables by first loading the data into staging tables. BillingCenter automatically creates a staging table for any entity defined as `loadable` with at least one property also defined as `loadable`.

Staging Table Names

In the default configuration of BillingCenter, most operational tables that have names with a `bc_` prefix have corresponding staging tables named with a `bcst_` prefix. Loadable data model extensions, which have operational tables named with a `bcx_` prefix, have corresponding staging tables named with a `bcst_` prefix.

Differences Between Staging Tables and Operational Tables

There are important differences between the columns of staging tables and operational tables. For example, staging tables have extra columns track to track the status of data imported into operational tables. Staging tables lack columns in their corresponding operational tables with certain non-essential business data, such as `CreateUserID` and `CreateTime`. However columns with essential business data are present in staging tables and their corresponding operational tables.

See also

- For more information on the differences between staging tables and operational tables, see “Populating the Staging Tables” on page 379

Zone Data

BillingCenter uses zone data for the following features:

- Assignment by location
- Address auto-fill
- Setting regional holidays

Steps to Import Zone Data

The process to import zone data comprises these main steps:

1. **Get or create your zone data files** – Create zone data files in comma separated value (CSV) format that contain columns for the postal code, state, city, and county of specific geographic zones. Each line in the file must use the following format:

```
postalcode,state,city,county
```

For example,

```
94114,CA,San Francisco,San Francisco
```

For more information, see “Zone Data Files Supplied by Guidewire” on page 369.

2. Run the zone import command to add zone data to the zone staging table – Run the command line tool `zone_import` in the directory `BillingCenter/admin/bin`. You can also use the web service `ZoneImportAPI` to add your zone data to the staging table.

For example with the command line tool, your command might look like this:

```
zone_import -clearstaging -import myzonedata.csv -server http://myserver:8080/pc -user myusername
```

For more information, see “Zone Import Command” on page 197 in the *System Administration Guide*.

3. Run the table import command to bulk load the zone operational table – Run the command line tool `table_import` in the directory `BillingCenter/admin/bin`. You can also use the web service `TableImportAPI` to bulk load the zone operational table.

For more information, see “Table Import Command” on page 195 in the *System Administration Guide*.

Zone Data Files Supplied by Guidewire

BillingCenter provides a collection of zone data files for various localities with small sets of zone data that you can load for development and testing purposes. The zone data files are in the following location in the Studio Project window:

```
configuration → config → geodata
```

Within the `geodata` folder, BillingCenter stores the zone information in country-specific `zone-config.xml` files, with each file in its own specific country folder. For example, the `zone-config.xml` file that configures address-related information in Australia is in the following location in the Studio Project window:

```
configuration → config → geodata → AU
```

Guidewire provides the `US-Locations.txt` and similar files for testing purposes to support autofill and autocomplete when users enter addresses. This data is provided on an as-is basis regarding data content. For example, the provided zone data files are not complete and may not include recent changes.

Also, the formatting of individual data items in these files might not conform to your internal standards or the standards of third-party vendors that you use. For example, the names of streets and cities are formatted with mixed case letters but your standards may require all upper case letters.

The `US-Locations.txt` file contains information that does not conform to United States Postal Service (USPS) standards for bulk mailings. You can edit the `US-Locations.txt` file to conform to your particular address standards, and then import that version of the file.

Your Conversion Tool

A critical component of any migration process is a custom conversion tool that you write. Your conversion tool converts your legacy data into BillingCenter format and inserts the converted data into the staging tables. The conversion tool must map your legacy data format, which might be a flat file format, into a format almost identical to BillingCenter operational tables. If your legacy format is dissimilar to the BillingCenter format, which is most often the case, this tool must support complex internal logic.

Integrity Checks

Before loading staging table data into operational database tables, BillingCenter runs many BillingCenter-specific data integrity checks. These checks find and report problems that would cause import to fail or might put BillingCenter into an inconsistent state. Integrity checks are a large set of auto-generated database queries (SQL queries) built into the application.

You can check if any integrity checks failed at `Server Tools → Info Pages → Load Errors`. For more information, see “Load Errors” on page 170 in the *System Administration Guide*.

Logical Units of Work, LUW, and LUWIDs

The data related to a account is spread out across many tables and potentially many rows. You must identify self-contained units of data that must load together or fail together if something is wrong within that account.

BillingCenter uses the generic term *logical unit of work* (LUW) to refer to all rows across all tables as a single unit for integrity checks and loading. For example, if a account fails an integrity check, all associated records such as related Address records in that logical unit of work fail the integrity check with the account.

Each staging table row has a *logical unit of work ID* property, called LUWID, which identifies the LUW grouping of this data. This topic sometimes refers to this logical unit of work ID as an LUWID. After your conversion tool populates the staging tables, your conversion tool must set the LUWID property to something useful for each row. For example, a pre-defined legacy account number/ID.

An LUWID is used in the following ways:

- LUWIDs help identify which data failed. See “Load Error Tables” on page 370.
- LUWIDs identify which data to exclude from future integrity checks or load requests. See “Exclusion Table” on page 370.

Load Error Tables

The *load error tables* hold data from failed data integrity checks. Do not directly read or write these tables. Instead, examine them using the **Load Errors** interface. You can view these errors in BillingCenter at **Server Tools** → **Info Pages** → **Load Errors**. For more information about using this screen, see “Load Errors” on page 170 in the *System Administration Guide*.

Most errors relate to a particular staging table row, so the **Load Errors** page shows the following information:

- Table
- Row number
- Logical unit of work ID
- Error message
- Data integrity check, also called the query, that failed.

In some cases, BillingCenter cannot identify or store a single LUWID for the error.

Exclusion Table

The *load exclusion table* is a table of logical unit of work IDs to exclude from staging table processing during the next integrity check or load request. Do not directly read from or write to these tables. Instead, use web services or command line tools to populate exclusion tables from logical units of work IDs (LUWIDs) in the load error tables.

Load History Tables

Load history tables store results for import processes, including rows for each integrity check, each step of the integrity check, and row counts for the expected results. Use these to verify that the table-based import tools loaded the correct amount of data. You can view this information in BillingCenter at **Server Tools** → **Info Pages** → **Load History**. For more information about using this screen, see “Load History” on page 170 in the *System Administration Guide*.

Load Commands and Loadable Entities

BillingCenter creates staging tables during the upgrade process when the server starts. If a BillingCenter table is loadable and has at least one loadable property, BillingCenter creates a corresponding staging table for it.

During staging table import, all loadable entities are copied from the staging tables to the operational tables. After an entity imports successfully, the application sets each entity's load command ID property (`LoadCommandID`) to correspond to the staging table conversion run that brought the row into BillingCenter.

An entity's `LoadCommandID` property is always `null` for rows that were created in some other way, in other words new entities that did not enter BillingCenter through staging table import.

The `LoadCommandID` property persists even after performing additional import jobs. The presence of the `LoadCommandID` property does not guarantee that the current data is unchanged since the row was imported. If the user, application logic, or integration APIs change the data, the `LoadCommandID` property stays the same as when the row was first imported.

You can use this feature to test whether an entity was loaded using database staging tables or some other way. From your business rules or from a Java plugin, test an entity's `LoadCommandID`. From Gosu, check `entity.LoadCommandID`. From Java, check the `entity.getLoadCommandID` method. If the load command is non-`null`, the entity was imported through the staging table import system. All entities with that same load command ID loaded together in one import request. If the load command is `null`, the entity was created in some way other than database table import.

These load command IDs correspond to results of programmatic load requests to import staging tables.

For example, use the command line `table_import` tool in the `BillingCenter/admin/bin` directory. The tool returns the `LoadCommandID`. Alternatively, you can call the web service to load database tables:

```
result = ITableImport.integrityCheckStagingTableContentsAndLoadSourceTables(...);
```

The result of that method is a `TableImportResult` entity instance, which contains a `LoadCommandID` property, which is the load command ID. Call `result.getLoadCommandID()` to get the load command ID for that load request. Save that value and test specific entities to see how they were loaded. Compare that value against that saved `LoadCommandID`. Similarly, if you used the command line tools to trigger database table import, those tools return the `LoadCommandID`.

You can also track load import history using the database load history tool user interface at [Server Tools → Info Pages → Load History](#). For more information about using this screen, see “Load History” on page 170 in the *System Administration Guide*.

Remember that even if an object has a load command ID that matches a known load request, its data may have changed after loading due to user actions or APIs.

Overview of a Typical Database Staging Table Import

As an example of the migration process that uses database staging tables and the database import tools, consider migrating accounts from your legacy system to BillingCenter. As a prerequisite to migrating legacy accounts, you must develop a conversion tool that selects accounts from your legacy system and transforms their data to BillingCenter format. Then, your tool must insert the converted data into the staging tables.

Server Modes and Run Levels for Database Staging Table Import

To import data by using database staging tables and the database import tools, the server must be set to the maintenance run level. The maintenance run level prevents new user connections, halts existing user sessions, and releases all database resources. The server prohibits access from the PolicyCenter user interface to the database whenever the server runs at the maintenance run level.

The database import tools do not require the server to run in a specific mode. You can perform database staging table imports regardless of production, test, or development mode.

Plan when to perform a database import carefully on production systems. Users are blocked from using the application during that time. A database import operation can take considerable time, depending on how much data you want to import. To mitigate the amount of down time for users, consider importing large amounts of data in phases rather than in a single operation.

Importing Zone Data

You can import zone data by using database staging tables and the database import command. However, the sources for zone data are CSV files for each country or region that you want to support. Guidewire provides the `zone_import` tool to load zone data from CSV files into the staging table for zone data. As a prerequisite to importing zone data, you must modify `zone-config.xml` files to configure which fields of a zone to data import.

See also

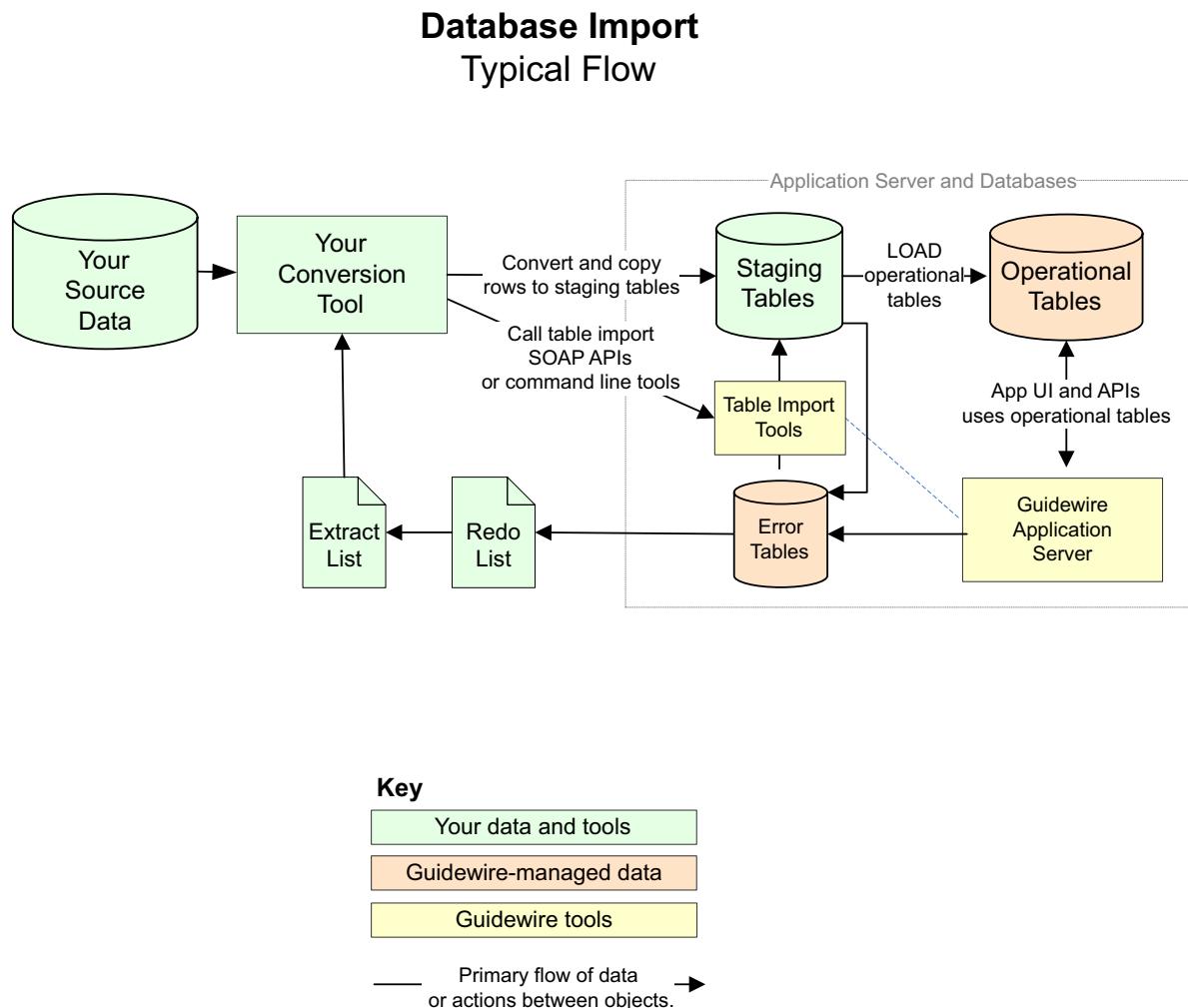
- “Zone Import Command” on page 197 in the *System Administration Guide*

High-Level Steps in a Typical Database Staging Table Import

The following procedure lists the high-level steps involved in a typical database import procedure to migrate account data from your legacy system to BillingCenter.

1. Run your conversion tool to select and convert legacy accounts to BillingCenter format and insert the covered data into staging tables.
2. If the server mode is production, set the server to the maintenance run level.
3. Run integrity checks on staging table data using the `table_import` command or invoke the `TableImportAPI` web service.
4. Set the server to the multiuser run level.
5. View load errors on the **Load Errors** page.
6. Fix errors and run integrity checks again, repeating some or all of the steps listed previously in this list.
7. After you are certain all integrity checks succeeded, set the server to the maintenance run level.
8. Load the data from the staging tables into operational tables by using web services or command line table import tools.
9. Set the server to the multiuser run level.

The following diagram shows the major steps in a typical database staging table import:



Detailed Steps in a Typical Database Staging Table Import

Migrating data from a source or legacy system to BillingCenter typically happens in the following steps:

1. **If you changed the data model, run the server to perform upgrades** – The staging table import tools require that legacy data be converted to the same format as the server data.

IMPORTANT If you add encryption or change encryption settings with data in operational tables, perform the upgrade successfully before running staging table import. For more information, see “Encryption Features for Staging Tables” on page 259.

2. **Set the server to the maintenance run level** – Set each BillingCenter server to the maintenance run level by using the `system_tools` command line tool or the web service `SystemToolsAPI.setRunLevel` method.
3. **Back up the operational tables in the database** – Back up operational tables before import.

WARNING As with any major database operation, back up all operational tables before importing.

4. Clear the staging tables, the error tables, and the exclusion tables – Your conversion tool would typically do this manually, but you can also do this using web service APIs described in “Table Import Tools” on page 377.

5. Run BillingCenter database consistency checks – Database consistency checks verify that the data in your operational tables does not contain any data integrity errors. Run database consistency checks using the web services API `SystemToolsAPI` method `checkDatabaseConsistency`, or using the command line:

```
system_tools -password password -checkdbconsistency
```

See “System Tools Command” on page 191 in the *System Administration Guide* for more information about the command line tool.

If you do not run consistency checks regularly, run these a long time before converting your data. For example, plan several weeks to correct any errors you may encounter.

6. Determine which records to migrate – Your conversion tool determines which legacy records to migrate according to your own criteria. For example as a first step, the tool might generate an extraction list of account numbers for accounts to convert.

7. Convert and insert legacy data into staging tables – Your conversion tool reads the accounts to convert from the source system converts their data to BillingCenter format. Then, your tool inserts the converted data for each account into the staging tables, along with associated subobjects in related tables. This step represents the bulk of your effort. Once completed, populate BillingCenter staging tables with account data ready to load into BillingCenter operational tables. For more information about populating staging tables, see “Populating the Staging Tables” on page 379.

8. Request integrity checks from the table import tools – These tools are available as a web service or from the command line. An integrity check ensures the data meets BillingCenter basic data integrity requirements. If problems are found, those records are saved in an error table. BillingCenter runs all integrity checks and reports all errors before stopping.

The command line tool is as follows:

```
table_import -integritycheck
```

Your conversion tool might automatically trigger the integrity check using the web service or command line tools after converting and loading the data into staging tables. At the time you request an integrity check, you can optionally clear error tables and exclusion tables using optional parameters. Also, you can choose to perform the integrity check as part of a load request, and if so the load proceeds only if no integrity check errors occur.

There is an optional command line flag to allow references to existing non-admin rows in the database: `-allowReferencesAllowed` (corresponding to the web service boolean parameter `allowRefsToExistingNonAdminRows`). Only use this flag (or for the web service, set it to `true`) if absolutely necessary. For example, in the rare case a policy period overlaps between the existing operational data and the data you are loading. This option flag can cause performance degradation during the check and load process.

9. Set the server to the multiuser run level – Set each BillingCenter server to the multiuser run level by using the `system_tools` command line tool or the web service `SystemToolsAPI.setRunLevel` method.

10. Check load errors from the Load History page – This page is available at `Server Tools`→`Info Pages`→`Load History`. For more information about using this page, see “Load History” on page 170 in the *System Administration Guide*.

11. Fix the errors and/or exclude records – If there are errors, the conversion tool must correct the errors or remove the bad data and try again. Use the data viewable in the user interface to generate a redo list, which is a list of records to fix and rerun. Or, use this data to find records to skip by adding to the exclusion table. If you want to exclude records with errors and you know the corresponding LUWIDs, add the LUWIDs to the exclusion table using web service APIs and command line tools. See “Table Import Tools” on page 377. Some errors are not associated with specific LUWIDs, in which case nothing can be moved to the exclusion table.

12. **Set the server to the maintenance run level** – Set each BillingCenter server to the maintenance run level by using the `system_tools` command line tool or the web service `SystemToolsAPI.setRunLevel` method.
13. **Repeat integrity checks until they succeed** – Repeat integrity checks until there are no errors.
14. **Optionally, remove excluded records from staging tables** – Remove all rows from all staging tables for records whose LUWIDs are listed in the exclusion table. To do this, use the `deleteExcludedRowsFromStagingTables` tool described in “Table Import Tools” on page 377.
15. **Handle encrypted properties** – If you have any entities with properties that support property-level encryption, your staging table data must have encrypted properties before importing them into BillingCenter. For more information, see “Encryption Features for Staging Tables” on page 259.
16. **Load staging tables into operational tables** – Eventually, integrity checks succeed for all records except for records in the exclusion table. At this point, you can load data from the staging tables into the operational tables used by BillingCenter. Do this using one of the various web service (SOAP) APIs or command line tools for loading. See “Table Import Tools” on page 377. For example, use the table import web service method `integrityCheckStagingTableContentsAndLoadSourceTables`. As BillingCenter inserts rows into the operational tables, it also inserts results into the load history table.

There is an optional command line flag to allow references to existing non-admin rows in the database: `-allreferencesallowed` (corresponding to the web service boolean parameter `allowRefsToExistingNonAdminRows`). Only use this flag (or for the web service, set it to true) if absolutely necessary. For example, in the rare case a policy period overlaps between the existing operational data and the data you are loading. This option flag can cause performance degradation during the check and load process.

If you use Oracle databases, Guidewire recommends you use the Boolean parameter `updateDBStatisticsWithEstimate` set to `true`. This indicates to update database statistics on any table loading with estimated changes based on contents of the associated staging table. This corresponds to the optional `-estimateorastats` option for the command line tool. If you load large amounts of data, the source table grows significantly. The optimizer could choose a bad query plan based on the existing state of the database statistics. Avoid this situation by updating the database statistics to reflect the expected size of the table after the load completes.

In other words, for non-Oracle databases, load staging tables with the command:

```
table_import -integritycheckandload
```

For Oracle databases, load staging tables with the command:

```
table_import -estimateorastats -integritycheckandload
```

WARNING For Oracle databases, failing to add the `-estimateorastats` option (or the equivalent web service parameter set to `true`) extremely reduces database performance. Remember to always add this additional option for Oracle databases.

The server automatically removes all data from staging tables at completion. If there were errors, data remains in the staging tables.

17. **BillingCenter populates user and time properties, as well as other internal or calculated values** – During import, BillingCenter sets the value of the `CreateUserID` and `UpdateUserID` properties to the user ID of the user that authenticated the web service or command line tool. To detect converted records in queries, create a BillingCenter user called `Conversion` (or something like that) to detect these records. Creating a separate user for conversion helps diagnose potential problems later on. This user must have the SOAP Administration permission to execute the web service or command line tool. Do not give this user additional privileges or access to the user interface or other portions of BillingCenter. At this step, BillingCenter sets the `CreateTime` and `UpdateTime` properties to the start time of the server transaction. All rows now have the same time stamp for a single import run.
18. **Set the server to the multiuser run level** – Set each BillingCenter server to the multiuser run level by using the `system_tools` command line tool or the web service `SystemToolsAPI.setRunLevel` method.

19. **Review the load history** – Review the load history at **Server Tools** → **Info Pages** → **Load Errors**. Ensure that the amount of data loaded is correct. For more information, see “Load Errors” on page 170 in the *System Administration Guide*.
20. **Update database statistics** – Particularly after a large conversion, update database statistics. See “Configuring Database Statistics” on page 39 in the *System Administration Guide*.
21. **Update database consistency checks again** – Assuming there were no consistency errors before importing, new consistency errors after imports indicate something was wrong in the staging table data uncaught by integrity checks. See step 5 for the data consistency check options.
22. **Convert more records if necessary** – If you have more data to convert, begin again at step 5.
23. **Run batch processing** – Run the following types of batch processing in the order listed:
 - a. Database Consistency Check
 - b. Charge ProRata Transaction
 - c. Write-off Staging
 - d. Invoice
 - e. Legacy Collateral
 - f. New Payment
 - g. Legacy Delinquency
 - h. Legacy Agency Bill
 - i. Statement Billed
 - j. Invoice Due
 - k. Statement Due
 - l. Database Statistics, a second time
 - m. Policy Closure
 - n. Commissions Payable Calculations
 - o. Producer Payment

In general, never interact with the BillingCenter database directly. Instead, use BillingCenter APIs to abstract access to entity data, including all data model changes. Using APIs removes the need to understand many details of the application logic governing data integrity. However, the staging tables are exceptional because conversion tools that you write can read/write staging table data before import.

WARNING You can directly manipulate the staging tables. Do not directly read or write the load error tables or the exclusion tables. The only supported access to these tables is the Server Tools user interface. See “Using BillingCenter Server Tools” on page 155 in the *System Administration Guide*.

BillingCenter attempts to perform the integrity check and then load all data in the staging tables, so each import attempt must start with a clean set of tables. Therefore remove the staging data after successfully loading the data. This is also why accounts that cannot pass the integrity check must be fixed or removed before the import proceeds.

As part of doing a conversion from a source system into BillingCenter, there are several ways in which errors can be handled. During the development of the conversion tool, errors frequently occur due to incorrect mapping data from the source system into the staging tables. Errors also occur if you do not properly populating all necessary properties. Typically, fix these errors by adjusting algorithms in your conversion tool. Typically, you run many trial conversions and integrity checks with iterative algorithm changes before finally handling all issues.

If the server flags an error that is simply a problem with the source data, then correct it directly in the staging tables using direct SQL commands. In contrast, never modify operational tables with direct SQL commands.

Alternatively, you may want to correct errors in the source system. In that case, remove records that fail the integrity check from the staging tables. Correct the errors in the source system. Then, rerun the entire conversion process.

Database Import Performance and Statistics

Guidewire strongly encourages you to design database import code in such a way as to isolate your code performance from BillingCenter database import API performance:

- If using Oracle, take statistics snapshots (statspack snapshots at level 10 or AWR snapshots) at the beginning and end of `integritycheck` commands and `integritycheckandload` commands.
- Generate the statistics reports (statspack reports or AWR reports) and debug any performance problems. Guidewire recommends you download the Oracle AWR/Oracle Statspack from **Server Tools** → **Info Pages** along with **Load History Info** to debug check and load performance problems.
- Save a copy of the **DatabaseCatalogStatistics** page and archive it before each database import attempt.

IMPORTANT Designing appropriate statistics collection into all database import code dramatically improves the ability for Guidewire to advise you on performance issues related to database import.

BillingCenter has an API to update database statistics for the staging tables. Use the **TableImportAPI** web service method `updateStatisticsOnStagingTables` or use the command line tool command:

```
table_import -updatedatabasestatistics
```

Table Import Tools

BillingCenter provides a set of staging table import functions to help you with the data import process. BillingCenter exposes the staging table import tools in the following ways:

- **Table import web service** – You can use table import methods defined in the **TableImportAPI** web service. All tools are provided as synchronous methods, which do not return until the command completes. Some tools have an additional asynchronous method, which returns immediately and then performs the command as a batch process. The asynchronous versions have method names that end in “`AsBatchProcess`”. For example, `deleteExcludedRowsFromStagingTablesAsBatchProcess`.

For general information about logging into and using web services, see “Web Services Introduction” on page 27.

- **Table import command line tools** – You can use table import options of the `table_import` command line tool. For a list of options, run the following command from `BillingCenter/admin/bin` and view the built-in help.

```
table_import -help
```

For more information about the `table_import` command, see “Table Import Command” on page 195 in the *System Administration Guide*.

All servers in your BillingCenter cluster must be at the maintenance run level to call any of these import functions. The maintenance run level ensures that no end users are on the system. This prevents new data added through the user interface from interfering with bulk data imports from the staging tables.

Use the following command for each server in the cluster to set this run level.

```
system_tools -maintenance -server url -password password
```

IMPORTANT All table import commands require all BillingCenter servers in a cluster to be at the maintenance run level or the table import command fails.

Alternatively, set the run level by using the `SystemToolsAPI` web service method `setRunLevel`.

The following sections briefly describe the most useful table-based import tools.

Integrity Check Tool

This tool performs the integrity check only. It does not attempt to load the operational tables. Optionally, the tool clears the error table before starting and load the exclusion table with the distinct list of logical unit of work IDs in the error table. This is the default behavior.

This simplifies the integrity check and increases the performance of the operation. If you cannot accept this limitation, then set the `allowRefsToExistingNonAdminRows` option.

- API method: `integrityCheckStagingTableContents`
- Command line option: `integritycheck`

Integrity Check And Load Tool

This tool runs the integrity check process and, if no errors are detected, proceeds with loading the operational tables. The tool has the same options as listed earlier in the section for the error and exclusion tables and limiting to references to admin tables.

In addition, if you are using Oracle only, you can instruct the system to update database statistics on each table after data is inserted into it. This helps the database avoid bad queries caused by large changes in the size of tables which occur during import.

- API method: `integrityCheckStagingTableContentsAndLoadSourceTables`
- API method: `integrityCheckStagingTableContentsAndLoadSourceTablesAsBatchProcess`
- Command line option: `-integritycheckandload`

This tool updates estimated row and block counts on the table and indexes. This helps avoid potential optimizer issues if you reference tables with a size that qualitatively changed in the same transaction.

The Boolean parameter `updateDBStatisticsWithEstimate` updates database statistics on any table loading with estimated changes based on contents of the associated staging table. This corresponds to the optional `-estimateorastats` option for the command line tool. If you load large amounts of data, the source table grows significantly. The optimizer could choose a bad query plan based on the existing state of the database statistics. Avoid this situation by updating the database statistics to reflect the expected size of the table after the load completes. Use this feature only with Oracle databases. Otherwise, BillingCenter ignores this parameter.

Populate Exclusion Table Tool

Populates the exclusion table with all of the distinct logical unit of work IDs that are in the error table.

- API method: `populateExclusionTable`
- API method: `populateExclusionTableAsBatchProcess`
- Command line option: `-populateexclusion`

Delete Excluded Rows From Staging Tables Tool

Deletes all rows from all staging tables that match a logical unit of work ID in the exclusion table.

- API method: `deleteExcludedRowsFromStagingTables`
- Command line option: `-deleteexcluded`

Clear Exclusion Table Tool

Deletes all rows (logical unit of work IDs) from the exclusion table.

- API method: `clearExclusionTable`
- Command line option: `-clearexclusion`

Clear Error Table Tool

Deletes all rows from the error table, typically in preparation for a new integrity check.

- API method: `clearErrorTable`
- Command line option: `-clearerror`

Clear Staging Table Tool

Deletes all rows from the staging table, typically in preparation for a new integrity check.

- API method: `clearStagingTables`
- Command line option: `-clearstaging`

Update Statistics

This tool updates the database statistics on all staging tables.

- API method: `updateStatisticsOnStagingTables`
- Command line option: `-estimateorastats`

Other Import-Related Tools

The following tools are available for system-wide settings during table import. For more information about the command line tools, see the “Using BillingCenter Command Prompt Tools” on page 185 in the *System Administration Guide*. For more information about logging into and using web service APIs, see “Web Services Introduction” on page 27

Setting Run Level

Set the run level to `maintenance` before performing a table import.

- API method: `SystemToolsAPI.setRunLevel(SystemRunlevel.GW_MAINTENANCE);`
- Command line option: `system_tools -password password -maintenance`

Checking Operational Table Consistency

Check the operational database consistency before running a table import.

- API method: `SystemToolsAPI.checkDatabaseConsistency(returnAllResults);`
- Command line option: `system_tools -password password -checkdbconsistency`

IMPORTANT If you do not run consistency checks regularly, run them long before starting conversion. Allow several weeks to correct any errors you may encounter.

Populating the Staging Tables

The first step in importing data using the staging tables is mapping external data in the tables. There is a one-to-one correspondence between BillingCenter operational tables (prefix: “bc_”) and the staging tables (prefix: “bcst_”).

For example, the BillingCenter `bc_account` table would load from the `bcst_account` table.

Not all operational tables have a corresponding staging table. For example, BillingCenter does not import group hierarchies, roles, and system parameters. Also, BillingCenter excludes internal tables for the message Send Queue, the internal SMTP email queue, and statistics information. However, it does include most data, including user data using the staging tables.

- The staging version of the table (starts with `bcst_`) contains a row number column and a `LUWID` column that do not exist in the `bc_` tables. The purpose of these is described earlier.
- The following properties do not exist in the `bcst_` versions of the tables: `CreateUserID`, `UpdateUserID`, `LoadCommandID`, `BeanVersion`, `CreateTime`, and `UpdateTime` properties. BillingCenter sets these properties automatically during import into the operational tables.
- Some properties that track internal system state are not included in staging tables because BillingCenter itself sets these properties at run time.
- There are internal properties (for example, `State`) that BillingCenter cannot be certain of the value of the property upon import. Mark the account's `state` as either open or closed upon import using the appropriate typelist codes.

In the BillingCenter *Data Dictionary*, you can tell which properties are included in the `bcst_` table by checking if a property is loadable.

For example, click on an entity in the *Data Dictionary* and look at each property to see whether the dictionary says “`(loadable)`”. If it does, that property is included in the staging table for that entity.

Note: Use load command IDs to tell whether an entity loaded from database staging tables. See “Load Commands and Loadable Entities” on page 370.

During database import planning, use the **Conversion View** in the *Data Dictionary*. Get to the conversion view from the main **Home** page of the *Data Dictionary*, with the links that say:

- [Data Entities \(Conversion View\)](#)
- [Typelists \(Conversion View\)](#)

The **Conversion View** hides all properties that are not backed by actual database columns or are not importable in staging tables. For example, all virtual properties, which are generated (calculated) from other properties at run time.

Important Properties and Concepts for Database Import

Most business data properties are straightforward to populate in the staging tables. You can look at the *Data Dictionary* to understand the meaning of the properties on the operational tables. Put the right values into the properties by the same name on the staging tables. However, there are some general guidelines for how to populate the staging tables, described in the following subsections.

Public IDs and the Retired Property

BillingCenter relies on `PublicID` strings to identify rows in the staging tables. Every row you insert must have a `PublicID`, which must be unique within both the staging table and its corresponding operational table. For tables with a `retired` property, the combination of `PublicID` and `Retired` must be unique.

In the operational tables, logical deletes are done by setting the `Retired` property to something other than 0 (in practice, it is set to the ID of the row). This allows more than one row to have the same `PublicID` as long as only one is currently not retired. BillingCenter uses this to make sure that you are not inserting duplicate rows and to resolve foreign keys (references between tables).

You must have an algorithm for generating unique `PublicID` strings in the conversion tool. For example, the `PublicID` for each account could be the company name, then a colon character, then the account number.

Set the retired property (if any) to the value 0, indicating an active row in the operational tables. To retire a row, set the `retired` property to the object's public ID. Once an object is retired, it can never revert to active.

WARNING Guidewire recommends that you not load retired data during staging table import.

Integration code must never explicitly set a public IDs to a string that starts with a two-character ID and then a colon because Guidewire reserves all such IDs. If you used the `PublicIDPrefix` configuration parameter to change the prefix of auto-created public IDs, integration code that sets explicit public IDs also must not conflict with that namespace. Additionally, plan your public ID naming carefully to support large (long) record numbers. Ensure your system can support a significant number of records over time and stay within the 20 character public ID limit. For detailed discussion on these issues, see “Public IDs and Integration Code” on page 22.

Foreign Key Fields

During the load phase, BillingCenter looks up the foreign key by public ID and insert its internal ID into the operational table.

All Columns in the Staging Tables Are Nullable

Most columns in the staging tables are nullable, even if the corresponding column in the operational table is not nullable. This is designed to allow you to insert rows and then set default values afterwards, if necessary, while loading the staging tables.

However, if the corresponding column in the operational table has a default value (check the *Data Dictionary*), then you can leave the staging table column `null`. In this case, BillingCenter sets the default value. If the property does not have a default value, you must set some value in the staging table even if merely a default value that your conversion tool chooses.

Type Key Columns

Type key columns must be loaded by a valid Code value from the type list. During population, BillingCenter converts this to an internal integer type code ID.

Subtyped Tables

If the operational table is subtyped, then there is a `subtype` column on the staging table. This property must load by the text name of the subtype. Look in the *Data Dictionary* for the subtype names.

Ignore the Row Number Column

Ignore the row number column. This column is automatically populated by BillingCenter by the database on inserting the row or by the integrity check process, depending on which database management system (DBMS) you are using. The row number property is only used for reporting errors.

Populating the Logical Unit of Work ID (LUWID) Column

Populate the `LUWID` column with something useful. For accounts, typically this is the account number, but other values could be used. If inserting users, it might appropriate to use the user's login name (for example, `ssmith`). This way, you can delete all the data related to a user (in other words, `ccst_user`, `ccst_credential`, `ccst_contact`) if the user already exists in BillingCenter.

Overwritten Tables and Columns

There are also tables that have staging tables but are always overwritten during import. These are marked within the data model files as `overwritteninstagingtables` set to `true`.

Likewise, some columns are defined with the `loadable` attribute and also the attribute `overwriteinstagingtables`. These columns in the staging table are usually cleared and populated by the staging table loader subroutines.

Virtual Properties

Some Guidewire entities contain properties called virtual properties (virtual fields). They act like regular properties in many ways from Gosu such as reading property values. However, virtual properties are not backed from columns in the database. Values of these properties generate dynamically if Gosu code requests the property value. In many cases, virtual properties are read-only from Gosu code, from Java plugins, and from Java classes called from Gosu. Some virtual properties are also writable. Refer to “Properties” on page 201 in the *Gosu Reference Guide* for related information.

Generally speaking, ignore all virtual properties for conversion projects using staging tables.

To determine where to load data for some virtual property (virtual field), there is no one answer. In some cases the virtual property calls a method that performs a complex calculation that pull from many different properties or even other entities.

As a general rule, to find out what properties the virtual property pulls information from, see the *BillingCenter Data Dictionary*. Sometimes the *Data Dictionary* contains the necessary information for staging table database import for how that virtual property is calculated. However, you are only responsible for populating loadable properties and columns. Use the Conversion View of the *Data Dictionary* to show only loadable entities and loadable columns that your conversion tool must populate.

Indexes in Staging Tables

All staging tables must contain a unique index on each table before loading the staging tables. The system applies an integrity constraint during load that there is a unique index on all staging tables. If the table has a retired property, the unique index must be on the combined columns (`publicID`, `retired`). If the table does not have a retired property, the unique index must be on the `publicID` column.

Loading BillingCenter-specific Entities

The BillingCenter data model is complex, so loading data through staging tables requires careful attention. Generally, you load data from staging tables only once, during migration from your legacy system to BillingCenter.

This topic includes:

- “BillingCenter Loadable Entities” on page 382
- “BillingCenter Derived Entities” on page 383
- “BillingCenter Non-Loadable Administrative Entities” on page 383
- “BillingCenter Data Loading Requirements” on page 384
- “Loading Commissions” on page 384
- “Loading Past (Closed) Delinquencies” on page 385
- “Loading Active (Open) Delinquencies” on page 385
- “Loading Agency Bill Data” on page 386
- “Loading Reversals” on page 387
- “Loading Disbursements” on page 388
- “Loading Payments” on page 388
- “Batch Processing Requirements” on page 388

BillingCenter Loadable Entities

Consult the *Data Dictionary* to learn whether a BillingCenter entity is loadable. The most important loadable entities include:

- Account
- Policy
- PolicyPeriod
- Invoice
- Charge
- Payment
- Contact
- Producer

Be sure to set the following properties manually before triggering staging table import:

- Account.InsuredID
- PolicyPeriod.PrimaryInsuredID
- Producer.PrimaryContactID

BillingCenter Derived Entities

BillingCenter automatically generates the following entities on loading:

- TAccount
- Transaction
- LineItem entities for each Transaction

Some entities have denormalized properties, which BillingCenter also generates:

- Account.ChargeHeld
 - PolicyPeriod.ChargeHeld
 - PolicyPeriod.PolicyNumberLong
- Populate this property manually. It has a denormalized value, but it depends on your customization of the PolicyPeriod.DisplayName property. You must carefully confirm this value, because its value is not verified by validations or integrity checks.
- PolicyPeriod.PrimaryProducerCodeID
 - DelinquencyProcess.CurrentEventDenorm
 - Invoice.Amount
 - Invoice.AmountDue
 - Invoice.PaidStatus
 - StatementInvoice.NetAmount
 - StatementInvoice.NetAmountPaid
 - InvoiceItem.PaidAmount
 - ItemCommission.PaidCommission

BillingCenter Non-Loadable Administrative Entities

Most entities that relate to the **Administration** tab in the user interface cannot be loaded. However, the following administrative entities can be referenced from data in staging tables:

- Administrative data, such as user, groups, and roles
- Plans, including charge patterns
- Suspense payments
- Trouble tickets

BillingCenter Data Loading Requirements

Follow these data loading requirements as you convert data from your legacy system to BillingCenter:

- Load an account once only. BillingCenter does not support reloading policies on existing accounts in the system. All policies, including open and future policies, must be loaded at the same time as their accounts.
- All accounts and policy periods must load with all invoices on the account, including future invoices. BillingCenter assumes that your legacy system generated all invoices at policy creation time.
- Public IDs must be unique across of all types of T-Account owners; for example, Account, PolicyPeriod, and Producer. The public IDs would be the public IDs of the generated TAccountOwner entities, so these must be unique.
- During the loading of charges, BillingCenter creates a single revenue recognition transaction on the initial date, even if the charge is a pro rata charge. This is different than charges that are created by BillingCenter where pro rata charges have an earning schedule. This might impact equity dating and may require careful planning during import.
- BillingCenter marks all plans in the system at the time of loading as In Use. They are un-editable after loading.
- TAccountContainer entities must load at the same time as entities they are associated with.
- During loading of payments, there is a difference between loading account payments for direct billing and agency payments for agency billing. Load account payments only as Payment entities. BillingCenter distributes payments by running New Payment batch processing against policies, charges, and invoice items. All InvoiceItem entities of AccountInvoice entities must have zero paid amounts. However, the distribution of agency payments must be specified in staging tables in the InvoiceItem.paidAmount property. BillingCenter loads agency non-distributed payments to the producer's unapplied T-Account, where they stay until distributed manually through the user interface.
- Run the types of batch processing specified in “Batch Processing Requirements” on page 388 to complete the migration of your legacy data and generate corresponding data within BillingCenter.

See also

- “High-Level Steps in a Typical Database Staging Table Import” on page 372

Loading Commissions

To load commissions:

1. Specify commission rate overrides on any charges that earn commission. For example, have a PolicyProducerCodeCharge associated with the charges.

To populate the CommissionRateOverride table, set the following properties:

- Set the Rate property to the rate of the commission in percent (10.00 is equivalent to 10%).
- Set the ChargeID property to the charge on which to override the commission.
- Set the Role property to the role of the producer code (Primary, Secondary, Referrer, and so on) on the associated PolicyPeriod.

If a commission amount differs from the initial commission rate in your legacy system, you must calculate the percentage from the amount of commission actually earned. On import, BillingCenter uses this percentage to create transactions of the appropriate amount.

2. Load the ProducerPayment table and rows in the OutgoingPayment table for each payment.

The OutgoingPayment table is used for both ProducerPayment and Disbursement, and represents the actual mechanism for sending payments out from the system, such as writing checks. The ProducerPayment entity has a 1-to-1 on OutgoingPayment. Each payment you send must have a row in that table with a foreign key that refers to the payment. The subtype of ProducerPayment must be ManualPayment.

3. Run the import.

Later, after you load and import your legacy commissions data, the following types of batch processing complete the migration.

- Policy Closure batch processing – Generates transactions that move funds from the Commissions Reserve to the Commissions Payable at the charge level.
- Producer Payment batch processing – Generates transactions that move funds from the Commissions Payable at the charge level to the Commissions Payable at the producer level. It then creates the necessary ProducerPaymentSent transactions, which subtract from the insurer's cash and producer's Payable T-Accounts. The insurer sends any positive balance on the producer as an auto-payment.

See also

- “Batch Processing Requirements” on page 388

Loading Past (Closed) Delinquencies

To load past (closed) delinquencies, populate the bcst_delinquencyprocess and bcst_delinquencyprocessevent tables. Every delinquency process has multiple ProcessEvent entities. Because these delinquencies have already run their course by the time of import, they are unrestricted in number and name. Be sure to give these past delinquency processes a status of closed.

Note: You can see these past events from the **Account / Delinquencies** screen.

Loading Active (Open) Delinquencies

To load active (open) delinquencies, create a delinquency plan in BillingCenter, based closely on the logic used in your legacy system. In your test cases, call this plan “Legacy Delinquency Plan”. Put this name in the script parameter LegacyDelinquencyPlan. This string must match the delinquency plan that you want to compare your legacy processes against.

For each reason on the plan, you must define a workflow, which is standard for BillingCenter. However, you need to provide additional code in the workflows you define for them to work after staging table import.

BillingCenter provides a template for your workflow definitions in the file LegacyDelinquency.1.xml. The most important code in the template allows BillingCenter to skip already completed process events. For example:

```
<Enter>
    // Get the first dunning event instance for the current work flow
    var eventName = DelinquencyEventName.TC_DUNNINGLETTER1
    var event = dlnqProcess.getProcessEventById(eventName.Code)

    // Send the first dunning letter if not sent already.
    if (! event.Completed) {
        plcypPeriod.sendDunningLetter()
        event.flagCompleted() // Set completion time and mark step as current.
    } else {
        event.flagCurrent() // Mark step as current without changing completion time.
    }
</Enter>
```

In the example above, TC_DUNNINGLETTER1 refers to whatever event name corresponds to this delinquency event, and sendDunningLetter is the action to take. The flagCompleted method marks the completed time (CompletionTime) on the event, and sets this step to be the current event (CurrentEvent) in the process. The flagCurrent method sets this step to be the CurrentEvent in the process but does not overwrite the CompletionTime loaded from staging tables.

Staging Tables

Populate the `bcst_delinquencyprocess` and `bcst_delinquencyprocessevent` staging tables. Every delinquency process has multiple process events. The process must have one event for every `DelinquencyPlanEvent` entity in the delinquency reason on the corresponding legacy plan, matched by their `EventName` properties. An integrity check verifies this.

WARNING No date restrictions ensure that dates occur in the correct order or to match appropriate `PlanEvent.DaysAfterInception` properties. Carefully set all dates, or problems may occur.

BillingCenter enforces the `CompletionTime` property to occur in the past. Events with the `CompletionTime` property specified do not execute the corresponding action for this event after import. Events without a completion time property do run, so BillingCenter sends the dunning letters.

After Import

After delinquency import, run Legacy Delinquency batch processing, which operates on delinquency processes that match the following criteria:

- The associated account uses the Legacy Delinquency Plan.
- The status is Open.
- There is no associated workflow.

The process starts a workflow for each delinquency. The workflow skips steps with a non-null `CompletionTime` property and stops after all these steps, which completed in the past or present. The workflow resumes uncompleted steps on future runs of Workflow batch processing.

After loading and importing delinquencies for a legacy delinquency plan, you can change the delinquency plan for an account by using the Workflow editor.

Loading Agency Bill Data

In a production environment, users can create payments and save them to the database without executing them. However, BillingCenter does not support loading payments in this saved, unexecuted state. All payments must load as executed. BillingCenter creates the corresponding transactions after import.

BillingCenter has an integrity check to ensure that the `AgencyMoneyReceived.MoneyReceivedDate` is non-null. BillingCenter needs these dates to create the transactions.

BillingCenter loads the following agency bill denormalized properties and transactions:

- `StatementInvoice.NetAmount`
- `StatementInvoice.NetAmountPaid`
- `InvoiceItem.PaidAmount`
- `ItemCommission.PaidCommission`
- `AgencyMoneyReceivedTxn`
- `ChargePaidFromProducer` (transaction)
- `ChargeWrittenOff` – also used for direct bill
- `CommissionWrittenOff` – also used for direct bill

Write-offs

Load an `AgencyPmtDiffWriteoff` entity for every invoice item that you want written off. BillingCenter ignores the `Amount` field of the write-off and creates a positive or negative write-off for the difference between the item amount and the paid amount. If the item is underpaid, BillingCenter creates a positive write-off for the difference. If the item is overpaid, BillingCenter creates a negative write-off for the difference.

An integrity check verifies that write-off amounts are zero.

Agency Cycle Processes

BillingCenter supports loading `AgencyCycleProcess` entities on any billed `StatementInvoices`. Processes to load must include whether the step has been completed and the scheduled date of the step. An integrity check verifies that any steps that are marked as completed have a date in the past.

The `LegacyAgencyBill.1.xml` file demonstrates how to modify a workflow to work with steps that have executed already. Any step you do not want to execute again, add a condition to the step to check if the bit on the process is already true; for example:

```
!agencyCycleProcess.Dunning1Sent
```

Some steps, such as promise steps, do not perform any actions in the base configuration. For steps that do not affect agents or insureds, conditional logic is unnecessary.

Run Legacy Agency Bill batch processing after successfully loading `AgencyCycleProcess` entities. The process finds all `AgencyCycleProcesses` under the Legacy Agency Bill plan and starts a workflow for that process. The process stops before any steps scheduled for the future. The script parameter `LegacyAgencyBillPlan` specifies which agency bill plan the process uses as the Legacy Agency Bill plan.

IMPORTANT Run Legacy Agency Bill batch processing once only, after you initially load `AgencyCycleProcess` entities.

See also

- “Guidewire Workflow” on page 341 in the *Configuration Guide*
- “Legacy Agency Bill Batch Processing” on page 132 in the *System Administration Guide*

Loading Reversals

The way you load reversals depends on the type of item being reversed:

- Write-offs
- Payments
- Charges

Write-off Reversals

For write-off reversals, populate the `bcst_writeoffreversal` table with `WriteoffID` referencing the write-off to be reversed. BillingCenter generates duplicate write-offs in the `writeoff` table. Set `reversed` to 1 on the original write-off, and add an entry to the `bcst_rev` edge table to relate original write-offs with reversal write-offs.

Later, when you run Write-off Staging batch processing, BillingCenter creates transactions for original write-offs, and then reverses them.

Payment Reversals

For payment reversals, set the `ReversalReason` and `ReversalDate` fields on the `bcst_payment` rows to be reversed. Also, set the `status` field to `distributed`. An integrity checks verifies that distributed payments have reversal fields and non-distributed payments do not.

Later, when you run New Payment batch processing, BillingCenter skips payment reversals because they are marked as distributed. Transactions and reversed transactions are not created, but you can find historic data based on the `ReversalReason` and `ReversalDate` fields.

Charge Reversals

For charge reversals, set the `reversed` field to 1. Add another charge with an amount equal to the negative of the charge amount, and set the `reversed` field to 0. Also, you need invoice items that are reversals for the original invoice items. Add a row to `bcst_revcharge` where `ownerid` references the new reversal charge and `foreignentityid` references the old reversal charge.

BillingCenter creates the `InitialChargeTxn` and `NegativeChargeBilled` transactions. These transactions move the amount of the charge into the `PP.Unapplied TAccount`. Like any negative charge, BillingCenter applies the amount the next time that `Invoice Due` batch processing runs.

Loading Disbursements

The way you load disbursements depends on the nature of specific disbursements:

- Account disbursements
- Agency disbursements

Note: Collateral and suspense disbursements are not loadable.

Account Disbursements

For account disbursements that are paid already, set `CloseDate` to occur in the past, set `Status` to `Sent`, and relate an `OutgoingPayment` with any status to them. BillingCenter creates `DisbursementPaid` transactions for these disbursements to move funds from `Account.Unapplied` to `Account.Cash`.

An integrity check verifies that `CloseDate` occurs in the past only if `Status` is set to `Sent`. Another integrity check verifies that account disbursements with `Status` set to `Sent` are referenced by an `OutgoingPayment`.

For voided account disbursements, set `CloseDate` to occur in the past and set `Status` to `Voided`. BillingCenter does not create disbursement transactions or reversals. It simply loads voided disbursements.

Agency Disbursements

For agency disbursements that are paid already, set `CloseDate` to occur in the past, set `Status` to `Sent`, and relate an `OutgoingPayment` with any status to them. BillingCenter creates `AgencyDisbursementPaid` transactions for these disbursements to move funds from `Producer.Unapplied` to `Producer.Cash`.

An integrity check verifies that `CloseDate` occurs in the past only if `Status` is set to `Sent`. Another integrity check verifies that agency disbursements with `Status` set to `Sent` are referenced by an `OutgoingPayment`.

For voided agency disbursements, set `CloseDate` to occur in the past and set `Status` to `Voided`. BillingCenter does not create disbursement transactions or reversals. It simply loads voided disbursements.

Loading Payments

The way you load direct billing payments differs from the way you load and agency billing payments. Load direct billing payments as `Payments` entities, and then run `New Payment` batch processing to distribute the payments to policies, charges, and invoice items. Load agency billing payments as `AgencyPaymentItems` with the `toward` toward which `InvoiceItems`. The denormalized `InvoiceItem.PaidAmount` field will be calculated based on these `AgencyPaymentItems`.

Batch Processing Requirements

After successfully loading and importing your legacy data into BillingCenter, run the following types of batch processing in the order listed:

1. Database Statistics
2. Charge Pro Rata Transaction

3. Write-off Staging
4. Invoice
5. Legacy Collateral
6. New Payment
7. Legacy Delinquency
8. Legacy Agency Bill
9. Statement Billed
10. Invoice Due
11. Statement Due
12. Database Statistics, a second time
13. Policy Closure
14. Commissions Payable Calculations
15. Producer Payment

See also

- For information on running the types of batch processing listed above, see “List of Work Queues and Batch Processes” on page 120 in the *System Administration Guide*.

Data Integrity Checks

Before loading staging table data into the operational database tables, BillingCenter runs a broad set of BillingCenter-specific data integrity checks. These checks find and report problems that would cause the import to fail or put BillingCenter into an inconsistent state.

BillingCenter requires that data integrity checks succeed as the first step in the load process. This means that even if errors are found and these rows were removed, BillingCenter still requires rerunning integrity checks before your data is reloaded.

Contrast data integrity checks with other validations:

- Integrity checks check different things from requirements that the user interface (PCF) code enforces. For example, a property that is nullable in the database may be a property that users must set in the BillingCenter user interface. Importing a `null` value in this property is acceptable for database integrity checks. However, if you edit the object containing the property in the BillingCenter interface, BillingCenter requires you to provide a non-`null` value before saving because of data model validation.
- Integrity checks are different from validation rule sets and the validation plugin.

See also

- For more details, see “Load Integrity Checks” on page 170 in the *System Administration Guide*

Examples of Integrity Checks

The following list is a partial list of data integrity checks that BillingCenter enforces during database import:

- No duplicate `PublicID` strings within the staging tables or in the corresponding operational tables
- No unmatched foreign keys
- No missing, required foreign keys
- No invalid codes for type key properties

- No `null` values in non-`null` (operational) properties that do not provide a default. Empty strings and text containing only space characters are treated as `null` values in data integrity checks for non-nullable properties.
- No duplicate values for any unique indexes.

You might notice that this list does not include enforcing property formats for properties that use a field validator in the user interface. For example, enforcing a standard format of account number.

See also

- For a full list of integrity checks, see the [Load Integrity Checks](#) page within the BillingCenter Server Tools tab. You can view all integrity check SQL queries.

Why Integrity Checks Always Run Before Loading

There are many reasons for BillingCenter to rerun integrity checks during any load request, even in situations in which the conversion tool believes that it fixed all load errors. For example:

- If the logical unit of work IDs were not populated correctly, removing a account could leave extra rows in the staging tables that were not properly tied to the account.
- If errors were found during population of the operational tables, the entire process must roll back the database. Rolling back the database changes typically is slow and resource-intensive. It is much better to identify problems initially rather than trigger exceptions during the load process that require rolling back changes.
- Even if you remove all error rows, integrity check violations can occur for certain errors that cannot be tied to a single row. Because some errors cannot be tied to a particular row, there is no associated logical unit of work ID

Table Import Tips and Troubleshooting

Some things to know about importing using the staging tables and safely and successfully running the process:

- All staging table import commands require the servers to be in maintenance mode, formally known as the maintenance run level. This prevents users from logging into the BillingCenter application.
- BillingCenter runs the load process inside a single database transaction to be able to roll back if errors occur. This means that a large rollback space may be required. Run the import in smaller batches (for example, a few thousand records at time) if you are running out of rollback space.
- As with any major database change, make a backup of your database prior to running a major import.
- After loading staging tables, update database statistics on the staging tables. Update database statistics on all the operational tables after a successful load. After a successful load, BillingCenter provides table-specific update database statistics commands in the `cc_loaddbstatistics` command table. You can selectively update statistics only on the tables that actually had data imported, rather than for all tables.
- Always run database consistency checks on the operational database tables both before and after table imports. Assuming there were no consistency errors before importing, consistency errors after an import indicates that there is something wrong with the data in the staging tables uncaught by integrity checks. See “Overview of a Typical Database Staging Table Import” on page 371 for example commands.
- During BillingCenter upgrade, the BillingCenter upgrader tool may drop and recreate the staging tables. This occurs for any staging table in which the corresponding operational table changed and requires upgrade.

WARNING Never rely on data in the staging tables remaining across an upgrade. Never perform a database upgrade during your import process.

- Integrity checks during staging table import do not use field validators. If you need to ensure that a field fits a certain pattern, be sure to include that logic in your conversion tool before the data gets to the staging tables

Staging Table Import of Encrypted Properties

If you are importing records by using staging table import and some data contains encrypted properties, you can use a BillingCenter tool that encrypts encrypted properties in your staging tables.

See also

- “Encryption Features for Staging Tables” on page 259

Other Integration Topics

Contact Integration

This topic discusses how to integrate BillingCenter with an external contact management system other than ContactManager.

The base BillingCenter configuration has built-in support for integration with Guidewire ContactManager. If you have a license for the optional Client Data Management module, you can install and integrate ContactManager with BillingCenter. For more information, see “Integrating ContactManager with Guidewire Core Applications” on page 45 in the *Contact Management Guide*.

This topic includes:

- “Integrating with a Contact Management System” on page 395
- “Contact Web Service APIs” on page 401

See also

- “Plugin Overview” on page 135
- “Web Services Introduction” on page 27

Integrating with a Contact Management System

Integrate with an external contact management system by registering a class that implements the `ContactSystemPlugin` plugin interface.

For use with internal contacts only, BillingCenter includes the following `ContactSystemPlugin` plugin interface implementation:

```
gw.plugin.contact.impl.StandAloneContactSystemPlugin
```

The BillingCenter base configuration supports integration with Guidewire ContactManager. If you have a Client Data Management module license and install ContactManager, do not write your own implementation of the `ContactSystemPlugin` plugin interface. Instead, use the plugin implementation `gw.plugin.contact.impl.ABContactSystemPlugin`.

IMPORTANT If you have a license for the optional Client Data Management module, you can install and integrate ContactManager with BillingCenter. For information on integration with ContactManager, see “Integrating ContactManager with Guidewire Core Applications” on page 45 in the *Contact Management Guide*. For more information about the mapping between BillingCenter and ContactManager contacts, see “ContactMapper Class” on page 243 in the *Contact Management Guide*.

Your implementation of the contact system plugin defines the contract between BillingCenter and the code that initiates requests to an external contact management system. BillingCenter relies on this plugin to:

- Retrieve a contact from the external contact management system.
- Search for contacts in the external contact management system.
- Add a contact to an external contact management system.
- Update a contact in an external contact management system.

BillingCenter uniquely identifies contacts by using unique IDs in the external system, known as the Address Book Unique ID (AddressBookUID). This unique ID is analogous to a public ID, but is not necessarily the same as the public ID, which is the separate property `PublicID`.

When BillingCenter needs to retrieve a contact from the contact management system, it calls the `retrieveContact` method of this plugin. The plugin takes only one argument, the unique ID of the contact. Your plugin implementation needs to get the contact from the external system by using whatever network protocol is appropriate, and then populate a contact record object.

For this plugin, the contact record object that you must populate is a `Contact` entity instance.

Inbound Contact Integrations

For inbound integration, BillingCenter publishes a WS-I web service called `ContactAPI`. This web service enables an external contact management system to notify ClaimCenter of updates, deletes, and merges of contacts in that external system. When ClaimCenter receives a notification from the external contact management system, ClaimCenter can update its local copies of those contacts to match. See “Contact Web Service APIs” on page 401.

Supporting a Contact Management System

To support a contact management system in BillingCenter, you must create a plugin implementation and register it in the plugin registry. Additionally, the plugin must implement the `supportsExternalContactSystemIntegration` method and return `true`. If you return `false`, BillingCenter treats the plugin implementation as if it were not registered.

Asynchronous Messaging with the Contact Management System

When a user in BillingCenter creates or updates a local `Contact` instance, BillingCenter runs the Event Fired rule set. These rules determine whether to create a message that creates or updates the `Contact` in the contact management system. In the default configuration, the process of sending the message to the contact system involves several messaging objects:

- A built-in `ContactSystemPlugin` plugin implementation.
- A built-in messaging destination that is responsible for sending messages to the contact system. The messaging destination uses two plugins:

- A message transport (`MessageTransport`) plugin implementation called `ContactMessageTransport`. BillingCenter calls the message transport plugin implementation to send a message.
- A message request (`MessageRequest`) plugin implementation called `ContactMessageRequest`. BillingCenter calls the message request plugin to update the message payload immediately before sending the message if necessary.

For more information about messaging and these plugin interfaces, see “Messaging and Events” on page 303

In the default configuration, BillingCenter calls the following two methods of `ContactSystemPlugin`:

- `createAsyncUpdate` – At message creation time, Event Fired rules call this method to create a message.
- `sendAsyncUpdate` – At message send time, the message transport calls this method to send the message.

Detailed Contact Messaging Flow

1. At message creation time, the Event Fired rules call the `ContactSystemPlugin` method called `createAsyncUpdate` to actually create the message. The Event Fired rules pass a `MessageContext` object to the `ContactSystemPlugin` method `createAsyncUpdate`. The `createAsyncUpdate` method uses the `MessageContext` to determine whether to create a message for the change to the contact, and, if so, what the payload is for that message.
2. At message send time on the batch server, ClaimCenter sends the message to the messaging destination. There are two phases of this process:
 - a. BillingCenter calls the message request plugin to handle late-bound `AddressBookUID` values. For example, suppose the `AddressBookUID` for a contact is unknown at message creation time but is known when at message send time. BillingCenter calls the `beforeSend` method of the `ContactMessageRequest` plugin to update the payload before the message is sent. For related information, see “Late Binding Data in Your Payload” on page 342.
 - b. BillingCenter calls the message transport plugin to send the message. The message transport implementation finds the `ContactSystemPlugin` class and calls its `sendAsyncUpdate` method to actually send the message. An additional method argument includes the modified late-bound payload that the message request plugin returned.

Contact Retrieval

To support contact retrieval, do the following in your `retrieveContact` method:

1. Determine what object type to populate locally for your contact to connect to the external system.

For example, suppose you define the web service in Studio with the name `MyContactSystem` and that web service has a `MyContact` XML type from a WSDL or XSD file. For this example, suppose your contact retrieval web service returns a `MyContact` object.

Note: For more information about web services and how BillingCenter imports types from external web services, see “Calling Web Services from Gosu” on page 65.

2. Write your web service code that connects to the external contact management system. Whatever type it returns must contain equivalents of the following `Contact` properties:

`PrimaryPhone`, `TaxID`, `WorkPhone`, `EmailAddress2`, `FaxPhone`,
`HomePhone`, `CellPhone`, `DateOfBirth`, and `Version`.

If the contact is a person, it must have the properties `FirstName` and `LastName`.

If the contact is a company, it must have the property `Name`, containing the company name.

3. Your plugin code synchronously waits for a response.

4. To return the data to BillingCenter, create a new instance of a contact in the current transaction's bundle.

Create the new instance by using the instance of ContactCreator that your plugin method gets as a method argument. This class standardizes finding and creating contacts within BillingCenter. It has a `loadOrCreateContact` method that you can use for creating a contact.

For example, your code might look like something like this:

```
override function retrieveContact(addressBookUID : String, creator : ContactCreator) : Contact {  
    var returnedContact : Contact = null  
    var contactXml = retrieveContactXML(addressBookUID)  
    if (contactXml != null) {  
        var contactType =  
            _mapper.getNameMapper().getLocalEntityName(contactXml.EntityType)  
        returnedContact = creator.loadOrCreateContact(contactXml.LinkID, contactType)  
  
        validateAutoSyncState(returnedContact, addressBookUID)  
        overwriteContactFromXml(returnedContact, contactXml)  
    }  
    return returnedContact  
}  
  
var c = new Contact()
```

5. Populate the new contact entity instance with information from your external contact.

6. Return that object as the result from your `retrieveContact` method.

If you ever need the BillingCenter implementation to retrieve more fields from the external contact management system, it is best to extend the data model for the `Contact` entity.

Note: If you need to view or edit these additional properties in BillingCenter, remember to modify the contact-related PCF files to extract and display the new field.

Contact Searching

To support contact searching, your plugin must respond to a search request from BillingCenter. BillingCenter calls the plugin method `searchContacts` to perform the search. The details of the search are defined in a contact search criteria object, `ContactSearchCriteria`, which is a method argument. This object defines the fields that the user searched on.

The important properties in this object are:

- `ContactIntrinsicType` – The type of contact. To determine if the contact search is for a person or a company, use code such as the following:

```
var isPerson = Person.Type.isAssignableFrom(searchCriteria.ContactIntrinsicType)
```

If the result is `true`, the contact is a person rather than a company.

- `FirstName`
- `Keyword` – The general name for a company name (for companies) or a last name (for people)
- `TaxID`
- `OfficialId`
- `OrganizationName`
- `Address.AddressLine1`
- `Address.City`
- `Address.State.Code`
- `Address.PostalCode`
- `Address.Country.Code`
- `Address.County`

Refer to the Data Dictionary for the complete set of fields on the search criteria object that you could use to perform the search. However, the built-in implementation of the user interface might not necessarily support populating those fields with non-null values. You can modify the user interface code to add any existing fields or extend the data model of the search criteria object to add new properties.

Note: Properties related to proximity search are unsupported. For example, you cannot search by attitude or longitude using this API.

The second parameter to this method is the `ContactSearchFilter`, which defines some metadata about the search, including:

- The start row of the results to be returned.
- The maximum number of results, if you are just querying for the number of results, as opposed to the actual results.
- The sort columns.
- Any subtypes to exclude from the search due to user permissions.

For the return results, populate a `ContactResult` object, which includes the number of results from the query and, if required, the actual results of the search as `Contact` entities. It is not expected that these `Contact` entities will contain all the contact information from the external contact management system. These entities are expected to contain just enough information to display in a list view to enable the user to select a result.

For example, the default configuration of the plugin for ContactManager includes the following properties on the `Contact` entities returned as search results:

- `AddressBookUID` – The unique ID for the contact as `String`
- `FirstName` – First name as `String`
- `LastName` – Last name as `String`
- `Name` – Company name as `String`
- `DisplayAddress` – Display version of the address, as a `String`
- `PrimaryAddress` – An address entity instance
- `CellPhone` – Mobile phone number as `String`
- `HomePhone` – Home phone number as `String`
- `WorkPhone` – Work phone number as `String`
- `FaxPhone` – Fax phone number as `String`
- `EmailAddress1` – Email address as `String`
- `EmailAddress2` – Email address as `String`

For the full list of properties, see the `ABContactAPISearchResult` Gosu class in ContactManager. This class is in the package `gw.webservice.ab.ab800.abcontactapi`.

Support for Finding Duplicates

Your plugin must tell BillingCenter whether the external contact management system supports finding duplicates. Implement the `supportsFindingDuplicates` method. Return `true` if the external system is capable of finding and returning duplicate contacts. If this method returns `false`, then a call to the `findDuplicates` method might throw an exception, but in any event cannot be relied upon to return legitimate results.

Finding Duplicates

BillingCenter calls the `findDuplicates` method in the plugin to find duplicate contacts for a specified contact.

The method takes two arguments:

- A `Contact` entity instance for which you are checking for duplicates

- A `ContactSearchFilter` that can specify subtypes to exclude from the results if contact subtype permissions are being used

The `findDuplicates` method must return an instance of `DuplicateSearchResult`, a class that contains a collection of the potential duplicates as `ContactMatch` objects and the number of results.

The `ContactMatch` object contains the `Contact` that was found to be a duplicate and a `boolean` property, `ExactMatch`, which indicates if the contact is an exact or a potential match. As with searching, the `Contact` returned in the `ContactMatch` object does not have to contain all the contact information from the external contact management system. The object contains just enough contact information to enable the BillingCenter user to determine if the duplicate is valid.

The list of properties returned by `ContactManager` in its default configuration is in the `ABContactAPIFindDuplicatesResult` class in the package `gw.webservice.ab.ab800.abcontactapi`.

To find duplicates

1. Query the external system for a list of matching or potentially matching contacts.
2. Optionally, if the results you receive are only summaries, if it is necessary for detecting duplicates, retrieve all the data for each contact from the external system.
3. Review the duplicates and determine which contacts are duplicates according to your plugin implementation.
4. Create a list of `ContactMatch` items, each of which contains a `Contact` and a `boolean` property that indicates if the contact is an exact match. Each of these `Contact` objects is a summary that contains many, but perhaps not all, `Contact` properties.
5. Create an instance of the concrete class `DuplicateSearchResult` with the collection of `ContactMatch` items. Pass the list of duplicates to the constructor.
6. Return that instance of `DuplicateSearchResult` from this method.

Adding Contacts to the External System

To support BillingCenter sending new contacts to the external contact management system, implement the `addContact` methods in your contact system plugin. This method must add the contact to the external system. Send as many fields on `Contact` as make sense for your external system. If you added data model extensions to `Contact`, you must decide which ones apply to the external contact management system data model. There might be side effects on the local contact in ClaimCenter, typically to store external identifiers.

Alternative Method Signatures for Adding a Contact to an External System

The `addContact` method has two signatures:

```
addContact(Contact contact, String payload, String transactionId)  
addContact(Contact contact, String transactionId)
```

The `payload` parameter uniquely describes the contact for the external system in whatever format the external system accepts, such as XML specified by a published XSD. For a typical integration, your Event Fired rules generate an XML serialization of your contact. The `payload` parameter represents the `Message.Payload` property of the message.

The simpler method signature takes a `Contact` entity and a transaction ID.

The transaction ID parameter uniquely identifies the request. Your external system can use this transaction ID to track duplicate requests from ClaimCenter.

Capturing the Contact ID from the External System

The `addContact` method returns nothing to BillingCenter. However, your implementation of the method must capture the ID for the new contact from the external system. Use the captured value to set the address book UID on the local BillingCenter contact before your `addContact` method returns control to the caller. Typically, the external system is the system of record for issuing address book UIDs.

Updating Contacts in the External System

If a BillingCenter user updates a contact's information, BillingCenter sends the update to the contact management system by calling the `updateContact` method of the contact system plugin. Just like the `addContact` method, one parameter to `updateContact` is a `Contact` entity. The second version of the method adds a transaction ID that can be used by the external contact management system to track if an update has already been applied.

Contact Web Service APIs

The web service `ContactAPI` provides external systems a way to interact with contacts in BillingCenter. This web service is WS-I compliant. Refer to the implementation class in Studio for details of all the methods in this API. Each of the methods in this API that changes contact data in BillingCenter requires a transaction ID, which must be set in the SOAP request header for the call.

See also

- “Web Services Introduction” on page 27

Delete a Contact

You can delete a contact in BillingCenter from an external system by using the following method in `ContactAPI`.

BillingCenter prevents you from deleting contacts that are *in use*. A contact is in use if it has any of the following qualities:

- The contact is used by an account.
- The contact is used by a `PolicyPeriod`.
- The contact has any account only roles.
- The contact is referenced from a user (a subobject of `User`).
- The contact is a participant in a reinsurance agreement.
- The contact is a broker in a reinsurance agreement.

If you try to delete a contact that is in use, these APIs throw an exception.

If the account is in use by an account, the exception message mentions which account uses it. If there are multiple accounts that use it, only one account is mentioned in the message.

The `removeContact` method takes the following argument:

- An address book UID (a `String`). This ID is unique and is separate from the public ID. The address book UID corresponds to the external contact management system's native internal ID for this contact.

The `removeContact` method returns nothing.

In addition to the limitations of deletion mentioned earlier, this method also requires that the contact is auto synced to an external address book.

The following Java example demonstrates this API:

```
contactAPI.removeContact("12345");
```

Determine if a Contact Can Be Deleted

You can check whether contacts can be deleted from an external system by using the `isContactDeletable` method. This method requires an `AddressBookUID` as a parameter.

The return value is `true` if it the contact can be deleted or `false` if the contact cannot be deleted.

BillingCenter prevents you from deleting contacts that are *in use*. A contact is in use if it has any of the following qualities:

- The contact is used by an account.
- The contact is used by a `PolicyPeriod`.
- The contact has any account only roles.
- The contact is referenced from a user (a subobject of `User`).
- The contact is a participant in a reinsurance agreement.
- The contact is a broker in a reinsurance agreement.
- The contact is a local-only contact, which means that auto sync is disabled for the contact

The contact can be deleted (returns `true`) unless any of the following are true.

- The contact is a local-only contact, which means that auto sync is disabled for the contact
- The contact is a `UserContact`, which means that it is associated with a BillingCenter user
- The contact is connected to any `PolicyPeriod`, `Account`, or `Producer` object.

The following Java example demonstrates this API:

```
Boolean isDeletableContact = contactAPI.isContactDeletableByPublicID("bc:12345");
```

Update Contacts

You can update a contact from an external system by using the `updateContact` method. It returns nothing.

To add a new contact you must populate and provide a contact as the first argument. Specify the contact in a XSD-defined XML object. The type's fully qualified name is:

```
gw.webservice.contactapi.beanmodel.XmlBackedInstance
```

The `XmlBackedInstance` is a type defined in the `BeanModel.xsd` file. You can find this XSD file in Studio. The `XmlBackedInstance` type in this XSD is basically a XML container for a series of pairs of property name and property value, both as `String` values.

To determine which contact to update, BillingCenter uses that object's address book UID, which is in the `LinkID` property in the `XmlBackedInstance` object. If the contact cannot be found, BillingCenter throws an exception. Finally, BillingCenter uses the other properties in the contact XML object to update the contact entity instance.

This method updates the contact only if automatic synchronization (`autosync`) is enabled for that contact.

The following Java example demonstrates these APIs:

```
gw.webservice.contactapi.beanmodel.XmlBackedInstance contactXML;  
  
// here you would populate fields in the contactXML object, especially the LinkID property,  
// which contains the Address Book UID for this contact  
  
contactAPI.updateContact(contactXML);
```

Merge Contacts

You can merge two contacts from an external system. You need to know the IDs of both contacts, and you must decide which one will survive after the merge.

To merge contacts, use the `mergeContacts` method. This method returns nothing.

Pass the following parameters in this order:

1. The ID of the contact to keep. This parameter is known as the *kept* contact.
2. The ID of the contact to delete. This parameter is known as the *deleted* contact.
3. A transaction ID (a `String`) that uniquely identifies the request from an external system. BillingCenter performs no built-in check with this parameter in the current release. You can modify BillingCenter to use this transaction ID to detect, log, and handle duplicate requests. A request is a duplicate if the transaction ID matches a previous request.

Merging contacts has the following results:

- Fields on the deleted contact are not preserved.
- Account contact (`AccountContact`) objects that reference the deleted contact now reference the kept contact. If both exist on the same account, the kept contact's `AccountContact` property value is used.
- Account contact roles (`AccountContactRole[]`) on a merged `AccountContact` move to the kept contact's account contact. If there are duplicate roles, the kept contact's roles are preserved.
- BillingCenter refreshes the kept contact from the external Contact Management System. BillingCenter calls the contact system plugin (`IContactSystemPlugin`) method called `retrieveContact(String, Bundle)`.
- Policy contact roles (`PolicyContactRole[]`) on a merged `PolicyPeriodContact` move to the kept contact's policy period contact. If there are duplicate roles, the kept contact's roles are preserved.
- Producer contact roles (`ProducerContactRole[]`) on a merged `ProducerContact` move to the kept contact's producer contact. If there are duplicate roles, the kept contact's roles are preserved.
- The deleted contact and any duplicate subobject is retired (deleted). This includes the following: `AccountContact`, `AccountContactRole`, `PolicyPeriodContact`, `PolicyContactRole`, `ProducerContact`, and `ProducerContactRole`.

The following Java example demonstrates these APIs:

```
contactAPI.mergeContacts("pc:uid:1234", "pc:uid:5550", "my-transaction-ID-12345");
```

Handling Rejection and Approval of Pending Changes

An external contact management system might, like ContactManager, support pending changes. *Pending changes* are changes to contacts that are applied in the core application, but require approval in the contact management system before being applied there.

Note: In the base configuration, BillingCenter and PolicyCenter do not use the pending changes feature. All changes to contacts in either of these core applications are applied in the contact management system. BillingCenter and PolicyCenter implement the pending change methods, as required by `ABCClientAPI`, and if a contact management system calls any of these methods, the application throws an exception.

The ClaimCenter implementation of `ContactAPI` has methods that the external contact management system can call to indicate if pending create or pending update operations have been approved or rejected. In each case, the methods pass in a parameter that contains the context of the original change, usually the user, claim, and contact information. This information enables ClaimCenter to notify the user of the results of the operation. If the change is rejected, ClaimCenter creates an activity for the user giving the details of the change that was rejected. If the rejection was for a pending update, ClaimCenter also copies the contact data from the contact management system and attaches it as a note to the activity.

These pending change methods are:

- `pendingCreateRejected` – ClaimCenter creates a pending create rejected activity for the user that created the contact.
- `pendingCreateApproved` – No action is taken. ClaimCenter already has the `AddressBookUID` for the contact.
- `pendingUpdateRejected` – ClaimCenter creates a pending update rejected activity for the user that changed the contact.

- `pendingUpdateApproved` – ClaimCenter updates the contact graph with any new `AddressBookUID` values that were created for any new entities that were created in the update. Additionally, if there was context information sent with the update approval, ClaimCenter synchronizes all contacts that have this `AddressBookUID` with the contact management system.



chapter 20

Multicurrency Integration between BillingCenter and PolicyCenter

BillingCenter and PolicyCenter support multicurrency integration with each other in their default configurations. However, you might need to modify integration code in your configurations of BillingCenter and PolicyCenter to enable the integration appropriately.

This topic includes:

- “Set up Currencies for Multicurrency Integration” on page 405
- “Configure Account Numbers for Multicurrency Accounts in BillingCenter” on page 406
- “” on page 406

See also

- “Multicurrency Integration Between BillingCenter and PolicyCenter” on page 417 in the *Application Guide*
- “Enabling Multicurrency Integration” on page 461 in the *Configuration Guide*

Set up Currencies for Multicurrency Integration

In a multicurrency InsuranceSuite integration between BillingCenter and PolicyCenter, you must set the default application currency the same in both applications. Using the same default application currency in each core application of Guidewire InsuranceSuite always is required. In addition, you must configure each application with the currencies that you want them to share for billing purposes.

For example, In PolicyCenter you allow policies to be settled in U.S. dollar, European Union euro, and Japanese yen. For European policies, you allow assets to be valued in European Union euro and British pound. In PolicyCenter you set up pounds for use as a coverage currency, and you set up dollar, euro, and yen for use as settlement currencies.

In BillingCenter, you also set up U.S. dollar, European Union euro, and Japanese yen as currencies. BillingCenter and PolicyCenter share these currencies for billing and settlement purposes. However, you do not setup British pound as a currency in BillingCenter. In the base configuration of PolicyCenter, each policy can have only one settlement currency. So, PolicyCenter converts pounds to euro before it calculates premiums and other charges on a multicurrency policy. PolicyCenter sends the charges to BillingCenter in euro.

See also

- “Configuring Currencies” on page 113 in the *Globalization Guide*

Configure Account Numbers for Multicurrency Accounts in BillingCenter

In PolicyCenter, for an account with policies in more than one currency, only a single account number is visible. However, in BillingCenter there is a set of affiliated accounts corresponding to each currency used by the original PolicyCenter account. BillingCenter does not generate special account numbers for affiliated accounts. In the base configuration, BillingCenter gives each account a unique and unrelated number, regardless of whether the account is an affiliated currency account or not.

Affiliated Account Numbers and Billing Communication

The insured can receive invoices or other correspondence that references the account number for an affiliated currency specific BillingCenter account. You can configure BillingCenter to assign an account number to affiliated accounts that relates in an obvious way to the original account number.

For example, a policyholder with policies in multiple currencies calls to inquire about an invoice that references an affiliated account number. It will help the billing clerk if the account number for that affiliated account is clearly related to the original account number.

Assign Account Numbers to Affiliated Multicurrency Accounts

When BillingCenter creates an affiliated currency account you can replace the automatically generated account number. Create a custom account number that include the principal account number visible in PolicyCenter. Then assign this custom account number to the affiliated account.

BillingCenter creates affiliated accounts in the `createAccountForCurrency` method in the Billing web service API. You can configure this method to create a specific and meaningful account number for affiliated accounts. Use the `parentAccount` parameter in the `createAccountForCurrency` method to access the principal account number.

For example, to assign a similar account number to BillingCenter affiliated accounts, you can take the principal account number and append the currency of the affiliated account. Make this change in the `createAccountForCurrency` method. After the affiliated account is created, you can insert code similar to the following:

```
account.AccountNumber = parentAccount.AccountNumber + "." + currency.DisplayName
```

See also

- “Policy Administration System Core Web Service APIs (BillingAPI)” on page 85

Custom Batch Processing

BillingCenter lets you implement custom batch processing to supplement the batch processing provided in the base configuration.

This topic includes:

- “Overview of Custom Batch Processing” on page 407
- “Developing Custom Work Queues” on page 410
- “Example Work Queues” on page 417
- “Developing Custom Batch Processes” on page 420
- “Example Batch Processes” on page 424
- “Enabling Custom Batch Processing to Run” on page 427
- “Monitoring Batch Processing” on page 429
- “Periodic Purging of Batch Processing Entities” on page 431

See also

- “Batch Processing” on page 107 in the *System Administration Guide*

Overview of Custom Batch Processing

Custom batch processing that you implement in BillingCenter operates like the batch processing that BillingCenter provides in the base configuration.

This topic includes:

- “Styles of Batch Processing” on page 408
- “Choosing a Style for Custom Batch Processing” on page 408
- “Nightly and Daytime Batch Processing” on page 408
- “Batch Processing Typecodes” on page 409

Styles of Batch Processing

BillingCenter supports two styles of batch processing:

- **Work queue** – A work queue operates on a batch of items in parallel. Work queues run partially on the active batch server and can run on other servers in a BillingCenter clustered environment.

A work queue comprises the following components:

- A processing thread, known as a *writer*, that selects a batch of items to process
- A queue of selected work items
- Processing threads, known as *workers*, that process the items to completion

Work queues are suitable for high volume batch processing that requires items to be processed in parallel to achieve an acceptable throughput rate.

- **Batch process** – A batch process operates on a batch of items sequentially. Batch processes run only on the active batch server in a BillingCenter clustered environment.

Batch processes are suitable for low volume batch processing that achieves an acceptable throughput rate when it processes items sequentially. For example, writers for work queues operate as batch processes because they can select items for a batch and write them to their work queues relatively quickly.

Choosing a Style for Custom Batch Processing

Consider the following factors to decide between developing a custom work queue and developing a custom batch process:

- The volume of items in a typical batch
- The duration of the batch processing window

For business oriented batch processing, such as invoicing or ageing, Guidewire recommends that you develop a custom work queue. Work queues process items in parallel on separate execution threads distributed across multiple servers in a BillingCenter clustered environment.

IMPORTANT Regardless of the volume of items or the duration of the processing window, Guidewire strongly recommends implementing any type of custom batch processing as a custom work queue.

Nightly and Daytime Batch Processing

Specific types of batch processing fall into one of two broad categories:

- **Nightly batch processing** – Processes business transactions accumulated at the end of specific business periods, such as business days, months, quarters, or years
- **Daytime batch processing** – Defers to periodic asynchronous background processing complex transactional processing triggered by user actions

Depending on the category of your custom batch processing, certain implementation details can vary.

Nightly Batch Processing

Nightly batch processing typically processes relatively large batches of work, such as processing premium payments that accumulate during the business day. Each type of nightly batch processing runs once during the night, when users are not active in the application.

Because nightly batch processing typically operates on large batches and has rigid processing windows, develop your nightly batch processing as a custom work queue. Nightly batch processing often requires processing items in parallel to achieve sufficient throughput, and the workloads of nightly batch processing are typically too severe for the batch server.

For nightly batch processing, you typically configure a custom work queue table to segregate the work items of different work queues from each other. Each type of nightly batch processing typically has many worker threads simultaneously accessing the queue for available work items. Using the same work queue table for all types of nightly work queues can degrade processing throughput due to table contention. In addition, segregating work items from different work queues into separate tables can ease recovery of failed work items before the nightly processing window closes.

Nightly batch processing frequently requires chaining so that completion of one type of batch processing starts another type of follow-on batch processing. Regardless of implementation style – work queue or batch process – you most likely must develop custom process completion logic for your type of nightly batch processing.

See also

- “Performing Custom Actions After Batch Processing Completion” on page 118 in the *System Administration Guide*

Daytime Batch Processing

Daytime batch processing typically processes relatively small batches of work, such as reassigning activities escalated by users. Daytime processing runs frequently during the day, while users are active in the application. You might schedule different types of daytime batch processing to run every hour or even every few minutes.

Even though daytime processing typically operates on small batches and lacks rigid processing windows, develop your type of daytime batch processing as a custom work queue. Although daytime batch processing often achieves sufficient throughput by processing items sequentially, its workload on the batch server can slow overall performance of the application. As a work queue, your daytime batch processing can be assigned to a worker thread on a server other than the batch server.

If you develop your daytime batch processing as a custom work queue, you typically can use the standard work queue table for your work items. Different types of daytime batch processing run intermittently and in different bursts of relatively short work. So, table contention between various types of daytime batch processing is often minimal.

Daytime batch processing seldom requires chaining. Completion of one type of daytime batch processing seldom starts another type of follow-on batch processing. Regardless of implementation style, you most likely do not need to develop custom process completion logic for your type of daytime batch processing.

Batch Processing Typecodes

BillingCenter identifies and manages batch processing by typecodes in the `BatchProcessType` typelist. Each type of batch processing, whether implemented as a batch process or a work queue, has a unique typecode in the typelist. Whenever you develop a type of custom batch processing, begin by defining a typecode for it the `BatchProcessType` typelist.

You use the typecode for your type of custom batch processing in the following ways:

- Arguments to some of the methods in your custom work queue or batch process class
- An argument to the maintenance tools command and web service that let you start a batch processing run
- Categorizing how your custom batch process can be run – from the administrative user interface, on a schedule, or from the maintenance tools command line or web service
- A case clause in the Batch Processing Completion plugin for your type of batch processing

Regardless the style of batch processing you implement, first define a new typecode in the `BatchProcessType` typelist for your type of batch processing.

See also

- “Defining the Typecode for Your Custom Work Queue” on page 412

Developing Custom Work Queues

A *work queue* is code that runs without human intervention as a background process on multiple servers to process the units of work in a batch in parallel. Develop a custom work queue for batch processing that requires processing the units of work in a batch in parallel to achieve an acceptable throughput rate.

This topic includes:

- “Custom Work Queue Overview” on page 410
- “Defining the Typecode for Your Custom Work Queue” on page 412
- “Defining the Work Item Type for Your Custom Work Queue” on page 412
- “Creating Your Custom Work Queue Class” on page 413
- “Developing the Writer for Your Custom Work Queue” on page 414
- “Developing the Workers for Your Custom Work Queue” on page 415

See also

- “Example Work Queues” on page 417

Custom Work Queue Overview

A custom work queues comprises the following components:

- **Writer** – The `findTargets` Gosu method on a class that extends `WorkQueueBase` to select the units of work for a batch and writes work items for them in the work queue table
- **Work queue** – An entity type that implements the `WorkItem` delegate, such as the `StandardWorkItem` entity, to establish the database table for the work queue and its work items
- **Worker** – The `processWorkItem` Gosu method on the same class that extends `WorkQueueBase` base class to process units of work identified on the work queue by work items

Starting the writer initiates a run of the type of batch processing that a work queue performs. The batch is complete when the workers exhaust the queue of all work items in the batch, except those they fail to process successfully.

Your Custom Work Queue Class

You implement the code for your writer and your workers as methods on a single Gosu class. You must derive your custom work queue class from the `WorkQueueBase` class. This base class and the work queue framework provide most of the logic for managing the work items in your work queue. You must override only a few methods in the base class to implement the code for your writer and your workers.

Work Queue Writer

You implement the writer for your work queue by overriding the `findTargets` method inherited from the base class. Your code selects the units of work for a batch and returns an iterator for the result set. The work queue framework then uses the iterator to create and insert work items into your work queue. Each work item holds the instance ID of a unit of work in your result set.

For example, your custom work queue sends email about overdue activities to their assignees. In this example, instances of the `Activity` entity type are the units of work for the work queue. In your `findTargets` methods, you query the database for activity instances where the last viewed date of an open activities exceeds 30 days. You return an iterator to the result set, and then the framework creates a work item for each element in the result.

Workers of a Work Queue

You implement the workers of your work queue by overriding the `processWorkItem` method inherited from the base class. The work queue framework calls the method with a work item as the sole parameter. Your code accesses the unit of work identified in the work item and processes it to completion. Upon completion, your code returns from the method, and the work queue framework deletes the work item from the work queue.

For example, the units of work for your work queue are open activities that have not been viewed in the past 30 days. In your `processWorkItem` method, you access the activity instance identified by the work item. Then, you generate and send an email message to the assignee of that activity. After your code sends the email, it returns from the method. The framework then deletes the work item and updates the count of successfully processed work items in the process history for the batch.

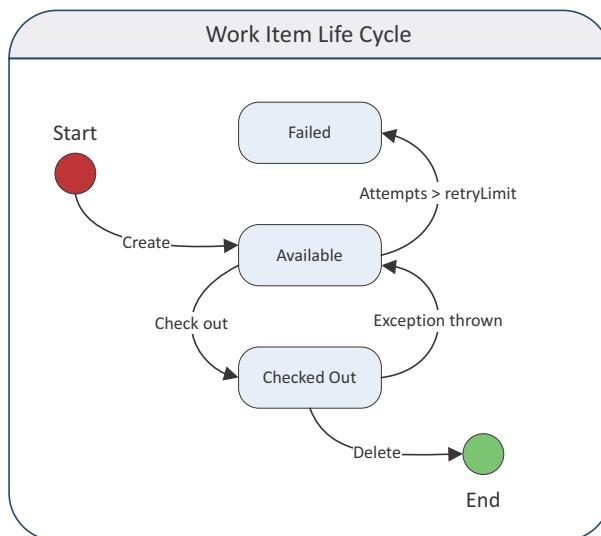
Work Queues and Work Item Entity Types

A work item is an instance of entity type that implements the `WorkItem` delegate, such as `StandardWorkItem`. The entity type provides the database table in which the work items for your custom work queue persist. Work items in the work queue table are accessible from all servers in a BillingCenter cluster, so workers can access them asynchronously, in parallel, and distributed on different servers.

Work items have many properties that the work queue framework uses to manage work items and the process histories of work queue batches. If you use `StandardWorkItem` for your work queue, the only field on a work item that your custom code uses is the `Target` field. The `Target` field holds the ID of a unit of work that your writer selected. The code for your writer and your workers can safely ignore the other properties on work item instances. However, you can define a custom work item type with additional fields for your custom code to use.

Lifecycle of a Work Item

The `Status` field of a work item records the current state of its lifecycle. The work queue framework manages this field and the lifecycle of work items for your writer and workers. The following diagram illustrates the life-cycle.



A work item begins life after the writer selects the units of work for a batch and returns an iterator for the collection. The framework then creates work items that reference the units of work for a batch. The initial state of a work item is **available**. At the time the framework checks out a quota of work items for a worker, the state of the work items becomes **checkedout**. The framework then hands the quota of work items to the worker one at a time. After a worker finishes processing a work item successfully, the framework deletes it from the work queue and updates the statistics in the process history for the batch.

Returning Work Items to the Work Queue

Sometimes a worker cannot finish processing a work item successfully. For example, a network resource may be temporarily unavailable. Or, a concurrent data change exception (CDCE) can occur when two workers simultaneously update common entity data in the domain graphs of two separate units of work. Whenever errors occur, the worker throws an exception to return the work item to the work queue as available again.

Whenever a worker returns a work item to the work queue, BillingCenter increments the `Attempts` property on the work item. If the value of `Attempts` remains below the retry limit for the worker, the state of the work item remains available. The work item is subsequently checked out in a quota for the same or another worker. If the temporary error condition clears, the next worker completes it successfully and BillingCenter removes the work item from the work queue.

Work Items that Fail

Whenever a worker returns a checked out work item and the `Attempts` property exceeds the work item retry limit, the state of the work item becomes failed. Failed work items end their lifecycle in this state. As long as failed work items remain in a work queue, the writer will not select another batch. Someone must take corrective action and remove all failed work items from the work queue before the next batch can run.

See also

- For more information on the lifecycle of work queues, see “Troubleshooting Work Queues” on page 120 in the *System Administration Guide*.

Defining the Typecode for Your Custom Work Queue

Before you begin developing the Gosu code for your custom work queue, define your type of custom work queue. Add a typecode for it in the `BatchProcessType` typelist. The constructor of your custom work queue class requires the typecode as an argument.

1. In the Project window in Studio, navigate to `configuration` → `config` → `Extensions` → `Typelist`, and then open `BatchProcessType.ttx`.
2. Click **Add Typecode** .
3. In the panel of fields on the right, specify the following values.

| Field | Description |
|-------------------|---|
| <code>code</code> | Specify a code that uniquely identifies your type of custom batch processing, for example <code>SendReminderEmail</code> . The code identifies your type of batch processing in configuration files. The code also is a parameter to the maintenance tools command and web service that allows someone to start your type of custom batch processing. |
| <code>name</code> | Specify a human readable identifier for your type of custom batch processing, for example <code>Send reminder email</code> for overdue activities. This name appears for your writer on the Batch Process Info screen. |
| <code>desc</code> | Provide a short description of what your custom batch processing accomplishes, for example, <code>Send reminder email for overdue activities</code> . This description appears for your writer on the Batch Process Info screen. |

See also

- “Categorizing Your Batch Processing Typecode” on page 428

Defining the Work Item Type for Your Custom Work Queue

Before you develop the Gosu code for your custom work queue, decide whether to use the `StandardWorkItem` entity type for your work queue table or a custom work item type. Define a custom work item type for the following reasons:

- Include custom fields on work items not available on the `StandardWorkItem` entity type
- Avoid table contention with other work queues that process typically large batches at the same time

Whenever you define a custom work item type, you must implement the `createWorkItem` method that your custom work queue inherits from `WorkQueueBase`.

If you create a custom work queue, your writer can pass more fields to the workers than a target field. On `StandardWorkItem`, the `Target` field is an object reference to an entity instance, which represents the unit of work for the workers. In a custom work item, you can define as many fields as you want to pass to the workers. Your custom work item itself can be the unit of work.

Guidewire recommends custom work queue types for work queues with typically large batches that potentially run at the same as other work queues with typically large batches. Especially if you develop a custom work queue for nightly batch processing, consider developing a custom work queue type instead of using `StandardWorkItem`. If you create a custom work item type to avoid table contention, you often define a single field for the writer to set, much like the `Target` field on `StandardWorkItem`. By convention, you name the single field with the same name as the entity type, not the generic name `Target`.

1. In the **Project** window in Studio, navigate to **configuration** → **config** → **Extensions** → **Entity**.
2. Right-click **Entity**, and then select **New** → **Entity**.
3. Enter a name and description for your work item type, and then click **OK**.
4. In the drop down list next to , select **implementsEntity**.
5. In the **name** field on the right, select **WorkItem** from the drop down list.
6. For each field on your custom work item type, do the following:
 - a. select the field type in the drop down list next to .

For example, you might select **column**.

 - b. In the panel of the fields on the right, specify the values appropriate to the field type.

For example, you might specify the following values for a column that represents an `Activity` instance as the sole unit of work for your custom work queue.

| Field | Description |
|-------------------|---|
| <code>name</code> | Specify <code>Activity</code> . |
| <code>type</code> | Select <code>softentityreference</code> . |
| <code>null</code> | Select <code>false</code> . |

Whenever you define a custom work item for a work queue, you must implement the `createWorkItem` method to write them to the queue.

See also

- “Writing Custom Work Item Types” on page 415

Creating Your Custom Work Queue Class

After you define a typecode for your custom work queue and possibly create a custom work item type for it, you are ready to create your custom work queue class. This class contains the programming logic for the writer and the workers of your work queue. You must derive your class from `WorkQueueBase`, and you generally must override the following two methods:

- `findTargets` – The logic for the writer, which selects units of work for a batch and returns an iterator for the result set

- `processWorkItem` – The logic for the workers, which operates on a single unit of work selected by the writer

The `WorkQueueBase` provides other methods that you override in certain circumstances, but generally you can develop a successful custom work queue by overriding the two required methods `findTargets` and `processWorkItem`.

WARNING Do not implement multi-threaded programming in custom work queues derived from `WorkQueueBase`.

Work Queue Class Declaration

In the declaration of your custom work queue class, you must include the `extends` clause to derive your work queue class from `WorkQueueBase`. Because the base class is a template class, your class declaration must specify the entity types for the unit of works and for the work items in your work queue.

The following example code declares a custom work queue type that has `Activity` as the target, or unit of work, type. It also declares that `StandardWorkItem` as the work queue type.

```
class MyWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> {
```

Work Queue Class Constructor

You must implement a constructor in your custom work queue class. The constructor that you implement calls the constructor in the `WorkQueueBase` class. Your constructor associates your custom work queue class at runtime with its typecode in the `BatchProcessType` typelist, its work queue item type, and its target type.

The following example code is the constructor for a custom work queue. It associates the class at runtime with its batch process typecode `MyWorkQueue`. The code associates the work queue with the `StandardWorkItem` entity type. So, the work queue shares its work queue table with many other work queues. The code associates the work queue with the `Activity` entity type as its target, or unit of work, type. So, the writer and the workers operate on `Activity` instances.

```
construct () {
    super (typekey.BatchProcessType.TC_MYWORKQUEUE, StandardWorkItem, Activity)
}
```

Do not include any code in the constructor of your custom work queue other than calling the super class constructor.

Developing the Writer for Your Custom Work Queue

You provide the programming logic for a writer thread by overriding the `findTargets` method that your custom work queue class inherits from `WorkQueueBase`.

IMPORTANT Do not operate on any of the units of work in a batch within your writer logic. Otherwise, process history statistics for the batch will be incorrect.

You return a query builder iterator with the results you want as targets for the work items in a batch.

BillingCenter uses the iterator to write the work items to the work queue. The `findTargets` method is a template method for which you specify the target entity type, the same type for which you make a query builder object.

The following example code is a writer for a work queue that operates on `Activity` instances. The query selects activities that have not been viewed for five days or more and returns the iterator.

```
override function findTargets (): Iterator <Activity> {
    // Query the target type and apply conditions: activities not viewed for 5 days or more
    var targetsQuery = Query.make(Activity)
    targetsQuery.compare(Activity#LastViewedDate.PropertyInfo.Name, LessThanOrEquals,
        java.util.Date.Today.addBusinessDays(-5))
```

```
    return targetsQuery.select().iterator()
}
```

Returning an Empty Iterator

Generally, if your writer returns an iterator with items found by the query, BillingCenter creates a `ProcessHistory` instance for the batch run. Sometimes the query in your writer finds no qualifying items. If your writer returns an empty iterator, BillingCenter does not create a process history for that batch processing run.

Writing Custom Work Item Types

If you defined a custom work item type instead of using `StandardWorkItem`, you must override the `createWorkItem` method to write new work items to your custom work queue table. The method has two parameters.

- `target` – An object reference to an entity instance in the iterator returned from the `findTargets` method
- `safeBundle` – A transaction bundle provided by BillingCenter to manage updates to the work queue table

Your method implementation must create a new work item of the custom work item type that you defined. Pass the `safeBundle` parameter in the new work item statement. Then, assign the `target` parameter to the target unit-of-work field that you defined for your custom work item type. If you defined additional fields, assign values to them, too.

The return type for the `createWorkItem` method is any entity type that implements the `WorkItem` delegate. Return your new custom work item type.

The following Gosu example code creates a new custom work item that has a single custom field, an `Activity` instance. The implementation gives the target field in the parameter list the name `activity` to clarify the code. The custom work item type is `MyWorkItem`.

```
override function createWorkItem (activity : Activity, safeBundle : Bundle) : WorkItem {
    var customWorkItem = new myWorkItem(safeBundle)
    customWorkItem.Activity = activity
    return workItem
}
```

Developing the Workers for Your Custom Work Queue

You provide the programming logic for a worker thread by overriding the `processWorkItem` method that your custom work queue class inherits from `WorkQueueBase`. The workers of a work queue are inherently single threaded. Do not attempt to improve the processing rate of individual workers by spawning threads from within your `processWorkItem` method.

The `processWorkItem` has a work item as its single parameter. You access the target, or unit of work, for the worker through the `WorkItem.Target` property. The work item is the type and the target type are the types that you specified in the constructor of your custom work queue class derived from `WorkQueueBase`. At the time BillingCenter calls the `processWorkItem` method, the `Status` field of the work item is set to `checkedout`.

Successful Work Items

When your worker finishes operating on a target instance, return from the `processWorkItem` method. BillingCenter then deletes the work item from the work queue.

The following example code extracts the targeted unit of work, an `Activity` instance, from the work item parameter. Then, the code sends the assigned user an email message.

```
override function processWorkItem (WorkItem : StandardWorkItem): void {
    // Extract the unit of work: an Activity instance
    var activity = extractTarget(WorkItem)
```

```

if (activity.AssignedUser.Contact.EmailAddress1 != null) {
    // Send an email to the user assigned to the Activity.
    mailUtil.sendEmailWithBody(null,
        activity.AssignedUser.Contact,
        null,
        "Activity not viewed for five days",
        "Take a look at activity " + target.Subject + ", due on " + target.TargetDate + ".") // Body:
    return
}

```

In your override of the `processWorkItem` method, specify the same work item entity type that you specified in the constructor of your custom work queue class.

Updating Entity Data

The `processWorkItem` method does not have a bundle to manage database transactions related to targeted units of work. A bundle exists at the time BillingCenter calls your `processWorkItem` method, but that bundle is for updates that BillingCenter makes to the work item. To modify entity data, you must use the `Transaction.RunWithNewBundle` API to create a bundle.

IMPORTANT You must use the `RunWithNewBundle` API to modify entity data with your custom work queue. For more information, see “Running Code in an Entirely New Bundle” on page 351 in the *Gosu Reference Guide*.

The following example code uses the `RunWithNewBundle` transaction method to update the `EscalationDate` on the `Activity` instance from the work items that it processes.

```

uses gw.transaction.Transaction
...
override function processWorkItem (WorkItem : StandardWorkItem): void {
    // Extract the unit of work: an Activity instance
    var activity = extractTarget(WorkItem)

    // Update the activity escalation date
    Transaction.RunWithNewBundle( \ bundle -> {
        activity = bundle.add(activity)           // add the activity to the new bundle
        activity.EscalationDate = java.util.Date.Today } ) // update the escalation date

    return
}

```

Failed Work Items

If your worker code encounters an error while operating on a target instance, throw an exception. Some types of exceptions are thrown automatically, such as a concurrent data change exception (CDCE). They generally resolve themselves quickly and automatically. Other logic errors that your code can detect may require human intervention to resolve. If so, implement a custom exception and throw it whenever your worker code detects the exceptional situation.

BillingCenter catches exceptions thrown by code within the scope of your `processWorkItem` method. Whenever BillingCenter catches an exception, it increments the `Attempts` property on the work item. If the value of the `Attempts` property then exceeds the value of `WorkItemRetryLimit` in `config.xml`, BillingCenter sets the `Status` property to `available`, otherwise it sets to `failed`.

Work Item Retry Interval

BillingCenter makes work items that trigger exceptions available again on the work queue, because many exceptions resolve themselves quickly. For example, a CDCE exception often occurs when two worker threads attempt to update common data related to their two separate units of work. By the time BillingCenter gives a work item that encountered an exception to another worker thread, the exceptional condition often is resolved. The second worker then processes the work item to completion successfully.

Failing Work Items Immediately

You can provide an implementation of the `handleException` method inherited from the base class to augment the actions that BillingCenter takes when code in the `processWorkItem` throws an exception. For example, your worker code might throw a custom exception when it encounters a logic error that cannot resolve itself without human intervention. In your implementation of the `handleException` method, you can check the exception type. If the type is your custom exception, your code sets the `Status` property to `failed` regardless of the number of retry attempts. If the exception is any other type, call the super class.

```
override function handleException (W workItem, java.lang.Throwable exception,
    int consecutiveExceptions): void {
    ...
    // Fail immediately for a custom work queue exception
    if (exception typeis myCustomWorkQueueException) {
        workItem.Status = failed
        return true
    }
    // Fail only for other exceptions if attempts exceeds retry count
    else
        return super (workItem, exception, consecutiveExceptions)
}
```

See also

- To configure how BillingCenter handles work items that trigger exceptions, see “Scheduling Work Queue Writers and Batch Processes” on page 113 in the *System Administration Guide*

Example Work Queues

This topic contains the following example work queues:

- “Simple Example of a Work Queue” on page 417
- “Example Work Queue for Updating Entities” on page 418
- “Example Work Queue with a Custom Work Item Type” on page 419

See also

- “Developing Custom Work Queues” on page 410

Simple Example of a Work Queue

The following Gosu code is an example of a simple custom work queue. It uses the `StandardWorkItem` entity type for its work queue table. Its unit of work is an activity that has not been viewed for five days or more. The process simply sends an email to the user assigned to the activity. The process does not update entity data, so the `processWorkItemMethod` does not use the `runWithNewBundle` API.

```
uses gw.processes.WorkQueueBase
uses gw.api.database.Query
uses gw.api.database.DBFunction
uses java.util.Iterator
uses gw.api.email.EmailUtil

/**
 * An example of batch processing implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> {

    /**
     * Let the base class register this type of custom batch processing by
     * passing the batch processing type, the entity type the work queue item, and
     * the target type for the units of work.
    */
    construct () {
```

```

super (typekey.BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, StandardWorkItem, Activity)
}

< /**
 * Select the units of work for a batch run: activities that have not been
 * viewed for five days or more.
 */
override function findTargets (): Iterator <Activity> {
    // Query the target type and apply conditions: activities not viewed for 5 days or more
    var targetsQuery = Query.make(Activity)
    targetsQuery.compare(Activity#LastViewedDate.PropertyInfo.Name, LessThanOrEquals,
        java.util.Date.Today.addBusinessDays(-5))
}

return targetsQuery.select().iterator()
}

< /**
 * Process a unit of work: an activity not viewed for five days or more
 */
override function processWorkItem (WorkItem : StandardWorkItem): void {
    // Extract an object referencet to the unit of work: an Activity instance
    var activity = extractTarget(WorkItem) // Convert the ID of the target to an object reference
    if (activity.AssignedUser.Contact.EmailAddress1 != null) {
        // Send an email to the user assigned to the activity.
        EmailUtil.sendEmailWithBody(null,
            activity.AssignedUser.Contact, // To:
            null, // From:
            "Activity not viewed for five days", // Subject:
            "See activity " + target.Subject + ", due on " + target.TargetDate + ".") // Body:
    }
    return
}

}

```

Example Work Queue for Updating Entities

The following Gosu code is an example of a custom work queue that updates entity data as part of processing a unit of work. It uses the `StandardWorkItem` entity type for its work queue table. Its unit of work is an activity that has not been viewed for five days or more. The process sends an email to the user assigned to the activity, and then it sets the escalation date on the activity to the current date. The process updates entity data, so the `processWorkItemMethod` uses the `runWithNewBundle` API.

```

uses gw.processes.WorkQueueBase
uses gw.api.database.Query
uses gw.api.database.DBFunction
uses java.util.Iterator
uses gw.transaction.Transaction
uses gw.api.email.EmailUtil

< /**
 * An example of batch processing implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> {

    /**
     * Let the base class register this type of custom batch processing by
     * passing the batch processing type, the entity type the work queue item, and
     * the target type for the units of work.
    */
    construct () {
        super (typekey.BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, StandardWorkItem, Activity)
    }
}

```

```

    /**
     * Select the units of work for a batch run: activities that have not been
     * viewed for five days or more.
     */
    override function findTargets (): Iterator <Activity> {
        // Query the target type and apply conditions: activities not viewed for 5 days or more
        var targetsQuery = Query.make(Activity)
        targetsQuery.compare(Activity#LastViewedDate.PropertyInfo.Name, LessThanOrEquals,
            java.util.Date.Today.addBusinessDays(-5))

        return targetsQuery.select().iterator()
    }

    /**
     * Process a unit of work: an activity not viewed for five days or more
     */
    override function processWorkItem (WorkItem : StandardWorkItem): void {
        // Extract an object referencnt to the unit of work: an Activity instance
        var activity = extractTarget(WorkItem)

        // Send an email to the user assigned to the activity.
        if (activity.AssignedUser.Contact.EmailAddress1 != null) {
            EmailUtil.sendEmailWithBody(null,
                activity.AssignedUser.Contact,
                null, // To:
                null, // From:
                "Activity not viewed for five days",
                "See activity " + target.Subject + ", due on " + target.TargetDate + ".") // Subject:
                // Body:
        }

        // Update the escalation date on the assigned activity
        Transaction.runWithNewBundle( \ bundle -> {
            activity = bundle.add(activity) // add the activity to the new bundle
            activity.EscalationDate = java.util.Date.Today // update the escalation date
        })
    }
}

```

Example Work Queue with a Custom Work Item Type

The following Gosu code is an example of a custom work queue that uses a custom work item type for its work queue table. So, the code provides an implementation of the `createWorkItem` method in addition to the `findTargets` method to add work items for a batch to the work queue.

The unit of work for the work queue is an activity that has not been viewed for five days or more. The process sends an email to the user assigned to the activity, and then it sets the escalation date on the activity to the current date. The work queue uses a custom work item type, so it uses the type its class declaration and constructor.

```

uses gw.processes.WorkQueueBase
uses gw.api.database.Query
uses gw.api.database.DBFunction
uses java.util.Iterator
uses gw.transaction.Transaction
uses gw.api.email.EmailUtil

/**
 * An example of batch processing implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyWorkQueue extends WorkQueueBase <Activity, MyWorkItem> {

    /**
     * Let the base class register this type of custom batch processing by
     * passing the batch processing type, the entity type the work queue item, and
     * the target type for the units of work.
     */
    construct () {
        super (typekey.BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, MyWorkItem, Activity)
    }
}

```

```

/**
 * Select the units of work for a batch run: activities that have not been
 * viewed for five days or more.
 */
override function findTargets (): Iterator <Activity> {
    // Query the target type and apply conditions: activities not viewed for 5 days or more
    var targetsQuery = Query.make(Activity)
    targetsQuery.compare(Activity#LastViewedDate.PropertyInfo.Name, LessThanOrEquals,
        java.util.Date.Today.addBusinessDays(-5))

    return targetsQuery.select().iterator()
}

/**
 * Write a custom work item
 */
override function createWorkItem (activity : Activity, safeBundle : Bundle) : WorkItem {
    var customWorkItem = new MyWorkItem(safeBundle)
    customWorkItem.Activity = activity

    return workItem
}
/**
 * Process a unit of work: an activity not viewed for five days or more
 */
override function processWorkItem (workItem : MyWorkItem): void {
    // Get an object reference to the activity
    var activity = workItem.Activity

    // Send an email to the user assigned to the activity.
    if (activity.AssignedUser.Contact.EmailAddress1 != null) {
        EmailUtil.sendEmailWithBody(null,
            activity.AssignedUser.Contact,
            null, // To:
            "Activity not viewed for five days", // From:
            "See activity " + target.Subject + ", due on " + target.TargetDate + ".") // Subject:
            "See activity " + target.Subject + ", due on " + target.TargetDate + ".") // Body:

    }

    // Update the escalation date on the assigned activity
    Transaction.runWithNewBundle( \ bundle -> {
        activity = bundle.add(activity) // add the activity to the new bundle
        activity.EscalationDate = java.util.Date.Today // update the escalation date
    }
}

}

```

Developing Custom Batch Processes

A *batch process* is code that runs without human intervention as background process on a single server to process the units of work in a batch sequentially. Develop a custom batch process for batch processing that can achieve an acceptable throughput rate by processing the units of work in a batch sequentially on a single thread.

IMPORTANT Regardless of the volume of items or the duration of the processing window, Guidewire strongly recommends implementing any type of custom batch processing as a custom work queue.

This topic includes:

- “Custom Batch Process Overview” on page 421
- “Creating a Custom Batch Process” on page 421

See also

- “Example Batch Processes” on page 424

Custom Batch Process Overview

Custom work queues are Gosu classes that extend the `BatchProcessBase` base class. The `BatchProcessBase` base class includes code that helps BillingCenter manage the batch processing of your custom batch process class.

Note: Custom batch processes are not intended as substitutes for shell scripts or batch files. Do not attempt to automate a batch of system commands by developing custom batch processes.

You extend the `BatchProcessBase` base class by providing your own implementations of the following methods:

- `doWork` – Select the units of work for a batch and process them sequentially on the single execution thread that the `DoWork` method provides.
- `incrementOperationsCompleted` – After completing a unit of work, increment the count of completed items for the process history of the batch.
- `incrementOperationsFailed` – If a unit or work failed to process successfully, increment the count of failed items for the process history of the batch.

Custom batch processes are inherently single threaded. Do not attempt to improve the throughput of units of work by spawning threads from within your `doWork` method. The methods `incrementOperationsCompleted` and `incrementOperationsFailed` are not thread-safe. Entity instances in transactional bundles on separate threads may have problems. If your type of batch processing requires processing units of work in parallel to achieve sufficient throughput, develop a custom work queue instead of a custom batch process.

WARNING Do not implement multi-threaded programming in custom batch processes derived from `BatchProcessBase`.

Creating a Custom Batch Process

To create a custom batch process

1. In Studio, edit the `BatchProcessType` typecode.
 - a. Add an element to represent your own batch process.
 - b. In the typecode editor, for the new typecode, add one or more categories as they apply for your batch process. Use the `BatchProcessTypeUsage` typelist with following values:
 - `UIRunnable` - the process is runnable both from the user interface and from the web service APIs.
 - `APIRunnable` - the process is only runnable from web service APIs.
 - `Schedulable` - the process can be scheduled.
 - `MaintenanceOnly` - only run the process if the system is at maintenance run level.You must add at least one category or else your batch process cannot run.
2. Create a class that extends the `BatchProcessBase` class (`gw.processes.BatchProcessBase`). The only method you must override is the `doWork` method, which takes no arguments. Do your batch processes work in this method. For more information about this class, including some optional methods to override, see “Batch Process Implementation Using the Batch Process Base Class” on page 422.
3. In Studio, in the Plugins editor, register your new plugin for the `IProcessesPlugin` plugin interface. For more information, see “Implementing the Processes Plugin” on page 429.

4. If you want your custom batch process to run regularly on a schedule instead of on demand, add the entry to the XML file `scheduler-config.xml`. For more information, see “Scheduling Work Queue Writers and Batch Processes” on page 113 in the *System Administration Guide*.

Batch Process Implementation Using the Batch Process Base Class

Your custom batch process must extend the `BatchProcessBase` Gosu class. You must override the `dowork` method, which takes no arguments. You can override other methods, but the base class defines meaningful defaults.

The `dowork` method does not have a bundle to track the database transaction. To modify data, you must use the `Transaction.RunWithNewBundle(...)` API to create a bundle.

IMPORTANT To modify entity data in your batch process, you must use the `RunWithNewBundle` API. For more information about creating bundles, see “Running Code in an Entirely New Bundle” on page 351 in the *Gosu Reference Guide*.

Ensure that your main `dowork` method frequently checks the `TerminateRequested` flag. If it is `true`, exit from your code. For example, if you are looping across a database query, exit from the loop. For more information, see “Request Termination” on page 422.

IMPORTANT BillingCenter calls the `requestTermination` method in a different thread from the thread that runs your batch process `dowork` method.

The following subtopics list other property getters or methods that you can call or override.

Check Initial Conditions

BillingCenter instantiates batch process classes at server startup. Later, BillingCenter calls the `checkInitialConditions` method to determine whether to start your batch process. If the method returns `true`, BillingCenter starts your batch process by calling its `dowork` method. If the method returns `false`, initial conditions are not met and BillingCenter does not call the `dowork` method, skipping the batch process for the time being.

The batch process base class always returns `true` from `checkInitialConditions`. Override this method if you want to provide a conditional response. If you override `checkInitialConditions`, be certain your code completes and returns quickly. Do not include long running code, such as queries of the database. The intent of the method is to determine environmental conditions, such as server run level. If you want to check initial conditions of data in the database, perform the query in the `dowork` method.

The batch process base class automatically ensures the server is at the maintenance run level if you configure your batch process typecode with the `MaintenanceOnly` category. You can perform additional checks on the current server run level by overriding the `checkInitialConditions` method.

If you override the `checkInitialConditions` method, forward the call to the superclass before returning. If the superclass returns `false`, you must return `false`. If the superclass returns `true`, then perform your additional checks of environmental conditions and return `true` or `false` appropriately.

Request Termination

If a user clicks the `Stop` button on the Batch Process Info page, this requests the batch process to terminate. BillingCenter calls the batch process `requestTermination` method to terminate a batch process if possible.

Your batch process must shut down any necessary systems and stop your batch process if you receive this message. If you cannot terminate your batch process, return `false` from this method. The batch process base class always returns `false`, which means that the request did not succeed. The base class also sets an internal `TerminateRequested` flag that you can check to see if a terminate request was received.

IMPORTANT BillingCenter calls the `requestTermination` method in a different thread from the thread that runs your batch process `doWork` method.

For typical implementations, use the following pattern:

- Override the `requestTermination` method and have it return `true`. When the user requests termination of the batch process, the application calls your overridden version of the method.
- Ensure that your main `doWork` method frequently checks the `TerminateRequested` flag. In your `doWork` code, exit from your code if that flag is set. For example, if you are looping across a database query, exit from the loop and return.

Override the `requestTermination` method and return `false` if you genuinely cannot terminate the process soon. Be warned that if you do this, you risk the server shutting down before your process completes.

WARNING Although you can return `false` from `requestTermination`, Guidewire strongly recommends you design your batch process so that you actually can terminate the action. It is critical to understand that returning either value does **not** prevent the application from shutting down or reducing run level. BillingCenter delays shutdown or change in run level for a period of time. However, eventually the application shuts down or reduces run level independent of this batch process setting. For maximum reliability and data integrity, design your code to frequently check and respect the `TerminateRequested` property.

BillingCenter writes a line to the system log to indicate whether the batch process says it could terminate. In other words, the log line includes the result of your `requestTermination` method.

If your batch process can only run one instance at a time, returning `true` does not remove the batch process from the internal table of running batch processes. This means that another instance cannot run until the previous one completes.

Exclusive

The property getter for the `Exclusive` property for a batch process determines whether another instance of this batch process can start while this process is running. The base class implementation always returns `true`. Override this method if you need to customize this behavior. This value does not affect whether other batch process classes can run. It only affects the current batch process class.

For maximum performance, be sure to set this `false` if possible. For example, if your batch process takes arguments in its constructor, it might be specific to one entity such as only a single `Account` entity. If you want to permit multiple instances of your batch process to run in parallel, you must ensure your batch process class implementation returns `false`. For example,

```
override property get Exclusive() : boolean {  
    return false  
}
```

Note: For Java implementations, you must implement this property getter as the method `isExclusive`, which takes no arguments.

Description

The `getDescription` method gets the batch type's description. The base class gets this string from the batch type typecode description. Override this method if you need to customize this behavior.

Detail Status

The property getter for the `DetailStatus` property gets the batch type's detailed status. The base class defines a default simple implementation. Override the default property getter to provide more useful detail information about the status of your batch process for the Administration user interface. The details might be important if your class experiences any error conditions.

Note: For Java implementations, you must implement this property getter as the method `getDetailStatus`, which takes no arguments.

Progress Handling

The `Progress` property in the batch process dynamically returns the progress of the batch process. The base class returns text in the form "x of y" where x is the amount of work completed and y is the total amount of work. If y is unknown, returns just "x". These values are determined from the `OperationsExpected` and `OperationsCompleted` properties. From Java, this is the `getProgress` method.

The following list includes related properties and methods you can use that the base class implements:

- `OperationsExpected` property - a counter for how many operations are expected, as an `int` value. From Java this counter is the `getOperationsExpected` method.
- `OperationsCompleted` property - a counter for how many operations are complete, as an `int` value. From Java this counter is the `getOperationsCompleted` method.
- `incrementOperationsCompleted` method - this no-argument method increments an internal counter for how many operations completed. For example, suppose your batch process iterates across entities of some type with special conditions and specially handle each found item. For each entity you modify, call this method once to track the progress of this batch process. The user interface can display this number or debugging code can track this number to see the progress. This method returns the current number of operations completed. For each entity for which you have an error condition, call this method once to track the progress of this batch process.
- `OperationsFailed` property - an internal counter for the number of operations that failed. From Java this counter is the `getOperationsFailed` method.
- `incrementOperationsFailed` method - this method increments an internal counter for how many operations failed. For example, suppose your batch process iterates across entities of some type with special conditions and specially handle each found item. For each entity for which you have an error condition, call this method once to track the progress of this batch process. The user interface can display this number or debugging code can track this number to see the progress. This method returns the current number of operations failed. You must also call the `incrementOperationsCompleted` method.

IMPORTANT For any operations that fail, call both the `incrementOperationsFailed` method and the `incrementOperationsCompleted` method.

-
- `Finished` property - return a Boolean value to indicate whether the process completed. Completion says nothing about the errors if any. From Java, this is the `isFinished` method.

Type

The base class maintains a `Type` property, which contains the batch process type, as a `BatchProcessType` type-code. From Java this property is the `getType` method.

Example Batch Processes

This topic contains the following example batch processes:

- “Example Batch Process for a Background Task” on page 425

- “Example Batch Process for Unit of Work Processing” on page 425

See also

- “Developing Custom Batch Processes” on page 420

Example Batch Process for a Background Task

The following Gosu code is an example of a custom batch process that operates as a background task instead of one that operates on a batch of units of work. Its process history does not track the number of items that processed successfully or that failed, because it has no formal units of work.

The following Gosu batch process purges old workflows. It takes one parameter that indicates the number of days for successful processes. Pass this parameter in the constructor to this batch process class. In other words, your implementation of `IProcessesPlugin` must pass this parameter such as the new `MyClass(myParameter)` when it instantiates the batch process. If the parameter is missing or `null`, it uses a default system setting.

```
package gw.processes

uses gw.processes.BatchProcessBase
uses java.lang.Integer
uses gw.api.system.PLConfigParameters
uses gw.api.admin.WorkflowUtil

class PurgeWorkflows extends BatchProcessBase
{
    var _succDays = PLConfigParameters.WorkflowPurgeDaysOld.Value

    construct() {
        this(null)
    }

    construct(arguments : Object[]) {
        super("PurgeWorkflows")
        if (arguments != null) {
            _succDays = arguments[0] != null ? (arguments[0] as Integer) : _succDays
        }
    }

    override function doWork() : void {
        WorkflowUtil.deleteOldWorkflowsFromDatabase( _succDays )
    }
}
```

IMPORTANT You must use the `runWithNewBundle` API if want to modify entity data in your custom batch process. For more information, see “Running Code in an Entirely New Bundle” on page 351 in the *Gosu Reference Guide*.

Example Batch Process for Unit of Work Processing

The following Gosu code is an example of a custom batch process as a type of batch processing that processes units of work rather than operating as background task. Its process history tracks the number of items that processed successfully or that failed. Its units of work are urgent activities.

The following Gosu batch process implements a notification scheme such that any urgent activities must be handled within 30 minutes. If it is not handled within that time range, the batch process notifies a supervisor. If still not resolved in 60 minutes, it sends a message further up the supervisor chain.

If there are no qualified activities, it returns false so that it will not create a process history. If there are items to handle, it increments the count. The application uses this count to display batch process status “n of t” or a progress bar. If there are no contact email addresses, the task fails and the application flags it as a failure.

This example checks `TerminateRequested` to terminate the loop if the user or the application requested to terminate the process.

In this Gosu example, it does not actually send the email. Instead it prints to the console. You can change this to use the real email APIs if desired.

```

package sample.processes
uses gw.processes.BatchProcessBase
uses java.util.Map
uses gw.api.util.DateUtil
uses java.lang.StringBuilder
uses java.util.HashMap
uses gw.api.profiler.Profiler

class TestBatch extends BatchProcessBase
{
    static var tag = new gw.api.profiler.ProfilerTag("TestBatchTag1","A sample tag")
    var work : ActivityQuery

    construct() {
        super( "TestBatch")
    }

    override function requestTermination() :Boolean {
        super.requestTermination() // set the TerminationRequested flag
        return true // return true to signal that we will attempt to terminate in our doWork method
    }

    override function doWork() : void { // no bundle
        var frame = Profiler.push(tag);
        try {
            work = find(a in Activity where a.Priority == "urgent" and a.Status == "open"
                and a.CreateTime < DateUtil.currentDate().addMinutes( -30 ) )
            OperationsExpected = work.getCount()
            var map = new HashMap<Contact, StringBuilder>()
            for (activity in work) {
                if (TerminateRequested) {
                    return;
                }
                incrementOperationsCompleted()
                var haveContact = false
                var msgFragment = constructFragment(activity)
                haveContact = addFragmentToUser(map, activity.AssignedUser, msgFragment) or haveContact
                var group = activity.AssignedGroup
                haveContact = addFragmentToUser(map, group.Supervisor, msgFragment) or haveContact
                if (activity.CreateTime < DateUtil.currentDate().addMinutes( -60 )) {
                    while (group != null) {
                        group = group.Parent
                        haveContact = addFragmentToUser(map, group.Supervisor, msgFragment) or haveContact
                    }
                }
                if (!haveContact) {
                    incrementOperationsFailed()
                    addFragmentToUser(map, User.util.UnrestrictedUser, msgFragment)
                }
            }
            if (not TerminateRequested) {
                for (addressee in map.Keys) {
                    sendMail(addressee, "Urgent activities still open", map.get(addressee).toString())
                }
            }
        }
        finally {
            Profiler.pop(frame)
        }
    }

    private function constructFragment(activity : Activity) : String {
        return formatAsURL(activity) + "\n\t"
            + " Subject: " + activity.Subject
            + " AssignedTo: " + activity.AssignedUser
            + " Group: " + activity.AssignedGroup
            + " Supervisor: " + activity.AssignedGroup.Supervisor
            + "\n\t" + activity.Description
    }

    private function formatAsURL(activity : Activity) : String {
        return "http://localhost:8080/bc/Activity.go(${activity.id})"

        // TODO: you must ADD A PCF ENTRYPPOINT THAT CORRESPONDS TO THIS URL TO DISPLAY THE ACTIVITY.
    }
}

```

```

private function addFragmentToUser(map : Map<String, StringBuilder>, user : User,
    msgFragment : String) : boolean {
    if (user != null) {
        var email = user.Contact.EmailAddress1
        if (email != null and email.trim().length > 0) {
            var sb = map.get(email)
            if (sb == null) {
                sb = new StringBuilder()
                map.put(email, sb)
            }
            sb.append(msgFragment)
            return true
        }
    }
    return false;
}

private function sendMail(contact : Contact, subject : String, body : String) {
    var email = new Email()
    email.Subject = subject
    email.Body = "<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">\n" +
        "<html>\n" +
        "  <head>\n" +
        "    <meta http-equiv=\"content-type\"\n" +
        "      content=\"text/html; charset=UTF-8\">\n" +
        "    <title>${subject}</title>\n" +
        "  </head>\n" +
        "  <body>\n" +
        "    <table>\n" +
        "      <tr><th>Subject</th><th>User</th><th>Group</td><th>Supervisor</th></tr>" +
        body +
        "    </table>" +
        "  </body>" +
        "</html>"
    email.addToRecipient(new EmailContact(contact))
    EmailUtil.sendEmailWithBody(null, email);
}
}

```

To use this entry point, use the following PCF entry point in the file `BillingCenter/modules/configuration/pcf/EntryPoints/Activity.pcf`:

```

<PCF>
<EntryPoint authenticationRequired="true" id="Activity" location="ActivityForward(actvtIdNum)">
<EntryPointParameter locationParam="actvtIdNum" type="int"/>
</EntryPoint>
</PCF>

```

Also include the following `ActivityForward` PCF file as the `ActivityForward.pcf` in each place in the PCF hierarchy that you would like it:

```

<PCF>
<Forward id="ActivityForward">
<LocationEntryPoint signature="ActivityForward(actvtIdNum : int)" />
<Variable name="actvtIdNum" type="int"/>
<Variable
    initialValue="find(a in Activity where a.ID == new Key(Activity, actvtIdNum))"
    name="actvt"
    type="Activity"/>
<ForwardCondition action="AccountForward.go(actvt.Account);>
    ActivityDetailWorksheet.goInWorkspace(actvt)>
</Forward>
</PCF>

```

IMPORTANT You must use the `runWithNewBundle` API if want to modify entity data in your custom batch process. For more information, see “Running Code in an Entirely New Bundle” on page 351 in the *Gosu Reference Guide*.

Enabling Custom Batch Processing to Run

Before your type of custom batch processing can run, you must perform the following procedures:

- “Categorizing Your Batch Processing Typecode” on page 428
- “Updating the Work Queue Configuration” on page 428
- “Implementing the Processes Plugin” on page 429

The procedures to perform depend on whether you implemented a custom work queue or a custom batch process.

See also

- To learn how to run your type of custom batch processing on a schedule or on demand, see “Batch Processing” on page 107 in the *System Administration Guide*

Categorizing Your Batch Processing Typecode

For your type of custom batch processing to run, you must associate its typecode with appropriate categories in the `BatchProcessType` typelist. You must categorize the typecode whether you implemented a custom work queue or a custom batch process. You must add at least one category or your type of custom batch processing cannot run.

The runnable categories are:

- `UIRunnable` - The process is runnable both from the user interface and from the web service APIs.
- `APIRunnable` - The process is runnable only from web service APIs.
- `Schedulable` - The process can be scheduled.
- `MaintenanceOnly` - Only run the process if the system is at maintenance run level.

The **Work Queue Info** screen shows your custom work queue, regardless how you categorize your batch processing type code. Categorizing a batch processing type code affects only whether the **Batch Process Info** screen displays custom batch processes and writers for work queues and their runnable characteristics.

See also

- To learn how to add categories to a typecode, see “Dynamic Filters” on page 260 in the *Configuration Guide*.

Updating the Work Queue Configuration

BillingCenter instantiates custom work queues that derive from `WorkQueueBase` at server startup. For BillingCenter to instantiate your custom work queue, you must add a `work-queue` element for it in the `workqueue-config.xml` file.

The `work-queue` element in `workqueue-config.xml` has the following syntax.

```
<work-queue workQueueClass="string" progressInterval="decimal">
  <worker instances="integer" throttleInterval="decimal"
    env="string" server="string"/>
  ...
  <worker instances="integer" throttleInterval="decimal"
    env="string" server="string"/>
</work-queue>
```

The `work-queue` element identifies your custom work queue to BillingCenter and the batch processing framework. Specify the fully qualified class name of your custom work queue Gosu class for the `workQueueClass` attribute.

Each `worker` element defines the worker threads for a server. To distribute workers among multiple servers, add multiple `worker` elements. You must include one `worker` element with the `instances` attribute set to 1 or greater for your custom work queue to operate.

See also

- For complete information, see “Scheduling Work Queue Writers and Batch Processes” on page 113 in the *System Administration Guide*.

Implementing the Processes Plugin

BillingCenter instantiates custom batch processes that derive from `BatchProcessBass` at server startup. For BillingCenter to instantiate your custom batch process, you must implement the Processes plugin (`IProcessesPlugin`). The `IProcessesPlugin` plugin relies on the typecode in `BatchProcessType` typelist for your type of batch processing.

For each typecode in the `BatchProcessType` that represents a custom batch process, BillingCenter calls the `createBatchProcess` method of the Processes plugin. The method takes a typecode from the `BatchProcessType` as a parameter. Implement the `createBatchProcess` method with a switch statement and a case clause for each type of custom batch process that you develop.

The following example code demonstrates how to implement the `IProcessesPlugin` plugin. The example instantiates two custom batch process.

```
uses gw.plugin.processing.IProcessesPlugin
uses gw.processes.BatchProcess
@Export

class ProcessesPlugin implements IProcessesPlugin {

    construct() {
    }

    override function createBatchProcess(type : BatchProcessType, arguments : Object[]) : BatchProcess {
        switch(type) {
            case BatchProcessType.TC_CLAIMHEALTHCALC:
                return new ClaimHealthCalculatorBatch();

            case BatchProcessType.TC_PURGEMESSAGEHISTORY:
                return new PurgeMessageHistory(arguments);

            default:
                return null
        }
    }
}
```

Monitoring Batch Processing

Although batch processing runs without human intervention as a background task, people must manage them and monitor their progress.

This topic describes various BillingCenter features for monitoring default custom batch processing:

- “The Work Queue Info Page” on page 429
- “The Batch Process Info Page” on page 430
- “Monitoring for Batch Processing Completion” on page 430
- “Maintenance Tools” on page 430
- “Process History” on page 430

The Work Queue Info Page

System administrators use the `Work Queue Info` page to control and view information about PolicyCenter and custom work queues.

See also

- “Work Queue Info” on page 157 in the *System Administration Guide*

The Batch Process Info Page

System administrators Use the **Batch Process Info** page to start and view information about PolicyCenter and custom batch processes, including writer processes for work queues.

See also

- “Batch Process Info” on page 156 in the *System Administration Guide*

Monitoring for Batch Processing Completion

Nightly batch processing frequently requires chaining, so completion of one type of batch processing starts another type of follow-on batch processing. Regardless of implementation style – work queue or batch process – you most likely must develop custom process completion logic for your type of nightly batch processing.

Characteristics of a completed run of batch processing vary depending on the implementation style:

- **Work queues** – Completed if no work items for a batch remain in the work queue, other than work items that failed
- **Batch processes** – Completed if the batch process stopped and its process history is available

BillingCenter provides Process Completion Monitor batch processing and the `IBatchCompletedNotification` plugin to permit custom code to react to the completion of any type of batch processing.

The `IBatchCompletedNotification` plugin has a `completed` method that you override to perform specific actions if a work queue or batch process completed a batch of work. The parameters of the `completed` method are the `ProcessHistory` and the number of failed items. In the `completed` method, add a case clause for your type custom batch processing by specifying its typecode from the `BatchProcessType` typelist.

See also

- “Performing Custom Actions After Batch Processing Completion” on page 118 in the *System Administration Guide*

Maintenance Tools

You can start, terminate, or get the status of batch processes, including writers for work queues, by using the `maintenance_tools` command or the `MaintenanceToolsAPI` web service. The web service provides additional functions not available with the command, such as accessing and modifying configuration properties for work queues and notifying workers of work on the queue.

See also

- “Maintenance Tools Web Service” on page 128
- “Maintenance Tools Command” on page 188 in the *System Administration Guide*

Process History

BillingCenter creates a process history instance for every run of a work queue or batch process. You can download completed process histories from the `Work Queue Info` and `Batch Process Info` pages and in the administrative user interface.

| Batch Processing Style | Download Format |
|------------------------|-----------------|
| Work Queue | CSV |
| Batch Process | HTML |

Periodic Purging of Batch Processing Entities

Entities related to batch processing, such as process history and work items, accumulate after batch processing runs have been completed. Outdated entities for completed batches must be purged periodically to avoid slowing performance of all types of batch processing.

BillingCenter provides the following batch processes to purge outdated entities related to batch processing:

- **Process History Purge** – Purges batch process history data from the process history table, the source for download history on the [Batch Process Info](#) page
- **Work Queue Instrumentation Purge** – Purges instrumentation data for work queues, the source for downloadable history on the [Work Queue info](#) page
- **Work Item Set Purge** – Purges work item sets that are more than a specified number of days old
- **Purge Failed Work Items** – Purges failed work items from all work queues

See also

- For more information on the purge processes, see “List of Work Queues and Batch Processes” on page 120 in the [System Administration Guide](#)

Servlets

You can define simple web servlets for your BillingCenter application using the `@Servlet` annotation.

Implementing Servlets

You can define simple web servlets inside your BillingCenter application. You can define extremely simple HTTP entry points to custom code using this approach. These are separate from web services that use the SOAP protocol. These are separate also from the Guidewire PCF entrypoints feature.

Use this technique to define arbitrary Gosu code that a user or tool can call from a configurable URL. You can define a Gosu block that can determine from the URL whether your servlet owns the HTTP request.

Servlets provide no built-in object serialization or deserialization, such as is done in the SOAP protocol.

The BillingCenter servlet implementation uses standard Java classes in the package `javax.servlet.http` to define the servlet request and response. The base class for your servlet must extend `javax.servlet.http.HttpServlet` directly, or extend a subclass of it.

Extending the base class `javax.servlet.http.HttpServlet` provides no inherent security for the servlet. By default users can trigger servlet code without authenticating. This is dangerous in a production system. BillingCenter includes a utility class called `gw.servlet.ServletUtils` that you can use in your servlet to require authentication.

WARNING You must add your own authentication system for servlets to protect information and data integrity. Carefully read “[Implementing Servlet Authentication](#)” on page 435. If you have any questions about server security, contact Guidewire Customer Support.

Creating a Basic Servlet

To create a basic servlet

1. Write a Gosu class that extends the class `javax.servlet.http.HttpServlet`.

2. Register the servlet class in the file:

`BillingCenter/configuration/config/servlets.xml`

Add one `<servlet>` element that references the fully qualified name of your class, for example:

```
<servlets xmlns="http://guidewire.com/servlet">
  <servlet class="com.example.servlets.MyServletClass"/>
</servlets>
```

3. Add the `@Servlet` annotation on the line before your class definition.

You must pass arguments to the constructors to specify which URLs your servlet handles. There are multiple versions of the constructors for different use cases.

You provide a search pattern to test against the *servlet query path*. The servlet query path is every character after the computer name, the port, the web application name, and the word "/service". In other words, your servlet gets the servlet URL substring identified as *YOUR_SERVLET_STRING* in the following URL syntax:

- `@Servlet(pathPattern : String)`

Use this annotation constructor syntax for high-performance pattern matching, though with less flexibility than full regular expressions.

This annotation constructor declares the servlet URL pattern matching to use Apache

`org.apache.commons.io.FilenameUtils` wildcard syntax. Apache `FilenameUtils` syntax supports Unix and Windows filenames with limited wildcard support for ? (single character match) and * (multiple character match) similar to DOS filename syntax. The syntax also supports the Unix meaning of the ~ symbol. Matches are always case sensitive.

- `@Servlet(pathMatcher : block(path : String) : boolean)`

Use this annotation constructor syntax for highly flexible pattern matching.

You pass a Gosu block that can do arbitrary matching on the servlet query path. Performance depends greatly on what you do in your block. As a parameter in parentheses for the annotation, pass a Gosu block that takes a URL String. Write the block such that the block returns `true` if and only if the user URL matches what your servlet handles.

If you only use the `startsWith` method of the `String` argument, for example

`path.startsWith("myprefix")`, the servlet URL checking code is high performance.

You could do more flexible pattern matching with other APIs, such as full regular expressions. Using regular expressions lowers performance for high volumes of servlet calls. The following example uses regular expressions with the `matches` method on the `String` type:

```
@Servlet(\ path : String ->path.matches("/test(/.*)?"))
```

This example servlet responds to URLs that start with the text "/test" in the servlet query path, and optionally a slash followed by other text.

4. Override the `doGet` method to do your actual work. Your `doGet` method takes a servlet request object (`HttpServletRequest`) and a servlet response object (`HttpServletResponse`).

5. In your `doGet` method, determine what work to do using the servlet request object.

WARNING You must add your own authentication system for servlets to protect information and data integrity. Carefully read "Implementing Servlet Authentication" on page 435. If you have questions about server security, contact Guidewire Customer Support.

Important properties on the request object include:

- `RequestURI` – Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
- `RequestURL` – Reconstructs the URL the client used to make the request.
- `QueryString` – Returns the query string that is contained in the request URL after the path.
- `PathInfo` – Returns any extra path information associated with the URL the client sent when it made this request.

- **Headers** – Returns all the values of the specified request header as an Enumeration of String objects.

For full documentation on this class, refer to these Sun Javadoc pages:

<http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html>
<http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html>

6. In your doGet method, write an HTTP response using the servlet response object. For example, the following simple response sets the content MIME type and the status of the response (OK):

```
resp.ContentType = "text/plain"
resp.setStatus(HttpServletResponse.SC_OK)
```

To write output text or binary data, use the `Writer` property of the response object. For example:

```
resp.getWriter.append("hello world output")
```

For example, the following simple Gosu servlet works although it provides no authentication:

```
package mycompany.test

uses gw.servlet.Servlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpServlet

@Servlet( \ path : String ->path.matches("/test(/.*)?"))
class TestingServlet extends HttpServlet {

    override function doGet(req: HttpServletRequest, response: HttpServletResponse) {

        // ** SECURITY WARNING - FOR REAL PRODUCTION CODE, ADD AUTHENTICATION CHECKS HERE!

        // Trivial servlet response to test that the servlet responds
        response.setContentType = "text/plain"
        response.setStatus(HttpServletResponse.SC_OK)
        response.getWriter.append("I am the page " + req.PathInfo)
    }
}
```

Add an element in the `servlets.xml` file for your new servlet class. Run the QuickStart server at the command prompt: `gwbc dev-start` and test the servlet at the URL:

<http://localhost:PORT/bc/service/test>

Change the port number as appropriate for your application.

For a production servlet, you must add authentication. See “[Implementing Servlet Authentication](#)” on page 435.

Implementing Servlet Authentication

Extending the base class `javax.servlet.http.HttpServlet` provides no inherent security for your servlet. By default, anyone can trigger servlet code without authenticating. This is dangerous in a production system, generally speaking. BillingCenter includes a utility class called `gw.servlet.ServletUtils` that you can use in your servlet to require authentication.

WARNING You must add your own authentication for your servlet to protect information and data integrity. If you have any questions about server security, contact Guidewire Customer Support.

There are three methods in the `ServletUtils` class, each of which corresponds to a different source of authentication credentials. The following table summarizes each `ServletUtils` method. In all cases, the first argument is a standard Java `HttpServletRequest` object, which is an argument to your main servlet method `doWork`.

| Source of credentials | ServletUtils method name | Description | Method arguments |
|-----------------------------------|--|---|---|
| Existing BillingCenter session | <code>getAuthenticatedUser</code> | <p>If this servlet shares an application context with a running Guidewire application, there may be an active session token. If a user is currently logged into BillingCenter, this method returns the associated <code>User</code> object.</p> <p>Always check the return value. The method returns <code>null</code> if authentication failed. For example:</p> <ul style="list-style-type: none"> • there is no active authenticated session with correct credentials • the user exited the application • the session ID is not stored on the client • the session ServiceToken timeout has expired | <ul style="list-style-type: none"> • <code>HttpServletRequest</code> object • A Boolean value that specifies whether to update the date and time of the session |
| HTTP Basic authentication headers | <code>getBasicAuthenticatedUser</code> | <p>Even if there is no active session, you can use HTTP Basic authentication. This method gets the appropriate HTTP headers for name and password and attempts to authenticate. You can use this type of authentication style even if there is an active session. This method forces creation of a new session. The method gets the headers to find the user name and password and returns the associated <code>User</code> object.</p> <p>Always check the return value. The method returns <code>null</code> if authentication failed.</p> <p>For login problems, this method might throw the exception <code>gw.api.webservice.exception.LoginException</code>.</p> | <ul style="list-style-type: none"> • <code>HttpServletRequest</code> object |
| Arbitrary user/ login name pairs | <code>login</code> | <p>Use the <code>login</code> method to pass an arbitrary user and password as <code>String</code> values and authenticate with BillingCenter. For example, you might use a corporate implementation of single-sign-on (SSO) authentication that stores information in HTTP headers other than the HTTP Basic headers. You can get the username and password and call this method. This method forces creation of a new session.</p> <p>Always check the return value. The method returns <code>null</code> if authentication failed.</p> <p>For login problems, this method might throw the exception <code>gw.api.webservice.exception.LoginException</code>.</p> | <ul style="list-style-type: none"> • <code>HttpServletRequest</code> object • Username as a <code>String</code> • Password as a <code>String</code> |

You can combine the use of multiple methods of `ServletUtils` in your code.

For example, a typical design pattern is to first call the `getAuthenticatedUser` method to test whether there is an existing session token that represents valid credentials. If the `getAuthenticatedUser` method returns `null`, attempt to use HTTP Basic authentication by calling the method `getBasicAuthenticatedUser`.

The following code demonstrates this technique in the doGet method and calls to a separate method to do the main work of the servlet.

```
package mycompany.test

uses gw.servlet.Servlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpServlet

@Servlet( \ path : String ->path.matches("/test(/.*)?""))
class TestingServlet extends HttpServlet {

    override function doGet(req: HttpServletRequest, response: HttpServletResponse) {
        //print("Beginning call to doGet()...")

        // SESSION AUTH : Get user from session if the client is already signed in.
        var user = gw.servlet.ServletUtils.getAuthenticatedUser(req, true)
        //print("Session user result = " + user?.DisplayName)

        // HTTP BASIC AUTH : If the session user cannot be authenticated, try HTTP Basic
        if (user == null)
            try {
                user = gw.servlet.ServletUtils.getBasicAuthenticatedUser(req)
                //print("HTTP Basic user result = " + user?.DisplayName)
            }
            catch (e) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
                    "Unauthorized. HTTP Basic authentication error.")
                return // Be sure to RETURN early because authentication failed!
            }

        if (user == null) {
            response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
                "Unauthorized. No valid user with session context or HTTP Basic.")
            return // Be sure to RETURN early because authentication failed!
        }

        // IMPORTANT: Execution reaches here only if a user succeeds with authentication.
        // Insert main servlet code here before end of function, which ends servlet request
        doMain(req, response, user )
    }

    // this method is called by our servlet AFTER successful authentication
    private function doMain(req: HttpServletRequest, response: HttpServletResponse, user : User) {
        assert(user != null)

        var responseText = "REQUEST SUCCEEDED\n"+
            "req.RequestURI: '${req.RequestURI}'\n" +
            "req.PathInfo: '${req.PathInfo}'\n" +
            "req.RequestURI: '${req.RequestURI}'\n" +
            "authenticated user name: '${user.DisplayName}'\n"

        // debugging in the console
        //print(responseText)

        // for output response
        response.ContentType = "text/plain"
        response.setStatus(HttpServletResponse.SC_OK)
        response.getWriter.append(responseText)
    }
}
```

To test session authentication servlet code

1. In Studio, create the `mycompany.test.TestingServlet` as shown earlier in this topic.
2. Register this servlet in `servlets.xml`.
3. Run the QuickStart server at the command prompt: `gwbc dev-start`
However, do not yet log into the BillingCenter application.
4. Test the servlet with no authentication by going to the URL:
`http://localhost:PORT/bc/service/test`

You see a message:

```
HTTP ERROR 401  
Problem accessing /bc/service/test. Reason:  
Unauthorized. No valid user with session context or HTTP Basic.
```

5. Log into the BillingCenter application with valid credentials.

6. Test the servlet with no authentication by going to the URL:

```
http://localhost:PORT/bc/service/test
```

You see a message:

```
REQUEST SUCCEEDED  
req.RequestURI: '/cc/service/test'  
req.PathInfo: '/test'  
req.RequestURI: '/cc/service/test'  
authenticated user name: 'Super User'
```

For testing of the HTTP Basic authentication, sign out of the BillingCenter application to remove the session context. Next, re-test your servlet and use a tool that supports adding HTTP headers for HTTP Basic authentication.

Alternative APIs for Authentication

Guidewire recommends using the `ServletUtils` APIs for managing authentication. See “[Implementing Servlet Authentication](#)” on page 435. If you use `ServletUtils`, you can use the session key if available and if not you can use HTTP Basic authentication headers or custom headers.

An alternative approach for authentication is to use the legacy class `AbstractGWAAuthServlet` to translate the security headers in the request and authenticate with the Guidewire server. There is a subclass called `AbstractBasicAuthenticationServlet`, which authenticates using HTTP Basic authentication. You can view the source code to both classes in Studio. These classes are provided primarily for legacy use because they do not support using more than one type of authentication at run time.

Abstract Guidewire Authentication Servlet Class

To use the session key created from a Guidewire application that shares the same application context, you can write your servlet to extend the class `AbstractGWAAuthServlet`. You must override the following methods:

- `doGet` – Your main task is to override the `doGet` method to do your main work. BillingCenter already authenticates the session key if it exists before calling your `doWork` method. For more information about the parameters in this method, see “[Creating a Basic Servlet](#)” on page 433.
- `authenticate` – Create and return a session ID
- `storeToken` – You can store the session token in this method, but you can also leave your method implementation empty.
- `invalidAuthentication` – Return a response for invalid authentication. For example:

```
override function invalidAuthentication( req: HttpServletRequest,  
                                      resp: HttpServletResponse ) : void {  
    resp.setHeader( "WWW-Authenticate", "Basic realm=\"Secure Area\""  
    resp.setStatus( HttpServletResponse.SC_UNAUTHORIZED )  
}
```

Abstract HTTP Basic Authentication Servlet Class

The `AbstractBasicAuthenticationServlet` class extends `AbstractGWAAuthServlet` to support HTTP Basic authentication.

Your main task is to override the `doGet` method to do your main work. BillingCenter already authenticates the using HTTP Basic authentication headers before calling your `doWork` method. For more information about the parameters in this method, see “[Creating a Basic Servlet](#)” on page 433.

Also, override the `isAuthenticationRequired` method and return `true` if authentication is required for this request.

The following example responds to servlet URL substrings that start with the string `/test/`. If an incoming URL matches that pattern, the servlet simply echoes back the `PathInfo` property of the response object, which contains the path.

```
package mycompany.test

uses gw.servlet.Servlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpServlet
uses gw.api.util.Logger

@Servlet( \ path : String ->path.matches("/test(/.*)?"))
class TestingServlet extends gw.servlet.AbstractBasicAuthenticationServlet {

    override function doGet(req: HttpServletRequest, resp : HttpServletResponse) {

        print("servlet test url: " + req.RequestURI)
        print("query string: " + req.QueryString)

        resp.ContentType = "text/plain"
        resp.setStatus(HttpServletResponse.SC_OK)
        resp.getWriter.append("I am the page " + req.PathInfo)
    }

    override function isAuthenticationRequired( req: HttpServletRequest ) : boolean {

        // -- TODO -----
        // Read the headers and return true/false if user authentication is required
        // -----
        return true;
    }
}
```

This servlet responds to URLs with the word `test` in the service query path, such as the URL:

`http://localhost:8080/bc/service/test/is/this/working`

To test this, you must use a tool that supports adding HTTP Basic authentication headers for the username and password.

Note that the text `"/test"` in the URL is the important part that matches the servlet string. Change the port number and the server name to match your application.

Your web page displays the following:

`I am the page /test/is/this/working`

Use this basic design pattern to intercept any of the following:

- A single page URL
- An entire virtual file hierarchy, as shown in the previous example
- Multiple page URLs that are not described in traditional file hierarchies as a single root directory with subdirectories. For example, you could intercept URLs with the regular expression:

`"(/.*)?/my_magic_subfolder_one_level_down"`

That would match all of the following URLs:

`http://localhost:8080/bc/service/test1/my_magic_subfolder_one_level_down`
`http://localhost:8080/bc/service/test2/my_magic_subfolder_one_level_down`
`http://localhost:8080/bc/service/test3/my_magic_subfolder_one_level_down`

Data Extraction Integration

BillingCenter provides several mechanisms to generate messages, forms, and letters in custom text-based formats from BillingCenter data. If an external system needs information from BillingCenter about an account, it can send requests to the BillingCenter server by using the HTTP protocol.

This topic includes:

- “Why Gosu Templates are Useful for Data Extraction” on page 441
- “Data Extraction Using Web Services” on page 442

Why Gosu Templates are Useful for Data Extraction

Incoming data extraction requests include what Gosu template to use and what information the request passes to the template. With this information, BillingCenter searches for the requested data such as account data, which is typically called the *root object* for the request. If you design your own templates, you can pass any number of parameters to the template. Next, BillingCenter uses a requested template to extract and format the response. You can define your own text-based output format. See “Gosu Templates” on page 359 in the *Gosu Reference Guide*.

Possible output formats include:

- A plain text document with *name=value* pairs
- An XML document
- An HTML or XHTML document

You can fully customize the set of properties in the response and how to organize and export the output in the response. In most cases, you know your required export format in advance and it must contain dynamic data from the database. Templates can dynamically generate output as needed.

Gosu templates provide several advantages over a fixed, pre-defined format:

- With templates, you can specify the required output properties, so you can send as few properties as you want. With a fixed format, all properties on all subobjects might be required to support unknown use cases, so you must send all properties on all subobjects.

- Responses can match the native format of the calling system by customizing the template. This avoids additional code to parse and convert fixed-format data.
- Templates can generate HTML directly for custom web page views into the BillingCenter database. Generate HTML within BillingCenter or from linked external systems, such as intranet websites that query BillingCenter and display the results in its own user interface.

The major techniques to extract data from BillingCenter using templates are as follows:

- **For user interaction, write a custom servlet that uses templates** – Create a custom servlet. A servlet generates HTML (or other format) pages for predefined URLs and you do not implement with PCF configuration. See “Servlets” on page 433. Your servlet implementation can use Gosu templates to extra data from the BillingCenter database.
- **For programmatic access, write a custom web service that uses templates** – Write a custom web service. Custom web services let you address each integration point in your network. Follow these design principles:
 - Write a different web service API for each integration point, rather than writing a single, general purpose web service API.
 - Name the methods in your web service APIs appropriately for each integration point. For example, if a method generates notification emails, name your method `getNotificationEmailText`.
 - Design your web service APIs to use method arguments with types that are specific to each integration point.
 - Use Gosu templates to implement data extraction. However, do not pass template data or anything with Gosu code directly to BillingCenter for execution. Instead, store template data only on the server and pass only typesafe parameters to your web service APIs.

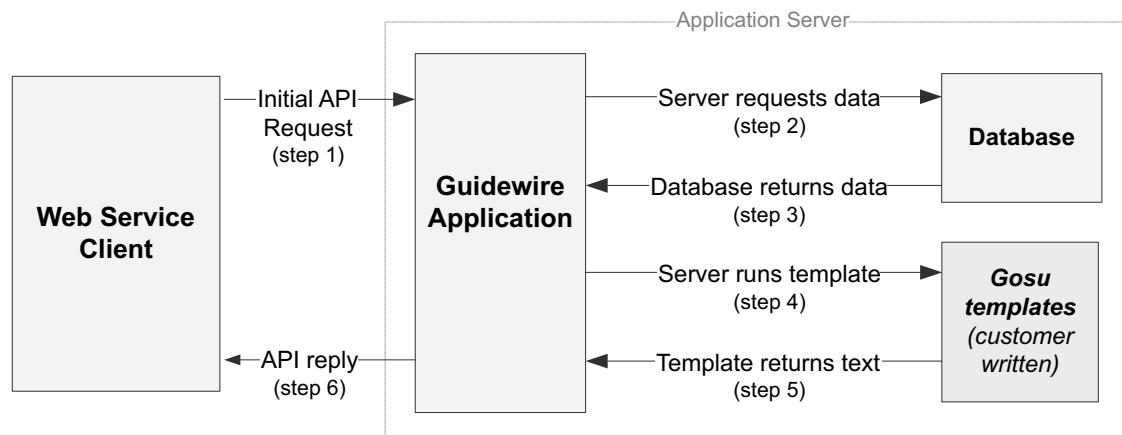
For more information, see “Data Extraction Using Web Services” on page 442.

Data Extraction Using Web Services

You can write your own web services that use Gosu templates to generate results. For more information about writing Gosu templates, see “Gosu Templates” on page 359 in the *Gosu Reference Guide*.

The following diagram illustrates the data extraction process for a web service API that uses Gosu templates..

Web Service Data Extraction Flow



Every data extraction request includes a parameter indicating which Gosu template to use to format the response. You can add an unlimited number of templates to provide different responses for different types of root objects. For example, given an account as a root object for a template, you could design different templates to export account data such as:

- A list of all notes on the account
- A list of all open activities on the account
- A summarized view of an account

To provide programmatic access to BillingCenter data, BillingCenter uses Gosu, the basis of Guidewire Studio business rules. Gosu templates allow you to embed Gosu code that generates dynamic text from evaluating the code. Wrap your code with <% and %> characters for Gosu blocks and <%= and %> for Gosu expressions.

Once you know the root object, refer to properties on the root object or related objects just as you do as you write rules in Studio.

Before writing a template, decide which data you want to pass to the template.

The simplest Gosu expressions only extract data object properties, such as `account.AccountNumber`. For example, `myAccount.accountNumber`.

```
The number is <%= myAccount.accountNumber %>.
```

At run time, Gosu runs the code in the template block “<%= ... %>” and evaluates it dynamically:

```
The number is H0-1234556789.
```

Error Handling in Templates

By default, if Gosu cannot evaluate an expression because of an error or `null` value, it generates a detailed error page. It is best to check for blank or `null` values as necessary from Gosu so that you do not accidentally generate errors during template evaluation.

Getting Parameters from URLs

If you want to get the value of URL parameters other than the root objects and/or check to see if they have a value use the syntax `parameters.get("paramnamehere")`. For instance, to check for the `xyz` parameter and export it:

```
<%= (parameters.get("xyz") == null)? "NO VALUE!" :parameters.get("xyz") %>
```

Built in Templates

There are example Gosu templates included with BillingCenter. Refer to the files within the BillingCenter deployment directory:

```
BillingCenter/modules/configuration/config/templates/dataextraction/*.gst
```

Structured Export Formats

HTML and XML are text-based formats, so there is no fundamental difference between designing a template for HTML or XML export compared to other plain text files. The only difference is that the text file must conform to HTML and XML specifications.

HTML results must be a well-formed HTML, ensuring that all properties contain no characters that might invalidate the HTML specification, such as unescaped “<” or “&” characters. This is particularly relevant for especially user-entered text such as descriptions and notes.

Systems that process XML often are very strict about syntax and well-formedness. Be careful not to generate text that might invalidate the XML or confuse the recipient. For instance, beware of unescaped “<” or “&” characters in a notes field. If possible, you could export data within an XML `<CDATA>` tag, which allows more types of characters and character strings without problems of unescaped characters.

Handling Data Model Extensions in Gosu

If you added data model extensions, such as new properties within the `Account` object, they are available work within Gosu templates just as in business rules within Guidewire Studio. For instance, just use expressions like `myAccount.myCustomField`.

Gosu Template APIs Common for Integration

Gosu Libraries

You can access Gosu libraries from within templates similar to how you would access them from within Guidewire Studio, using the `Libraries` object:

```
<%= Libraries.Financials.getTotalIncurredForExposure(exposure) %>
```

Java Classes Called From Gosu

Just like any other Gosu code, your Gosu code in your template use Java classes.

```
var myWidget = new mycompany.utils.Widget();
```

See also

- For information about calling Java from Gosu, see “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.

Logging

You can send messages to the BillingCenter log file using the BillingCenter logger object. Access the logger from Gosu using the following code:

```
libraries.Logger.logInfo("Your message here...")
```

Typecode Alias Conversion

As you write integration code in Gosu templates, you may also want to use BillingCenter typelist mapping tools. BillingCenter provides access to these tools by using the `$typecode` object:

This `$typecode` API works only within Gosu templates, not Gosu in general.

In addition, you can use the `TypecodeMapperUtil` Gosu utility class.

See also

- For more information on the Gosu utility class, see “Using Gosu or Java to Translate Typecodes” on page 126.

Logging

BillingCenter provides a robust logging system based on the open source Apache log4j project. The log4j system flexibly outputs log statements with arbitrary granularity at a specific logging level that indicate what to write to files. You can configure the logging system fully at runtime by using external configuration files. You can use the BillingCenter logging system throughout your application code. However, special features of the logging system are particularly useful for integration developers, and this topic discusses them.

This topic includes:

- “Logging Overview For Integration Developers” on page 445
- “Logging Properties File” on page 446
- “Logging APIs for Java Integration Developers” on page 447

See also

“Configuring Logging” on page 21 in the *System Administration Guide*

Logging Overview For Integration Developers

Logging Elements

The logging system in BillingCenter comprises these elements:

- **Logging properties files** – A `logging.properties` file specifies what information to log. For instance, you can choose to log absolutely everything relevant to integration, log plugin information, or log nothing. You can choose a large set of logging properties that log certain errors for certain cases or warnings for other cases. The logging properties file uses the format for the Java tool log4j.
- **Log files** – Create log files anywhere accessible from the server. You can create different types of log messages for different contexts in different files or even multiple server directories.
- **Code that triggers logging messages** – Many built-in parts of BillingCenter can log information, warnings, errors, or arbitrary debugging information. As an integration developer, your code can log anything you want. However, code that triggers logging does not force log messages to be written to files. The logging properties file specifies what BillingCenter actually does with any specific logging information.

Logging Types: Category-based and Class-based

BillingCenter provides two ways to configure logging:

- **Category-based logging** – BillingCenter provides a hierarchical, abstract category system for logging that avoids dependencies on specific Java classes. Category-based logging operates independently of Java packages and supports quick and easy configuration of log levels, log file locations, and other logging settings. Guidewire strongly recommends that you use category-based logging.
- **Class-based logging** – If you have legacy Java code that uses `log4j` class-based logging, it might be easier to continue your use class-based logging. However, Guidewire strongly recommends that you migrate your `log4j` code to category-based logging so your code integrates with the pre-defined logging categories of BillingCenter.

Logging Properties File

The logging properties file (`logging.properties`) configures what to log and where to log it. The logging properties file is in the `log4j` format, which the Apache project defined for the `log4j` system. This topic only briefly mentions the details of how to configure `log4j` files such as these.

Each logging properties file is separated into blocks such as the following example:

```
# set logging levels for the category: "log4j.category.Integration.plugin"
log4j.category.Integration.plugin=DEBUG, PluginsLog
# To remove logging for that category, comment out the previous line with an initial "#"

# The following lines specify how to write the log, where to write it, and how to format it
log4j.additivity.PluginsLog=false
log4j.appenders.PluginsLog=org.apache.log4j.DailyRollingFileAppender
log4j.appenders.PluginsLog.File=C:/Guidewire/BillingCenter/logs/plugins.log
log4j.appenders.PluginsLog.DatePattern = .yyyy-MM-dd
log4j.appenders.PluginsLog.layout=org.apache.log4j.PatternLayout
log4j.appenders.PluginsLog.layout.ConversionPattern=%-10.10X{server} %-4.4X{userID} %d{ISO8601} %p %m%n
```

In this example, the following line defines the class, log level, and a log name (`AdaptersLog`):

```
log4j.category.Integration.plugin=DEBUG, AdaptersLog
```

This line specifies to use a file appender (a standard local log file):

```
log4j.appenders.PluginsLog=org.apache.log4j.DailyRollingFileAppender
```

The line after that specifies the location of the log:

```
log4j.appenders.PluginsLog.File=C:/Guidewire/BillingCenter/logs/plugins.log
```

See also

<http://logging.apache.org/log4j/1.2/index.html>

Logging Categories for Integration

Logging categories are hierarchical. For example, enabling a log file appender for the category `log4j.category.plugin` enables `log4j.category.plugin.IValidationAdapter`. If there are subcategories defined for that category, those are enabled also. If you use logging categories from Java code, use the `LoggerCategory` class, which includes several static instances of the `Logger` interface that are pre-defined for common use.

See also

- “Category-based Logging” on page 447.
- For a current list of logging categories, see “Understanding Logging Categories” on page 23 in the *System Administration Guide*

Logging APIs for Java Integration Developers

Category-based Logging

BillingCenter provides an API to the log4j-based logging system by using the `LoggerCategory` class. The `LoggerCategory` class contains predefined static instances of the `Logger` interface. The `LoggerCategory` class is available to your Java code and your web services API client code.

For your Java plugins, logger configuration is automatic because the server already instantiated and configured a *logger factory*. A logger factory is the object that configures what to log and where to log it. A Java plugin automatically inherits the logging properties of the application server.

See also

If you want to use logging within a Java plugin, see “[Logger Classes](#)” on page 447.

See also

The API Reference Javadoc for `gw.api.system.logging.LoggerFactory`

Logger Classes

To use the `LoggerCategory` class, you first need an instance of the `Logger` class. The easiest way to get an instance is to use the static instances of this class predefined for common top level loggers, such as `PLUGIN` and `API`. You can access these loggers as properties of the `LoggerCategory` class. For instance, `LoggerCategory.PLUGIN` refers to the static instance of the `Logger` interface for plugins.

For example, you can use code like the following to log an error.

```
LoggerCategory.PLUGIN.error(Document + ", missing template: " + stringWithoutDescriptor);
```

To use this tool within a Java class, declare a private class variable of interface `Logger` like this.

```
private Logger _logger = null;
```

Then at runtime, get an instance of the `Logger` object. For example, use the built-in static instances of the `LoggerCategory` class.

```
_logger = LoggerCategory.PLUGIN;
```

You can now write to this logger object with `Logger` methods. The methods you typically use most are methods that log a message at a specific log4j logging level: `info`, `warn`, `trace`, `error`, and `debug`.

The following example logs a message at the `INFO` logging level.

```
_logger.info("Setting up logger for MySpecialCode...");
```

The logger does not append this message to a file unless the logger is configured to do so. You must set logging properties to enable that logging level and define a filename path for the log file output.

BillingCenter uses the initial setup of the logging factory to determine the logging level for this logger based on its category. For plugins, it inherits the server settings. However, you can override the logging level by changing `logging.properties`. You can also temporarily adjust the logging level for a logger by using the [Set Log Level](#) screen in BillingCenter. Changes made with the [Set Log Level](#) screen only persist until the BillingCenter server is restarted.

If you want to use a logger for which there is no static instance, use one of two alternate calls to the `LoggerFactory` class. The most common approach is to create a logger as a sublogger of an existing logger.

```
Logger logger = LoggerFactory.getLogger(LoggerCategory.PLUGIN, "IApprovalAdapter");
logger.info("My info message here")
```

Alternatively, create a new root category.

```
LoggerCategory logger = LoggerFactory.getLogger("MyRootCategoryName");
logger.info("My info message here")
```

To configure your new category in `logging.properties`, define the new logger and give it its own appender, as the following example shows.

```
log4j.category.IApprovalAdapter=DEBUG, MyLog
log4j.additivity.MyLog=false
log4j.appender.MyLog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.MyLog.File=c:/gwlogs/messaging.log
log4j.appender.MyLog.DatePattern = .yyyy-MM-dd
log4j.appender.MyLog.layout=org.apache.log4j.PatternLayout
log4j.appender.MyLog.layout.ConversionPattern=%-10.10X{server} %-4.4X{userID} %d{ISO8601} %p %m%n
```

If you created a new root category, replace `IApprovalAdapter` in the example above with your new category.

See also

- “Understanding Logging Levels” on page 23 in the *System Administration Guide*
- “Set Log Level” on page 160 in the *System Administration Guide*

Class-based Logging (Not Generally Recommended)

Instead of using abstract logging categories to identify related code, you can use the fully-qualified name of a class. By using a fully qualified class name, The class name and package define the hierarchy you use to define logging configuration settings.

IMPORTANT Guidewire strongly recommends you use the category-based approach, instead of the class-based approach described in this topic.

For instance, instead of your plugin code writing log messages with `LoggerCategory` to the `log4j.category.Integration.plugin.IValidationAdapter` class, configure a logger based on the actual class of your plugin. For example, you might use `com.mycompany.myadapters.myValidationAdapter`. To use the class-based approach, use the `Logger` and `LoggerFactory` classes.

Just as for category-based logging, plugin logger configuration is automatic because the server already instantiated and configured a *logger factory*. The logger factory configures what to log and where to log it. However, for web services API client code, you must explicitly set up the logger factory using the `LoggerFactory` class. Plugin code and web services API client code can set up a logger factory using the `LoggerFactory` class.

To use class-based logging in your code

- Configure the logger in the `logging.properties` file by using the class name instead of the category name.
- Use a `LoggerFactory` instance to create a `Logger` instance.
- Send logging messages to that `Logger` instance.

For a typical example for a plugin, set a class private variable.

```
private Logger _logger = null;
```

Then in your set up code, initialize the logger.

```
_logger = LoggerFactory.getLogger(MyJavaClassName.class);
```

And then you can send logger messages with it, with similar methods as in `LoggerCategory`.

```
_logger.info("Setting up logger ...");
```

See also

- “Category-based Logging” on page 447

Dynamically Changing Logging Levels

You can change logging levels without redeploying BillingCenter. For more information, see “Making Dynamic Logging Changes without Redeploying” on page 28 in the *System Administration Guide*.

Java and OSGi Support

This topic describes ways to write and deploy Java code in BillingCenter, including accessing entity data from Java. For example, you could implement BillingCenter plugin interfaces using a Java class instead of a Gosu class. If you implement a plugin interface in Java, optionally you can write your plugin implementation as an OSGi bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java libraries. Guidewire recommends OSGi for all new Java plugin development.

See also

- You can write Gosu code that uses Java types. See “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.
- For an introduction to Gosu from a Java perspective, see “Gosu Introduction” on page 15 in the *Gosu Reference Guide*.

This topic includes:

- “Overview of Java and OSGi Support” on page 451
- “Accessing Entity and Typecode Data in Java” on page 455
- “Accessing Gosu Classes from Java Using Reflection” on page 466
- “Gosu Enhancement Properties and Methods in Java” on page 467
- “Class Loading and Delegation for non-OSGi Java” on page 467
- “Deploying Non-OSGi Java Classes and JARs” on page 468
- “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 469
- “Advanced OSGi Dependency and Settings Configuration” on page 476
- “Updating Your OSGi Plugin Project After Product Location Changes” on page 477

Overview of Java and OSGi Support

You can deploy Java code within BillingCenter. There are several different ways you can use Java.

Typical customers write Java code primarily to implement BillingCenter plugin interfaces.

However, you can write Java code that you can call from any Gosu code in BillingCenter, such as from rule sets or other Gosu classes.

In all cases for Java development, you must use an IDE for Java development separate from BillingCenter Studio. It is unsupported to add or modify Java class files in the BillingCenter Studio user interface. Although BillingCenter Studio does not hide user interface tools that add Java classes to the file hierarchies, those features are unsupported.

If you are deploying Java plugins or OSGi plugins, carefully read about your IDE options in “[Implementing Plugin Interfaces in Java and Optionally OSGi](#)” on page 452.

Learning More About Entity Java APIs

If you want to deploy Java code and you do not use Guidewire entity data, the main thing you need to know is where to put Java classes and libraries. See “[Deploying Non-OSGi Java Classes and JARs](#)” on page 468.

If your Java code needs to get, set, or query Guidewire entity data, you must also understand how BillingCenter works with entity data in Java. See “[Accessing Entity and Typecode Data in Java](#)” on page 455.

Accessing Gosu Types from Java

From Gosu, you can call Java types, including added third-party Java classes and libraries. However, from Java you cannot access Gosu types without using a language feature called *reflection*. Reflection means to ask the type system at run time about types. Using reflection is not typesafe, which means you cannot catch some types of errors at compile time. For example, you must use reflection to access the following from Java: Gosu classes, Gosu interfaces, and Gosu enhancements. See the following topics:

- “[Accessing Gosu Classes from Java Using Reflection](#)” on page 466
- “[Gosu Enhancement Properties and Methods in Java](#)” on page 467

Note that some of the supported BillingCenter types that you might use in Gosu are actually implemented in Java and thus require no special access from Java.

Implementing Plugin Interfaces in Java and Optionally OSGi

Many customers write Java code primarily to implement plugin interfaces. For general information about plugins, see “[Plugin Overview](#)” on page 135.

Conceptually, there are two main steps to implement a plugin:

1. **Write a class that implements the interface** – See “[Implementing Plugin Interfaces](#)” on page 137
2. **Register your plugin implementation** – See “[Registering a Plugin Implementation Class](#)” on page 139

If you implement a plugin interface in Java, there are two ways to deploy your code:

- **Java plugin** – A Java class. You must use an IDE other than Studio. If you write your plugin in Java, you must regularly regenerate the Java API libraries to compile against the latest libraries. You can use any Java IDE. You can choose to use the included IntelliJ IDEA with OSGi Editor for Java plugin development even if you do not choose to use OSGi.
- **OSGi plugin** – A Java class encapsulated in an *OSGi* bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java libraries. To simplify OSGi configuration, BillingCenter includes an application called IntelliJ IDEA with OSGi Editor. For details of deploying the files, see “[Deploying Non-OSGi Java Classes and JARs](#)” on page 468.

IMPORTANT Guidewire recommends OSGi for all new Java plugin development. BillingCenter supports OSGi bundles only to implement a BillingCenter plugin interface and any of your related third-party libraries. It is unsupported to install OSGi bundles for any other purpose.

For more information about the OSGi standard, refer to:

<http://www.osgi.org/Technology/WhyOSGi>

The most important benefits of OSGi for BillingCenter plugin development are:

- Safe encapsulation of third-party Java JAR files. OSGi loads types in a way that reduces compatibility problems between OSGi bundles, or between an OSGi component and libraries that BillingCenter uses. For example, dependencies on specific third-party packages and classes are explicit in manifest files and validated at startup time.
- Dependency injection support using *declarative services*. Declarative services use Java annotations and interfaces to declare dependencies between your code and other APIs. For example, your code does not declare that it needs a specific class for a task. Instead, your code uses Java interfaces to define which services it needs. The OSGi framework ensures the appropriate API or objects are available. The OSGi framework tracks dependencies and handles instantiates objects as needed. For example, your OSGi plugin might depend on a third-party OSGi library that provides a service. Your plugin code can use declarative services to access the service.

Beware of a terminology issue in Guidewire documentation with the word *bundle*:

- In nearly all Guidewire documentation, *bundle* refers to a programmatic abstraction of a database transaction and the set of database rows to update. See “Bundles and Database Transactions” on page 341 in the *Gosu Reference Guide*.
- In the OSGi standard, *bundle* refers to a registered OSGi component. BillingCenter documentation relating to Java and plugins sometimes refers to *OSGi bundles*.

IDE Options for Plugin Development in the Java Language

You can use any Java IDE that is separate from BillingCenter Studio. It is unsupported to edit Java directly in Studio.

If you are writing an OSGi plugin implementation, there are several IDE options:

- IntelliJ IDEA with OSGi Editor (included with BillingCenter)** – A specially-configured instance of IntelliJ IDEA and included with BillingCenter. This is a different application than BillingCenter Studio. This IntelliJ IDEA instance includes a special IntelliJ IDEA plugin for OSGi plugin configuration. For OSGi plugin development, Guidewire recommends using IntelliJ IDEA with OSGi Editor. The included plugin editor configures OSGi configuration files such as the bundle manifest.
- Other Java IDEs such as Eclipse** – You can use other Java IDEs such as Eclipse or your own version of IntelliJ IDEA. However, you must manually configure OSGi files and bundle manifest manually according to the OSGi standard.

The following table summarizes options for development environments for plugin development in Java.

| IDE | Gosu plugin | Java plugin (no OSGi) | OSGi plugin (Java with OSGi) |
|--|-------------|-----------------------|--|
| BillingCenter Studio | Yes | -- | -- |
| IntelliJ IDEA with OSGi Editor (included with BillingCenter) | -- | Yes | Yes |
| Eclipse or other Java IDE of your own choice | -- | Yes | Yes, though requires manual configuration of OSGi files and bundle manifest. |

Inspections to Flag Unsupported Internal Java APIs

The Java API allows you to use the same Java types that you can use in Gosu. However, Guidewire specifies some methods and fields on these types for internal use only. Do not use any of these *internal APIs*. Guidewire indicates internal API methods and properties with the annotation `@gw.lang.InternalAPI`.

In Gosu, methods and fields with that annotation are hidden. Gosu code that uses internal APIs triggers compilation errors.

In Java, when you are using your own IDE separate from Studio, internal APIs are visible although unsupported. Depending on what IDE you use, you may require additional configuration of your IDE. The following table summarizes your options for internal API code inspections.

| IDE | Java IDE Includes Internal API Inspection |
|--|--|
| IntelliJ IDEA with OSGi Editor (included with BillingCenter) | Yes |
| Separate IntelliJ IDEA instance of your own choice | Optional installation. For compatibility with specific IntelliJ IDEA versions, please contact Guidewire Customer Support. See “Installing the Internal API Inspection in Your Own Instance of IntelliJ IDEA” on page 454 |
| Eclipse or other Java IDE of your own choice | No. There is no equivalent code inspection available. If you use another IDE and you are unsure of the status of a particular method or field, navigate to it and see if the declaration has the annotation <code>@gw.lang.InternalAPI</code> . |
| BillingCenter Studio | WARNING: BillingCenter Studio does not support Java coding directly in the IDE. Do not create Java classes directly in BillingCenter Studio. If you want to code in Java, you must use a separate IDE for Java development. See “IDE Options for Plugin Development in the Java Language” on page 453 |

Installing the Internal API Inspection in Your Own Instance of IntelliJ IDEA

The recommended approach for Java development is with the included IntelliJ IDEA with OSGi Editor. If you instead choose to use your own separate instance of IntelliJ IDEA, you may be able to use the Internal API inspection, depending on your version of IntelliJ IDEA. For compatibility with specific IntelliJ IDEA versions, please contact Guidewire Customer Support.

IMPORTANT The IntelliJ IDEA with OSGi Editor automatically includes the code inspection for the `@InternalAPI` annotation. If you use this included application, you do not need to install any code inspections to flag internal API usage.

To install the IntelliJ IDEA Internal API Code Inspection

1. Open your separate instance of IntelliJ IDEA.
2. Navigate to **Settings → Plugins**.
3. Click **Install from disk...**
4. Navigate to the directory:
`BillingCenter/studio/configstudio/internal-api-idea-plugin/lib/`
5. Select the JAR in that directory.
6. Let IntelliJ IDEA restart after installing the plugin.
7. Navigate to **Settings → Inspections**.
8. Expand the node **Portability Issues**.
9. Check the box **Use of internal API**.
10. Click **OK**.

Accessing Entity and Typecode Data in Java

An *entity* type is an abstract representation of Guidewire business data of a certain type. Define entity types in data model configuration files. For entity types in the default configuration, entity types have built-in properties and you can optionally add extension properties. User and Address are examples of entity types.

You can write Java code that accesses entity data from:

- Java plugin implementations
- OSGi plugin implementations (written in Java)
- Other Java classes called from Gosu.

After you call a script that regenerates BillingCenter Java API libraries, you can write Java code that uses them to access entity data. See “Regenerating Integration Libraries and WSDL” on page 20.

Using a separate Java-capable IDE, add the Java API library directories as a dependency in your project in the separate IDE. Compile your Java code against these libraries.

IMPORTANT Do not create Java classes directly in Studio. To code in Java, you must use a separate IDE for Java development. For example, use a separate instance of IntelliJ IDEA or Eclipse.

Your Java code can get or set entity data or call additional *domain methods* with similar functionality in Java as in Gosu. For example, a Message object as viewed through the Java API library interface has data getter and setter methods to get and set data. The Message object has `getPayload` and `setPayload` methods that manipulate the `Payload` property. Additionally, the object has additional methods that trigger complex logic, such as the `reportAck` method.

Understanding and using the entity libraries is critical in the following situations:

- **Java plugin development.** Most BillingCenter plugin implementations must access entity data or change entity data. Also, some plugin interface methods explicitly have entity data as method arguments or return values. See “Plugin Overview” on page 135.
- **Other Java classes that use entity data.** Even if your Java code does not implement a plugin interface, your Java code can access entity data.

In both cases, your code can:

- take entity data as method arguments or return values
- search for entity data
- change entity data

When you are done writing your Java code, deploy your class files and JAR files:

- For non-OSGi Java, see “Deploying Non-OSGi Java Classes and JARs” on page 468
- For OSGi plugins, see “Using Third-Party Libraries in Your OSGi Plugin” on page 473s

Regenerating Java API Libraries

Anytime you change the data model due to extensions or customizations, you must regenerate the entity libraries and compile your Java code against the latest libraries.

To regenerate the Java API libraries

1. In Windows, open a command prompt.
2. Change your working directory with the following command:

```
cd BillingCenter/bin
```

- Use the `regen-java-api` command:

```
gwbc regen-java-api
```

This command generates Java entity interfaces and typelist classes in libraries in the location:

```
BillingCenter/java-api/lib
```

This command generates Javadoc documentation in the location:

```
BillingCenter/java-api/doc
```

Entity Packages and Customer Extensions from Java

In Gosu, you can refer to an entity type using the syntax simply `entity.ENTITYNAME` or simply the entity name because the package `entity` is always in scope.

In the BillingCenter Java API, you can reference a type directly by its fully-qualified name. However, for BillingCenter entity types, from Java the fully-qualified name of an entity is not `entity.ENTITYNAME` nor simply the entity name. The syntax `entity.ENTITYNAME` or using the entity name with no package is a shortcut of the Gosu type system.

If you want only the base configuration properties, the type name is the same in Java as in Gosu but the package varies by entity type. Some aspects of the fully-qualified names of the interfaces are configurable. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 459. BillingCenter exposes each entity type as several interfaces, which the following table summarizes.

| Terminology | Description | When it exists | Entity name suffix | For more information, including the Java package for the interface |
|---|--|--|--------------------|---|
| Entity types that Guidewire originally creates | | | | |
| <i>base entity interface</i> | Contains only the base configuration properties. | All entity types have a base entity interface. | n/a | “Accessing Entity Properties and Methods With Base and Core Extension Interfaces” on page 457 |
| Entity types that you originally create | | | | |
| <i>customer extension entity interface</i> | <p>Includes customer data model extension properties.</p> <p>If a core extension interface exists for that entity type, the customer extension interface extends from the core extension entity interface. Otherwise, the customer extension interface extends from the base entity interface.</p> | Exists only if there are customer data model extensions. | Ext | “Customer Extension Entity Interface” on page 458 |
| <i>customer entity interface</i> | Contains all properties | All entities that you create have this interface | n/a | “Entity Interfaces for Completely Custom Entity Types” on page 459. |

Accessing Entity Properties and Methods With Base and Core Extension Interfaces

From Java, the base entity interface package includes a *prefix* and a *subpackage* that defines a general area of functionality in the application. The general pattern for the fully-qualified base entity type name is:

```
PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAME
```

For entity types that multiple Guidewire applications share, the pattern is:

```
gw.p1.SUBPACKAGE.entity.ENTITYNAME
```

For entities specific to the BillingCenter application, the pattern is:

```
gw.bc.SUBPACKAGE.entity.ENTITYNAME
```

For example, fully-qualified name of the Address entity is:

```
gw.p1.contact.entity.Address
```

Instead of memorizing fully-qualified names of the entities, use the auto-completion features of your IDE. For example, if you type Address then type CTRL+Space in IntelliJ IDEA, the IDE presents options including the entity with the pattern described above. After you choose the correct entity type, the IDE inserts an `import` statement at the top of the file. For example, the IDE might add the following line:

```
import gw.p1.contact.entity.Address;
```

If you write Java code that implements a BillingCenter plugin interface, interface method arguments and return types that reference entity types always uses the Java name for the type. The Java fully-qualified name for the entity type is different than the type name in Gosu, and in fact in Java is represented by three interfaces. Remember this difference as you review documentation examples that may show code in Gosu.

Some aspects of the fully-qualified names of the interfaces are configurable. See “Configuring Entity and Type-list Fully-Qualified Names from Java” on page 459.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

Core Extension Entity Interfaces

For some entity types, the Guidewire platform defines a base version and then BillingCenter extends the base entity interface with properties and methods for application-specific business logic.

Only for such entities, BillingCenter adds a *core extension entity interface* with the suffix `CoreExt`. For example, the Message entity contains many properties in its base entity interface. However, each Guidewire application adds additional properties to Message for business logic unique to each application to reference a primary object for that message.

The package for the core extension entity interface is slightly different from the base entity interface. For entities that have core extension entity interfaces, the base entity interface has the pattern:

```
gw.p1.SUBPACKAGE.entity.ENTITYNAME
```

By definition, core extension entity interfaces are specific to the BillingCenter application. Therefore, the pattern matches the pattern for application-specific entities:

```
gw.bc.SUBPACKAGE.entity.ENTITYNAMECoreExt
```

For example, Java code that references the base entity interface and core extension entity interface for the Message entity has the following lines at the top of the Java file:

```
import gw.p1.messaging.entity.Message;
import gw.bc.messaging.entity.MessageCoreExt;
```

The core extension entity interface extends the base entity interface, so it also contains all the methods of the base entity interface. To use application-specific properties, downcast an object reference to the core extension entity interface.

For example, in a BillingCenter message transport plugin, to use the `Message.Account` property, you must downcast to the *core extension interface* to access that property:

```
import gw.bc.claim.entity.Account;
import gw.p1.messaging.entity.Message;
```

```

import gw.bc.messaging.entity.MessageCoreExt;
class MyMessageTransport implements MessageTransportPlugin {
    // ...
    public void send(Message message, String transformedPayload) {
        // IMPORTANT: To access some properties, cast Message to subinterface MessageCoreExt
        MessageCoreExt m = (MessageCoreExt) message;

        // access a property on MessageCoreExt that is absent on Message
        Account a = m.getAccount();

        // ... use this Account information and send a message...
    }
}

```

Alternatively, to access the application-specific properties or methods, you can use the customer extension entity interface. Only if there are customer data model extensions, BillingCenter creates a customer extension entity interface, which extends from the base entity interface or the core extension interface if it exists. For details, see “Customer Extension Entity Interface” on page 458.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

Customer Extension Entity Interface

For each entity, if there are any customer data model extensions, there is an additional interface called the *customer extension entity interface*. If there are no customer data model extensions, BillingCenter does not create this interface.

Important qualities of the customer extension entity interface:

- The interface contains your data model extension properties.
- The interface name has the suffix `Ext`.
- The interface package is different from the package of the base entity interface. The package prefix changes to the *Java API package prefix* as defined in the file `extensions.properties`. See the table in “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 459.
- The customer extension entity interface extends from the base entity interface or the core extension interface if it exists. Therefore if there is a customer extension entity interface, it contains the complete set of all possible properties defined by data model configuration files.
- Like all the other Java entity interfaces, the customer extension entity interface does not include properties added by Gosu enhancements. See “Gosu Enhancement Properties and Methods in Java” on page 467.

The Java fully-qualified name of the customer extension entity interface is:

`PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAMEExt`

By default, `PACKAGE_PREFIX` is `extensions.bc`, and the subpackage varies by entity type in the entity declaration. For configuration of the `PACKAGE_PREFIX`, see “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 459.

For example, consider the `Address` entity type. In Gosu, use either of the following syntax styles:

```

Address
entity.Address

```

The Java fully-qualified name of the base entity interface is:

`gw.pl.contact.entity.Address`

By default, the Java fully-qualified name of the customer extension interface is:

`extensions.bc.contact.entity.AddressExt`

If the customer extension entity interface exists, you can downcast a variable declared to the base entity interface to the customer extension entity class. The customer extension class contains the customer extension properties as well as all the properties and methods from the base and core extension entity interfaces.

For example, the following Java code downcasts to access extension properties:

```
import gw.pl.contact.entity.Address;
import extensions.bc.contact.entity.AddressExt;

...
public void sendAddressProperties ( Address a ) {

    // downcast from Address to AddressExt. See the import statements above.
    AddressExt addressWithExtension = (AddressExt) a;

    // use extension properties from Java...
    System.out.println(addressWithExtension.getMyExtensionProperty());
}
```

After modifying the data model configuration in ways that affect extension properties, remember to regenerate the Java libraries. See “Regenerating Java API Libraries” on page 455.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

Entity Interfaces for Completely Custom Entity Types

If you create custom entity types (rather than extend existing ones), BillingCenter creates a customer entity interface with the following Java fully-qualified name:

`PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAME`

By default:

- `PACKAGE_PREFIX` is `extensions.bc`, though can be configured. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 459.
- `SUBPACKAGE` is defined by the `subpackage` attribute in the entity declaration (or the default) in the data model configuration file. If it is omitted, a default package is used. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 459.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

Configuring Entity and Typelist Fully-Qualified Names from Java

You can control the Java package names for entity types that you add in BillingCenter data model configuration XML files. As mentioned earlier, the generated entity package includes a package prefix and a subpackage using the following pattern for fully-qualified entity type names:

`PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAME`

There are several ways to configure the package of entity types in Java, as described in the following table.

| Package configuration for Java entity or type-list types | How to configure | Affects types in the base configuration and not in the extensions folder | Affects types that contain customer extensions | Affects types that you add, including typelists and subtypes of existing entities |
|--|--|--|--|---|
| custom subpackage | <p>In an entity type or typelist declaration, there is an attribute called subpackage. Set this attribute to a custom subpackage.</p> <p>If you omit the subpackage attribute, BillingCenter uses a default subpackage. See later in this table regarding configuring the default subpackage.</p> | No | No | Yes |
| default subpackage | <p>If you omit the subpackage attribute on the entity or typelist definition, BillingCenter uses a default subpackage. To change the default subpackage, edit the file <code>extensions.properties</code> in Studio. Change the property <code>gw.pl.metadata.codegen.package.default.subpackage</code> to your desired subpackage.</p> <p>In the base configuration, the default subpackage is <code>billing</code>.</p> <p>In the base configuration of <code>ContactManager</code>, the default subpackage is <code>contact</code>.</p> | No | No | Yes |
| package prefix | <p>You can change the package prefix from Java for entity types that you create by editing the file <code>extensions.properties</code> in Studio. Change the property <code>gw.pl.metadata.codegen.package.prefix</code> to the desired prefix. In the default configuration, the package prefix is <code>extensions.bc</code>.</p> <p>This package prefix is used for generating entity type interfaces for two cases:</p> <ul style="list-style-type: none"> • Entity types that you originally create • Customer extension entity interface, which BillingCenter generates when you extend an existing entity type with new properties. See “Customer Extension Entity Interface” on page 458. <p>In contrast, BillingCenter ignores this package prefix configuration when generating:</p> <ul style="list-style-type: none"> • The base entity interface • The core entity interface <p>For typelists, the same rules apply for the package prefix in typelist classes. BillingCenter uses the package prefix in generating typelist types for:</p> <ul style="list-style-type: none"> • Typelists that you originally create • Customer extension typelist classes, which BillingCenter generates when you extend an existing typelist. See “Typecode Classes from Java” on page 460. <p>Warning: In the file <code>extensions.properties</code>, there is an additional package prefix <code>gw.pl.metadata.codegen.package.prefix.internal</code>. That is for internal use only. Do not change it.</p> | No | Yes | Yes |

Typecode Classes from Java

The Java API exposes typelists and typecodes as Java classes, in contrast to entity types which BillingCenter exposes as interfaces.

To access a typecode, first get a reference to the appropriate typelist class. The package naming is similar to the pattern for entity data, with some differences. See “Entity Packages and Customer Extensions from Java” on page 456.

BillingCenter exposes each typelist type as several classes, which the following table describes:

- For the rightmost column, *TL* represents the typelist name, and *SUBPACKAGE* represents the subpackage. The typelist type declaration `subpackage` attribute specifies the *SUBPACKAGE*. This attribute only exists on the original `.tti` file that declares the typelist.
- If the typelist type declaration omits the `subpackage` attribute, a default is used based on whether the typelist is a *platform typelist*. If the typelist declaration attribute `platform` has value `true`, the typelist is a Guidewire platform typelist.
- Whether a typelist is a platform typelist also affects the package prefix. Refer to the table for details.

| Terminology | Description | Fully-qualified type name |
|--|---|--|
| Typelists that Guidewire defines | | |
| <i>base typelist class</i> | This class contains only the base configuration typecodes. All typelists that Guidewire creates have a base typelist class. This class contains typecodes in the following data model files: <code>.tti</code> | <p>For platform typelists:</p> <ul style="list-style-type: none"> The class is <code>gw.p1.<i>SUBPACKAGE</i>.typekey.<i>TL</i></code> The default subpackage is <code>platform</code> and is not configurable. <p>For non-platform typelists:</p> <ul style="list-style-type: none"> The class is <code>gw.bc.<i>SUBPACKAGE</i>.typekey.<i>TL</i></code> The default subpackage is <code>billing</code>. For ContactManager, the default subpackage is <code>contact</code> The default subpackage is configurable in the <code>extensions.properties</code> file. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 459 <p>For example, for platform typelist <code>BatchProcessType</code>:</p> <ul style="list-style-type: none"> Class name is <code>BatchProcessType</code> Package is <code>gw.p1.batchprocessing.typekey</code> |
| <i>internal extension typelist class</i> | This class contains base typecodes and application-specific typecodes. This class exists only if the Guidewire platform defines a base version but BillingCenter provides additional typecodes for application-specific use. This class contains typecodes in the following data model files: <code>.tti</code> , <code>.tix</code> | <p>For all internal extension typelist classes:</p> <ul style="list-style-type: none"> The class is <code>gw.bc.<i>SUBPACKAGE</i>.typekey.<i>TLConstants</i></code> The subpackage is defined by the subpackage of the typelist this extends. If it is undeclared, the default is <code>platform</code> and is not configurable. <p>For example, for platform typelist <code>BatchProcessType</code>:</p> <ul style="list-style-type: none"> Class name is <code>BatchProcessTypeConstants</code> Package is <code>gw.bc.batchprocessing.typekey</code> |

| Terminology | Description | Fully-qualified type name |
|--|--|--|
| Typelist extensions and new typelists | | |
| <i>customer extension typelist class</i> | This class contains base typecodes, application-specific typecodes, and customer typecodes. This class exists only if there are customer data model extensions. If an internal extension typelist class exists for that typelist type, the customer extension typelist class includes everything from the internal extension typelist class. Otherwise, the customer extension typelist class includes typecodes from the base typelist class. This class contains typecodes in the following data model files: .tti, .tix, .tx. | <p>For all customer extension typelist classes:</p> <ul style="list-style-type: none"> The class is: <code>PACKAGE_PREFIX.SUBPACKAGE.typekey.TLExtConstants</code> <code>SUBPACKAGE</code> is defined by the subpackage of the typelist this extends. Refer to the row in this table for the <i>base typelist class</i>. By default, <code>PACKAGE_PREFIX</code> is <code>extensions.bc</code> but is configurable. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 459 <p>For example, for platform typelist <code>BatchProcessType</code>:</p> <ul style="list-style-type: none"> Class name is <code>BatchProcessTypeExtConstants</code> Package is <code>extensions.bc.batchprocessing.typekey</code> |
| <i>customer typelist class</i> | This class contains all typecodes. All typelists that you create have this class. Contains typecodes in the following data model files: .tti. | <p>For all customer typelist classes:</p> <ul style="list-style-type: none"> The class is <code>PACKAGE_PREFIX.SUBPACKAGE.typekey.TL</code> The default subpackage is <code>billing</code>. For <code>ContactManager</code>, the default subpackage is <code>contact</code> The default subpackage is configurable in the <code>extensions.properties</code> file. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 459 By default, <code>PACKAGE_PREFIX</code> is <code>extensions.bc</code> but is configurable. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 459. <p>For example, for typelist <code>Example</code> with omitted subpackage:</p> <ul style="list-style-type: none"> Class name is <code>Example</code> Package is <code>extensions.bc.billing.typekey</code> |

Getting Typecodes Using the get method

The static properties on a typelist that represent a typecode have the `TC_` prefix, just like from Gosu. However, to actually work with the typecode, you must call the `get` method on the static property. The `get` method returns the appropriate typecode object.

For example, suppose you want the typecode `Approval` from a fictional example typelist called `ExampleType`. Assume that the typelist is declared with the example subpackage `testsub`.

- If the type is in the base configuration for multiple Guidewire applications, use the syntax:
`tc = gw.pl.testsub.typekey.ExampleType.TC_APPROVAL.get()`
- If the type is defined in the base configuration and extended for BillingCenter, use the syntax:
`tc = gw.bc.testsub.typekey.ExampleTypeConstants.TC_APPROVAL.get()`
- If the type is defined in a customer data model extension, use the syntax:
`tc = extensions.bc.testsub.typekey.ExampleTypeExtConstants.TC_APPROVAL.get()`

If you forget to call the `get` method, the resulting object is inappropriate for comparing typecodes, as well as most other contexts. For related information, see “Comparing Entity Instances and Typecodes” on page 463.

Wherever possible, use the static constants for typecode values. Using the typecode static constants ensures that your code is type safe. Mistakes in the typecode string are caught as compile errors rather than only at run time. If the code is known at compile time, always use the static constants to get the typekey object.

In the rare cases where you need to get a typecode reference from a `String` value known only at run time, use the `getTypeKey` method on the typelist class. Be warned that the `getTypeKey` method uses reflection, and typecode values cannot be validated at compile time. For example:

```
tc = gw.pl.testsub.typekey.ExampleType.getTypeKey(typeCodeString)
```

Entity Subtype Typelists

In addition to standard typelists, there are special typelists that exist only to distinguish *subtypes* of entities. These *subtype typelists* help BillingCenter specify the entity subtype in each database row.

The typelist name matches the name of the entity type. Based on this typelist name, BillingCenter creates the typelist classes for the Java API using the same basic pattern as regular typelists in “Typecode Classes from Java” on page 460.

If the entity subtype is shared among multiple Guidewire applications, the Java class name is:

```
gw.pl.SUBPACKAGE.typekey.ENTITYNAME
```

If BillingCenter overrides the entity subtype, there is a typelist class with the Java class name:

```
gw.bc.SUBPACKAGE.typekey.ENTITYNAMEConstants
```

If you extend an entity subtype of a pre-existing entity, there is a typelist class with the Java name:

```
PACKAGE_PREFIX.SUBPACKAGE.typekey.ENTITYNAMEExtConstants
```

If you created the entity subtype, there is a typelist class with the Java name:

```
PACKAGE_PREFIX.SUBPACKAGE.typekey.ENTITYNAMEExtConstants
```

Comparing Entity Instances and Typecodes

To compare entity instances for equality, always use the `entity.equals(otherEntity)` method. Using the `==` (double equals) operator is unsafe to test for equality.

To compare typecode values for equality, similarly use the `equals` method. However, be sure that you have a reference to the actual typecode class using the `get` method as mentioned in “Typecode Classes from Java” on page 460.

For example, suppose that the code `claim.getState()` returns a typecode type called `AccountState` and you want to compare the result to a specific value.

It is incorrect to do either of the following because the Java `==` (double equals) operator is unsupported for comparing typecodes:

```
account1.getState() == account2.getState()  
account.getState() == AccountState.TC_OPEN.get()
```

The following line is incorrect because it omits the `get` method after getting a static instance by name:

```
account.getState().equals(AccountState.TC_OPEN)
```

Instead, use the following expression syntax:

```
account.getState().equals(AccountState.TC_OPEN.get())  
account1.getState().equals(account2.getState())
```

Be very careful with code that compares two entities or two typecodes.

Entity Bundles and Transactions from Java

Before writing Java code, regenerate the Java API libraries. This process ensures that any data model changes or extensions are in the most recently generated libraries. See “Regenerating Java API Libraries” on page 455.

If you implement a Java plugin, see “Plugin Overview” on page 135 and “Special Notes For Java Plugins” on page 142.

Getting a Reference to an Existing Bundle in Java

To use entity instances, in many cases you need a reference a *bundle*. A bundle is a programmatic abstraction that represents one database transaction. See “Bundles and Database Transactions” on page 341 in the *Gosu Reference Guide*.

There are some programming contexts in BillingCenter in which there is a current database transaction, which means there is a current *bundle*. For example, the following code contexts include a current bundle:

- Code called from business rules
- Plugin interface implementation code, or code called by a plugin implementation
- Most PCF user interface application code

WARNING There are many contexts in which it is unsafe to commit a bundle. For example, all plugin code and most PCF code. For more information, see “Bundles and Database Transactions” on page 341 in the *Gosu Reference Guide*.

To get the current bundle from Java, use the same API as in Gosu:

```
gw.pl.persistence.core.Bundle b = gw.transaction.Transaction.getCurrent();
```

If there is no current bundle, you must create a bundle before creating entity instances or updating entity instances that you get from a database query.

New Bundles In Java

In general, Java code does not need to create a new bundle because there is already a bundle to use for that kind of code context. There are rare cases in which you need to create a new bundle, but only do this if necessary. If you have questions, please contact Guidewire Customer Support.

WARNING There are many contexts in which it is unsafe to commit a bundle. For example, all plugin code and most PCF code. For more information, see “Bundles and Database Transactions” on page 341 in the *Gosu Reference Guide*.

You can directly create a new bundle using the `newBundle` method on the `Transaction` class:

```
import gw.pl.persistence.core.Bundle;
import gw.transaction.Transaction;

...
Bundle bundle = Transaction.newBundle();
```

Additionally, you can use the `runWithNewBundle` API, which is the same as in Gosu for this task. However, the corresponding Gosu method takes a Gosu block as an argument. From Java, the syntax is slightly different, with an anonymous class instead of a Gosu block. For example:

```
gw.transaction.Transaction.runWithNewBundle(new Transaction.BlockRunnable() {
    @Override
    public void run(Bundle bundle) {
        // your code here...
    }
    // The bundle commits automatically if no exceptions happen before the end of the run method
});
```

Also see “Running Code in an Entirely New Bundle” on page 351 in the *Gosu Reference Guide*.

Creating New Entity Instances from Java

The recommended API for creating an entity instance is to call the `newInstance` method on the entity type’s `TYPE` property. Pass a bundle reference as a method argument. Depending on whether you need customer extension properties, the syntax varies. For discussion of the two separate interfaces for each entity type, see “Entity Packages and Customer Extensions from Java” on page 456.

From Java plugin code, there is a current bundle. You can use the `Transaction.getCurrent()` method to get the current bundle if one exists.

If there is a current bundle, create a new `Address` entity instance using the following code:

```
import gw.pl.persistence.core.Bundle;
import gw.pl.contact.entity.Address;
import extensions.bc.contact.entity.AddressExt;
import gw.transaction.Transaction;

...
Bundle b = Transaction.getCurrent();

// if you need only the base entity interface properties and methods
Address a1 = Address.TYPE.newInstance(b);

// if you need customer extension properties, downcast to the customer extension interface...
AddressExt a2 = (AddressExt) Address.TYPE.newInstance(b);
```

Alternative API for New Instances

Although not recommended for typical use, you can also create a new entity instance with the `newBeanInstance` method on the `Bundle` class as shown in the following example:

```
Bundle b = Transaction.getCurrent();
Address a1 = (Address) b.newBeanInstance(Address.TYPE.get());
```

This approach requires explicit downcasting from the base class of all entity types and thus it is easier to make mistakes that cannot be caught at compile time.

Getting and Setting Entity Properties from Java

Entity properties appear from Java as getter and setter methods. Getter and setter methods are methods to get or set properties with names that start with `get` or `set`. For example, a readable and writable property named `MyField` appears as the methods `getMyField` and `setMyField`. Read-only properties do not expose a `set` method on the object. If the property named `MyField` contains a value of type `Boolean` or `boolean`, it appears in the interface as `isMyField` instead of `getMyField`.

Examples:

```
address.setFirstName("John");
lastName = address.getLastName();
tested = someEntity.isFieldname();
```

If the property is an extension property, you must use the entity extension interface as discussed in “Entity Packages and Customer Extensions from Java” on page 456. Read that topic carefully for the fully-qualified names of the interfaces and a code example of downcasting.

Calling Entity Object Methods from Java

Most entity methods on entity instances appear as regular methods, for example:

```
account.addEvent("MyCustomEventName");
```

In some cases you need to know how the method is declared before you can write the necessary Java code:

- The base entity interface may declare the method, in which case no special downcast is necessary.
- The core extension entity interface may declare the method, in which case downcast to that interface or the customer extension interface. See “Accessing Entity Properties and Methods With Base and Core Extension Interfaces” on page 457.
- The customer extension entity interface may declare the method, in which case you may need to downcast to the customer extension interface. See “Customer Extension Entity Interface” on page 458.
- A Gosu enhancement may declare the method. See “Gosu Enhancement Properties and Methods in Java” on page 467.

Querying for Entity Data in Java

In the Java API, if you need to find entity instances, use the query builder API. See “Overview of the Query Builder APIs” on page 129 in the *Gosu Reference Guide*.

The Java API syntax to those classes is nearly identical to the syntax in Gosu for APIs that do not use Gosu blocks.

For some of the query API that uses Gosu blocks, there is no direct Java equivalent.

For other APIs that use blocks as arguments or return values, there are special Java-specific variants of the methods. Contact Guidewire Customer Support for details.

Accessing Gosu Classes from Java Using Reflection

From Java, you must use *reflection* to access Gosu types. Reflection means asking the type system about types at run time to get data, set data, or invoke methods. Reflection is a powerful way of accessing type information at run time, but is not type safe. For example, language elements such as class names and method names are manipulated as `String` values. Be careful to correctly type the names of classes and methods because the Java compiler cannot validate the names at compile time.

To use reflection from Java or Gosu, use the utility class `gw.lang.reflect.ReflectUtil`. The `ReflectUtil` class contains various APIs to get information about a class, get information about features (properties and methods), and invoke object methods or instance methods.

For example, the following Java code calls a static method on a Gosu class using the `ReflectUtil` method `invokeStaticMethod`.

The Gosu class definition:

```
package test1

class MyGosuClass {
    public static function myMethod(input : String) : String {
        return "MyGosuClass returns the value ${input} as the result!"
    }
}
```

To call the static method from Java with the argument "hello", use the following code:

```
package test1;

import gw.lang.reflect.ReflectUtil;
import gw.transaction.Bundle;

public class MyClass {
    public void doIt() {

        // call a static method on a Gosu class
        String r = (String) ReflectUtil.invokeStaticMethod("test1.MyGosuClass", "myMethod", "hello") ;

        // print the return result
        System.out.print(r);
    }
}
```

If you write your own Gosu class and you want to call methods on it from Java, there is pattern that increases the degree of type safety. See the following procedure.

To make your Java code that calls Gosu classes more typesafe

1. Create a Java interface containing only the set of methods that you need to call from Java. For getting and setting properties, define the interface using getter and setter methods, such as `getMyField` and `setMyField`.
2. Create a Gosu class that implements that interface. It can contain additional methods if desired if they are not needed from Java.

3. In code that needs to get a reference to the object, use `ReflectUtil` to create an instance of the desired Gosu class. There are several approaches:

- Create an instance using `ReflectUtil`.
- Alternatively, define a method on an object that creates an instance. Next, call that method with `ReflectUtil`.

In both cases, at compile time any new object instances returned by `ReflectUtil` have the type `Object`. At run time, downcast to your new Java interface and assign to a new variable of that interface type.

4. You now have an instance of an object that conforms to your interface. Call methods on it as you normally would from Gosu. The getters, setters, and other method names are defined in your interface, so the method calls are typesafe. For example, the Java compiler can protect against misspelled method names.

Gosu Enhancement Properties and Methods in Java

Gosu enhancements are a way of adding properties and methods to a type, even if you do not control the source code to the class. Gosu enhancements are a feature of the Gosu type system. See “Enhancements” on page 235 in the *Gosu Reference Guide*.

Gosu enhancements are not directly available on types from Java.

You can use the `gw.lang.reflect.ReflectUtil` class to call enhancement methods and access enhancement properties. The syntax is more complex than it would be from Gosu. Because it uses reflection, it is less typesafe. There is more chance of errors at run time that were not caught at compile time. For example, if you misspell a method name passed as a `String` value, the Java compiler cannot catch the misspelled method name at compile time.

For more information about reflection and the `gw.lang.reflect.ReflectUtil` class, see “Accessing Gosu Classes from Java Using Reflection” on page 466.

Class Loading and Delegation for non-OSGi Java

Java Class Loading Rules

To load custom Java code into Gosu or to access Java classes from Java code, the Java virtual machine must locate the class file with a *class loader*. Class loaders use the fully-qualified package name of the Java class to determine how to access the class.

BillingCenter follows the rules in the following list to load Java classes, choosing the first rule that matches and then skipping the rules listed after it:

1. General delegation classes

The following classes *delegate load*, which means to delegate class loading to a parent class loader:

- `javax.*` - Java extension classes
- `org.xml.sax.*` - SAX 1 & 2 classes
- `org.w3c.dom.*` - DOM 1 & 2 classes
- `org.apache.xerces.*` - Xerces 1 & 2 classes
- `org.apache.xalan.*` - Xalan classes
- `org.apache.commons.logging.*` - Logging classes used by WebSphere

2. Internal classes

If the class name starts with `com.guidewire`, then BillingCenter delegate loads in general, but there are some internal classes that locally load.

WARNING Java code you deploy must never access any internal classes other than supported classes and documented APIs. Using internal classes is dangerous and unsupported. If in doubt about whether a class is supported, immediately ask Customer Support. Never use classes in the package `com.guidewire`.

3. All your classes

Any remaining non-internal classes load locally.

WARNING Java classes that you deploy must **never** have a fully-qualified package name that starts with `com.guidewire`. Additionally, never rely on classes with that prefix because they are internal and unsupported.

Java Class Delegate Loading

If the BillingCenter class loader delegates Java class loading, BillingCenter requests the parent class loader to load the class, which is the BillingCenter application. If the BillingCenter application cannot find the class, then the ClaimCenter class loader attempts to load the class locally.

Deploying Non-OSGi Java Classes and JARs

The following deployment instructions work for non-OSGi Java class files or JAR files, independent of whether your code uses Guidewire entity data. Carefully deploy Java class files and JAR files in the locations defined in this topic. Putting files in other locations is dangerous and unsupported. If you are deploying an OSGi plugin, see the separate section “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 469.

Note: This section discusses the deployment options for the Java API introduced in version 8.0. If you are using the deprecated Java API from BillingCenter 7, see “Important Changes for Java Code” on page 62 in the *New and Changed Guide*.

Place your non-OSGi Java classes and libraries in the following locations. Any subdirectories must match the package hierarchy. The `shared` directory works for any non-OSGI Java code.

If the class implements a BillingCenter plugin interface, you can define a separate plugin directory in the BillingCenter Studio in the Plugins registry for that interface. In the following paths, `PLUGIN_DIR` represents plugin directory that you define in the Plugins registry. If the `Plugin Directory` field in the Studio Plugins registry is empty, the default is the plugin directory name `shared`. For more about defining plugin directories in BillingCenter Studio, see “Adding an Implementation to a Plugins Registry Item” on page 110 in the *Configuration Guide*. Also see “Registering a Plugin Implementation Class” on page 139.

If any Gosu class calls your Java code, for the `PLUGIN_DIR` value you can also use the value `Gosu` instead of `shared`. Be careful to notice the capitalization of `Gosu`.

Place non-OSGi Java classes in the following locations as the root directory of directories organized by package:

```
BillingCenter/modules/configuration/plugins/shared/basic/classes  
BillingCenter/modules/configuration/plugins/PLUGIN_DIR/basic/classes // for Java plugin code only
```

For example, for a class file with fully qualified name `mycompany.MyClass`, create files at one of the following:

```
BillingCenter/modules/configuration/plugins/shared/basic/classes/mycompany/MyClass.class  
BillingCenter/modules/configuration/plugins/PLUGIN_DIR/basic/classes/mycompany/MyClass.class
```

Place your libraries (JAR files) and any third-party libraries in the following locations:

```
BillingCenter/modules/configuration/plugins/shared/basic/lib  
BillingCenter/modules/configuration/plugins/PLUGIN_DIR/basic/lib // for Java plugin code only
```

OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor

For a summary of the IDE options for OSGi plugin development, see “IDE Options for Plugin Development in the Java Language” on page 453.

This topic describes specific steps for OSGi plugin development:

- “Generate Java API Libraries” on page 469
- “Launch IntelliJ IDEA with OSGi Editor” on page 469
- “Create an OSGi-compliant Class that Implements a Plugin Interface” on page 470
- “Compiling and Installing Your OSGi Plugin as an OSGi Bundle” on page 471
- “Using Third-Party Libraries in Your OSGi Plugin” on page 473

Generate Java API Libraries

Before starting work with Java and OSGi, regenerate the BillingCenter Java API libraries with the `regen-java-api` tool. See “Regenerating Java API Libraries” on page 455. This is a requirement that is independent of which Java IDE that you use.

Launch IntelliJ IDEA with OSGi Editor

For OSGi plugin development, Guidewire recommends that you use the included application IntelliJ IDEA with OSGi Editor. To launch IntelliJ IDEA with OSGi Editor, open a command prompt in the `BillingCenter/bin` directory and type the following:

```
gwbc plugin-studio
```

IMPORTANT If you use other Guidewire applications, such as Guidewire ContactManager, each Guidewire application includes its own version of IntelliJ IDEA with OSGi Editor. Be sure to run this command prompt from the correct application product directory.

Creating a New Project With OSGi Plugin Module

1. If this command is executed for the first time for one Guidewire product, IntelliJ starts with an empty workspace and no current project
2. To create a new project, select **Create New Project**. In the module type list, click **OSGi Plugin Module**. In the **Project name** field, type the project name. In the **Project location** field, type the project location. Do not yet click **Finish**.
Alternatively, if you want to add or import an OSGi module to an existing empty project, in the **Project Structure** dialog, select **Empty Project** and set the **Project name** field. Next, add a module in the **Project Structure** dialog by clicking the plus sign (+). Choose either **New Module** (for new module) or **Import Module** (to select another module to import). For a new module, select the module type **OSGi Plugin Module**. In the **Module name** field, type the module name. Continue to follow the rest of this procedure.
3. Open the **More Settings** pane, which may be initially closed. Set the name of the new module as appropriate. By default, the **Bundle Symbolic Name** field matches the name of the module. You can optionally change the symbolic name to a different value in this dialog. The bundle symbolic name defines the main part of the output JAR file name before the version number. You can also optionally change the version of the bundle in the **Bundle Version** field.
4. Click **Finish**.

5. If this is a new project, you must set the project JDK:
 - a. Click File → Project Structure → Project Settings. In the Project section, set the Project JDK picker to your Java 7 JDK, which might be labelled 1.7.
 - b. If there is no Java 7 JDK listed, click the New... button. Click JDK, then select your Java 7 JDK on your disk. Next, set the Project JDK picker to your newly-created Java 7 JDK configuration.
6. For advanced OSGi settings, see “Advanced OSGi Dependency and Settings Configuration” on page 476.

Create an OSGi-compliant Class that Implements a Plugin Interface

Follow the procedure below to implement an OSGi plugin with IntelliJ IDEA with OSGi Editor.

To implement a new OSGi plugin

1. Remember to regenerate the Java libraries. See “Generate Java API Libraries” on page 469.
2. In IntelliJ IDEA with OSGi Editor, navigate under your OSGi module to the `src` directory.
3. Create new subpackages as necessary. Right-click on `src` and choose **New → Package**. To follow along with an example of a simple startable plugin, create a `mycompany` package to contain your new classes.
4. Right-click on the desired package for your class.
5. Choose **New → New OSGi plugin**. IntelliJ IDEA with OSGi Editor opens a dialog.
6. In the **Plugin class name** field, enter the name of the Java class that you want to create. For our example, type `DemoStartablePlugin`.
7. In the **Plugin interface** field, enter a fully qualified name of the plugin interface you want to implement. Alternatively, to choose from a list, click the ellipsis (...) button. Type some of the name or scroll to the desired plugin interface. Select the desired interface and then click **OK**.
8. IntelliJ IDEA with OSGi Editor displays a dialog **Select Methods to Implement**. By default, all methods are selected. Click **OK**.
9. IntelliJ IDEA with OSGi Editor displays your new class with stub versions of all required methods.
10. If the top of the Java class editor has a yellow banner that says **Project SDK is not defined**, you must set your project SDK to a JDK. See “Creating a New Project With OSGi Plugin Module” on page 469. If the SDK settings are uninitialized or incorrect, your project shows many compilation errors in your new Java class.
11. If you have several tightly-related OSGi plugin implementations, you can optionally deploy them in the same OSGi bundle. If you have additional plugins to implement in this same project, repeat this procedure for each plugin implementation class.
For example, one OSGi bundle can encapsulate messaging code for a messaging destination. A messaging destination may implement the `MessageTransport` interface. The same messaging destination may optionally also implement the `MessageReply` and `MessageRequest` plugin interfaces. If the multiple plugin implementations have shared code and third-party libraries, you could deploy them in the same OSGi bundle.

Example Startable Plugin in Java Using OSGi

The following example defines a class that implements the `IStartablePlugin` interface. The following class simply prints a message to the console for each application call to each plugin method. Use the following example to confirm you can successfully deploy a basic OSGi plugin using IntelliJ IDEA with OSGi Editor.

Create a class with fully-qualified name `mycompany.DemoStartablePlugin` with the following contents:

```
package mycompany;  
  
import aQute.bnd.annotation.component.Activate;  
import aQute.bnd.annotation.component.Component;  
import aQute.bnd.annotation.component.ConfigurationPolicy;
```

```
import aQute.bnd.annotation.component.Deactivate;
import gw.api.startable.IStartablePlugin;
import gw.api.startable.StartablePluginCallbackHandler;
import gw.api.startable.StartablePluginState;

import java.util.Map;

@Component(provide = IStartablePlugin.class, configurationPolicy = ConfigurationPolicy.require)
public class DemoStartablePlugin implements IStartablePlugin {

    private StartablePluginState _state = StartablePluginState.Stopped;

    private void printMessageToGuidewireConsole(String s) {
        System.out.println("*****");
        System.out.println("***** --> STARTABLE PLUGIN METHOD = " + s);
        System.out.println("*****");
    }

    @Activate
    public void activate(Map<String, Object> config) {
        printMessageToGuidewireConsole("activate -- OSGi plugin init");
        // The Map contains the plugin parameters defined in the Plugins registry in Studio
        // There are additional OSGi-specific parameters in this Map if you want them.
    }

    @Deactivate
    public void deactivate() {
        printMessageToGuidewireConsole("deactivate -- OSGi plugin shutdown");
    }

    // Other IStartablePlugin interface methods...

    @Override
    public void start(StartablePluginCallbackHandler startablePluginCallbackHandler,
                      boolean b) throws Exception {
        printMessageToGuidewireConsole("start");
    }

    @Override
    public void stop(boolean b) {
        printMessageToGuidewireConsole("stop");
    }

    @Override
    public StartablePluginState getState() {
        printMessageToGuidewireConsole("getState");
        return _state;
    }
}
```

Compiling and Installing Your OSGi Plugin as an OSGi Bundle

The main script for OSGi compilation and deployment is an Ant build script. See “Advanced OSGi Dependency and Settings Configuration” on page 476.

The Ant build script does the following:

1. compiles Java code
2. generates OSGi metadata
3. packages code and library dependencies into an OSGi bundle
4. installs an OSGi bundle to the correct BillingCenter bundles directory as defined in the OSGi plugin project’s `build.properties` file. For related information, see “Advanced OSGi Dependency and Settings Configuration” on page 476. The script copies your final JAR file to the following BillingCenter directory:
`BillingCenter/modules/configuration/deploy/bundles`

To compile and install your OSGi plugin implementation as an OSGi bundle implementation

1. Open a command prompt in the directory that contains your OSGi plugin module.

2. Type the following command:

```
ant install
```

The command generates messages about compiling source files, generating files, copying files, and building a JAR file. If it succeeds, it prints:

BUILD SUCCESSFUL

3. Switch to or open the regular BillingCenter Studio application, not the IntelliJ IDEA with OSGi Editor. Navigate in the Project pane to the path **configuration → deploy → bundles**. Confirm that you see the newly-deployed file **YOUR_JAR_NAME.jar**. The JAR name is based on the module symbolic name followed by the version.

IMPORTANT If the JAR file is not present at that location, check the Ant console output for the directory path that the script copied the JAR file. If you recently installed a new version of BillingCenter at a different path, or moved your Guidewire product directory, you must immediately update your OSGi settings in your OSGi project. See “Updating Your OSGi Plugin Project After Product Location Changes” on page 477.

4. In BillingCenter Studio (not IntelliJ IDEA with OSGi Editor), register your OSGi plugin implementation. Registering a plugin implementation defines where to find a plugin implementation class and what interface the class implements.
 - a. In the Project window, navigate to **configuration → config → Plugins → registry**. Right-click on the item **registry**, and choose **New → Plugin**.
 - b. In the plugin dialog, enter the plugin name in the **Name** field. For our example, use **DemoStartablePlugin**.
For plugin interfaces that only support one implementation, just enter the interface name without the package. For example, **IStartablePlugin**. The text you enter becomes the basis for the file name that ends in **.gwp**. The **.gwp** file represents one item in the Plugins registry.
If the plugin interface supports multiple implementations, like messaging plugins or startable plugins, this can be any arbitrary name. If you are registering a messaging plugin, the *plugin name* must match the plugin name in fields in the separate **Messaging** editor in Studio.
For our demonstration implementation of the **IStartablePlugin** interface, enter the plugin name **DemoStartablePlugin**.
 - c. In the plugin dialog, next to the **Interface** field, click the ellipsis (...), find the interface class, and select it. If you want to type it, enter the interface name without the package.
For our demonstration implementation of the **IStartablePlugin** interface, find or type the plugin name **IStartablePlugin**.
 - d. Click **OK**.
 - e. In the Plugins registry main pane for that **.gwp** file, click the plus sign (+) and select **Add OSGi Plugin**.
 - f. In the **Service PID** field, type the fully-qualified Java class name for your OSGi implementation class. Note that this is different from the bundle name or the bundle symbolic name. The ellipsis (...) button does not display a picker to find available OSGi classes, so you must type the class name in the field.
For our demonstration implementation of the **IStartablePlugin** interface, type the fully-qualified class name **mycompany.DemoStartablePlugin**.
 - g. Set any other fields that your plugin implementation needs, such as plugin parameters.

IMPORTANT For more information about the Plugins registry, see “Plugin Overview” on page 135

- h. Make whatever other changes you need to make in the regular BillingCenter Studio application. For example, if your plugin is a messaging plugin, configure a new messaging destination. See “Using the Messaging Editor” on page 131 in the *Configuration Guide* and “Messaging and Events” on page 303. You may need to make other changes such as changes to your rule sets or other related Gosu code.
- i. Start the server.

To run the server from Studio, if BillingCenter Studio was already running when you installed or re-installed the bundle, there is an extra required step before running the server. Open a command prompt in the `BillingCenter/bin` directory and type:

```
gwbc dev-deploy
```

Next, start the server from Studio, such as clicking the **Run** or **Debug** menu items or buttons.

Alternatively, run the QuickStart server from the command line. Open a command prompt in the `BillingCenter/bin` directory and type the following:

```
gwbc dev-start
```

An earlier topic showed an example startable plugin. See “Create an OSGi-compliant Class that Implements a Plugin Interface” on page 470. If you used that example, closely read the console messages during server startup. The example OSGi startable plugin prints messages during server startup. This proves that BillingCenter successfully called your OSGi plugin implementation.

Using Third-Party Libraries in Your OSGi Plugin

If your OSGi plugin code uses third-party libraries, there are two deployment options, depending on your type of third-party JAR file:

- **Embed inside your OSGi bundle** – You can embed any Java library in your OSGi plugin bundle. The library is not required to be an OSGi bundle, but OSGi third-party libraries are also supported. Deploy your library JAR in the module directory within the `inline-lib` subdirectory. See “Embed a Third-Party Java Library in your OSGi bundle” on page 473.
- **Deploy as a separate OSGi bundle (requires third-party JAR to support OSGi)** – If your library contains a properly-configured OSGi bundle with OSGi properties in the manifest, you can optionally deploy it as a separate OSGi bundle outside your main OSGi plugin bundle. For use within your OSGi plugin project, deploy your library JAR in the module directory within the `lib` (not `inline-lib`) subdirectory. See “Deploying a Third-party OSGi-compliant Library as a Separate Bundle” on page 473

Deploying a Third-party OSGi-compliant Library as a Separate Bundle

If your library contains a properly-configured OSGi bundle with OSGi properties in the manifest, you can optionally deploy it as a separate OSGi bundle outside your main OSGi plugin bundle.

To use third-party OSGi-compliant libraries separate from your OSGi plugin bundle

1. Put any third-party OSGi JAR files in the `lib` (not `inline-lib`) folder inside your OSGi module in IntelliJ IDEA with OSGi Editor.
2. Write code that uses the third-party JAR.
3. Confirm there are no compile errors.
4. Open a command prompt at the root of your module and type:

```
ant install
```

The tool generates various messages, and concludes with:

```
BUILD SUCCESSFUL
```

The `ant install` script copies your bundle JAR files to the `BillingCenter/deploy/bundles` directory.

Embed a Third-Party Java Library in your OSGi bundle

You can embed any Java library in your OSGi plugin bundle. The library is not required to be an OSGi bundle, but OSGi third-party libraries are also supported. Your bundle manifest may require special configuration for how to import the Java packages that your embedded libraries reference.

For example, suppose you want to use a third-party library that has 100 classes, although you only use 5 of them, and only some of the methods on those classes. If the classes and APIs that you use only rely on available and embedded libraries, there is no problem at run time.

It may be that some of the classes in your third-party library but that you do not use have dependencies on external classes. The library might use an API that relies on classes that are unavailable at run time. OSGi tries to avoid this risk by ensuring at server startup that all required classes exist. Even if you never call those APIs, there will be errors on server startup because OSGi tries to verify all libraries are available, including ones you never called.

Fortunately, you can modify the configuration file to ignore unavailable packages during OSGi library validation phase on server startup. The following instructions include techniques to mitigate these problems.

To use third-party libraries in your OSGi plugin

1. Put any third-party JAR files in the `inline-lib` folder inside your OSGi module in IntelliJ IDEA with OSGi Editor.
2. Write code that uses the third-party JAR.
3. Confirm there are no compile errors.
4. Open a command prompt at the root of your module and type:
`ant dist`
The tool generates various messages, and concludes with:
`BUILD SUCCESSFUL`
5. Open the file `MODULE_ROOT/generated/META-INF/MANIFEST.MF`.
6. In that file, check that the import package (`Import-Package`) header includes no unexpected packages. All classes of the embedded libraries are copied inside your bundle such that all library classes become part of your bundle. This process this might cause the `Bnd` tool to generate unexpected imports to appear in the list.
7. If you see unexpected imports, change the import package instruction in the `bnd.bnd` file, rather than modifying the `MANIFEST.MF` file.

WARNING Never directly change the file `MANIFEST.MF`. It is auto-generated.

In the `bnd.bnd` file, set the `Import-Package` instruction to a comma-delimited list of Java packages to ignore by including each with the exclamation point (!) prefix. The exclamation point means “not” or “exclude”. Finally add an asterisk (*) character as the last item in the list to indicate to import all other packages. For example, to ignore packages `javax.inject` and `sun.misc`, set the line as follows:

`Import-Package=!javax.inject,!sun.misc,*`

8. Deploy your OSGi plugin and test it. Look for server startup errors that look similar to:

```
ERROR Server.OSGi Could not start bundle messaging-OSGi-plugins from reference:  
    file:/D:/GuidewireApp801/webapps/pc/bundles/messaging-OSGi-plugins-1.0.0.jar  
org.osgi.framework.BundleException: The bundle "messaging-OSGi-plugins_1.0.0 [12]" could not  
be resolved. Reason: Missing Constraint: Import-Package: javax.inject; version="0.0.0"
```

If there are errors, repeat this procedure and add more Java packages to the `Import-Package` line in `bnd.bnd`.

For more information about the `bnd.bnd` file, see “Advanced OSGi Dependency and Settings Configuration” on page 476.

Example of Embedding Third-Party Libraries in an OSGi Plugin

Suppose your OSGi plugin requires the Java library called Guava, specifically version 15. First, download `guava-15.0.jar` from the project web site. Next, move the `guava-15.0.jar` file to the `inline-lib` folder within your OSGi project within IntelliJ IDEA with OSGi Editor.

Next, write some Java code that uses this library. For example:

```

import com.google.common.escape.Escaper;
...
// use the class com.google.common.escape.Escaper in guava-15.0.jar
Escaper escaper = XmlEscapers.xmlAttributeEscaper();
s = escaper.escape(s);

```

From a command prompt, run the following command:

```
ant dist
```

That tool generates OSGi metadata and prints various messages. If successful, the final line is:

```
BUILD SUCCESSFUL
```

Open the file MODULE_ROOT/generated/META-INF/MANIFEST.MF and check that Import-Package header does not include any unexpected packages. Our example file might look like the following:

```

Manifest-Version: 1.0
Service-Component: OSGI-INF/com.qa.MessageTransportOSGiImpl.xml
Private-Package: com.google.common.html,com.google.common.net,com.google.common.collect,
    com.google.common.primitives,com.google.common.base,com.google.common.escape,com.qa,
    com.google.common.base.internal,com.google.common.cache,com.google.common.eventbus,
    com.google.common.util.concurrent,com.google.common.hash,com.google.common.io,
    com.google.common.xml,com.google.common.reflect,com.google.common.math,
    com.google.common.annotations
Bundle-Version: 1.0.0
Tool: Bnd-1.50.0
Bundle-Name: messaging-OSGi-plugins
Bnd-LastModified: 1382994235749
Created-By: 1.7.0_45 (Oracle Corporation)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: messaging-OSGi-plugins
Import-Package: gw.pl.messaging.entity,gw.plugin.messaging,javax.annotation,javax.inject,sun.misc
Bundle-RequiredExecutionEnvironment: JavaSE-1.7

```

Note the following line:

```
Import-Package: gw.pl.messaging.entity,gw.plugin.messaging,javax.annotation,javax.inject,sun.misc
```

By default, the packages javax.inject and sun.misc are unavailable at runtime. The package javax.inject is a JavaEE package that is not exported by default. If you install the generated OSGi bundle in its current form and start the server, the server generates the following error:

```

ERROR Server.OSGi Could not start bundle messaging-OSGi-plugins from reference:
    file:/D:/GuidewireApp801/webapps/pc/bundles/messaging-OSGi-plugins-1.0.0.jar
org.osgi.framework.BundleException: The bundle "messaging-OSGi-plugins_1.0.0 [12]" could not
    be resolved. Reason: Missing Constraint: Import-Package: javax.inject; version="0.0.0"

```

In this case, both problematic packages are optional.

In the bnd.bnd file, set the Import-Package instruction to a comma-delimited list of Java packages to ignore by including each with the exclamation point prefix. Finally, add an asterisks(*) character as the last item in the list to indicate to import all other packages. For example, to ignore packages javax.inject and sun.misc, set the line as follows:

```
Import-Package=!javax.inject,!sun.misc,*
```

Run the "ant install" tool again. The scripts embed the guava JAR in your OSGi bundle.

Open the MANIFEST.MF file and notice two changes:

- An additional Ignore-Package instruction.
- The Import-Package instruction does not include the problematic packages.

For example:

```

Manifest-Version: 1.0
Service-Component: OSGI-INF/com.qa.MessageTransportOSGiImpl.xml
Private-Package: com.google.common.html,com.google.common.net,com.google.common.collect,
    com.google.common.primitives,com.google.common.base,com.google.common.escape,com.qa,
    com.google.common.base.internal,com.google.common.cache,com.google.common.eventbus,
    com.google.common.util.concurrent,com.google.common.hash,com.google.common.io,
    com.google.common.xml,com.google.common.reflect,com.google.common.math,
    com.google.common.annotations
Ignore-Package: javax.inject,sun.misc
Tool: Bnd-1.50.0

```

```

Bundle-Name: messaging-OSGi-plugins
Created-By: 1.7.0_45 (Oracle Corporation)
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Bundle-Version: 1.0.0
Bnd-LastModified: 1382995392983
Bundle-ManifestVersion: 2
Import-Package: gw.pl.messaging.entity,gw.plugin.messaging,javax.annotation
Bundle-SymbolicName: messaging-OSGi-plugins

```

With this change, the server now starts up with no OSGi class loading errors.

Advanced OSGi Dependency and Settings Configuration

The IntelliJ IDEA with OSGi Editor application configures dependencies and several additional files related to module configuration and Ant build script. These files are created automatically.

You can edit these files directly in the file system if necessary without harming OSGi module settings. The only file that IntelliJ IDEA with OSGi Editor modifies is `build.properties`, to edit the bundle symbolic name and version.

Dependencies

The IntelliJ IDEA with OSGi Editor auto-creates the following dependencies:

- `PROJECT/MODULE/inline-lib` directory – Directory for third-party Java libraries to embed in your OSGi bundle. See “Embed a Third-Party Java Library in your OSGi bundle” on page 473.
- `PROJECT/MODULE/lib` directory – Directory for third-party OSGi-compliant libraries to use but deploy into a separate OSGi bundle. See “Deploying a Third-party OSGi-compliant Library as a Separate Bundle” on page 473.
- `BillingCenter/java-api/lib` – The BillingCenter Java API files that the `regen-java-api` tool creates.

If you have any JAR files that you require, put them in the `lib` or `inline-lib` directories, as specified earlier. You do not need to modify any build scripts to add JAR files.

If you add dependencies in the IntelliJ project from other directories, change the associated `build.xml` (Ant build script) to include those directories.

Build Properties

The `build.properties` file at the root directory of the module contains the bundle symbolic name, version, and the path to Guidewire product. From within IntelliJ IDEA with OSGi Editor, you can edit these fields but optionally you can directly modify this file as needed. The following is an example of this file:

```

Bundle-SymbolicName=messaging-OSGi-plugins
Bundle-Version=1.0.0
-gw-dist-directory=C:/BillingCenter/

```

The Ant build script (`build.xml`) uses these properties.

OSGi Bundle Metadata Configuration File

The OSGi bundle metadata configuration file is called `bnd.bnd`. The contents of this file configures how scripts in the IntelliJ IDEA with OSGi Editor application generate OSGi bundle metadata, such as the `MANIFEST.MF` file and other descriptor files.

By default, this file looks like:

```

Import-Package=*
DynamicImport-Package=
Export-Package=
Service-Component=*
Bundle-DocURL=
Bundle-License=
Bundle-Vendor=
Bundle-RequiredExecutionEnvironment=JavaSE-1.7
-consumer-policy=${range;[==,+)}

```

```
-include=build.properties
```

For the format of this file, see:

<http://www.aqute.biz/Bnd/Format>

However, for the `Bundle-SymbolicName` and `Bundle-Version` properties, you must set or change those settings in the `build.properties` file. Both the `bnd.bnd` and `build.xml` file use those properties from the `build.properties` file.

You may need to edit this file if you want to export some Java packages from your bundle, or you want to customize the `Import-Package` header generation.

Build Configuration Ant Script

The build configuration Ant script (`build.xml`) relies on properties from the files `build.properties` and `bnd.bnd`.

For advanced OSGi configuration, you can modify the `build.xml` build script.

Updating Your OSGi Plugin Project After Product Location Changes

After you initially configure a project within IntelliJ IDEA with OSGi Editor, the project includes information that references the Guidewire product directory on your local disk. If you move your product directory, you must update your project with the new product location on your local disk. Similarly, if you upgrade your Guidewire product to a new version with a different install path, you must update the OSGi project.

To update your OSGi project after Product Location Changes

1. If you need to move your OSGi project on disk, quit IntelliJ IDEA with OSGi Editor and move the files on disk.
2. Launch IntelliJ IDEA with OSGi Editor from your new Guidewire product location. See “Launch IntelliJ IDEA with OSGi Editor” on page 469.
3. In IntelliJ IDEA with OSGi Editor, open your OSGi plugin project.
4. In IntelliJ IDEA with OSGi Editor, update the module dependency for your OSGi module:
 - a. Open the Project Structure window.
 - b. Click the **Modules** item in the left navigation.
 - c. In the list of modules to the right, under the name of your module, click **OSGi Bundle Facet**.
 - d. To the right of the **Guidewire product directory** text field, click the **Change** button.
 - e. Set the new disk path and click **OK**.
 - f. Click **OK** in the dialog. The tool updates IDE library dependencies and the `build.properties` file.
5. Test your new configuration. See “Compiling and Installing Your OSGi Plugin as an OSGi Bundle” on page 471.

