Guidewire ClaimCenter ™

Contact Management Guide

Release: 10.2.4

GUIDEWIRE

Product Name: Guidewire ContactManager
Product Release: 10.2.4
Document Name: Contact Management Guide
Document Revision: 23-August-2025

Contents

Support **9**

1 Managing integrated contacts **11**
Client Data Management 11
PolicyCenter role in managing client data 12
ClaimCenter role in managing client data 12
BillingCenter role in managing client data 13
Vendor Data Management 13
Vendor contacts 13
Using and configuring vendor data management 13
Overview of ClaimCenter integration with ContactManager 13

2 Installing ContactManager **17**
Installing ContactManager with QuickStart for development 17
Install ContactManager with QuickStart 18
Load sample data for ContactManager 18

3 Integrating ContactManager with Guidewire core applications **21**
Integrating ContactManager using QuickStart 21
Integrating ContactManager with ClaimCenter in QuickStart 21
Integrating ContactManager with PolicyCenter in QuickStart 28
Integrating ContactManager with BillingCenter in QuickStart 35
Configuring core application authentication with ContactManager 41
Configure ClaimCenter-to-ContactManager authentication 42
Configure PolicyCenter-to-ContactManager authentication 43
Configure BillingCenter-to-ContactManager authentication 44
Configuring ContactManager authentication with core applications 45
Overview of ContactManager authentication configuration 46
Configure ContactManager-to-ClaimCenter authentication 46
Configure ContactManager-to-PolicyCenter authentication 48
Configure ContactManager-to-BillingCenter authentication 51

4 Searching for contacts **55**
Overview of contact search 55
ContactManager support for contact searches 56
Local ContactManager search criteria 56
Defining minimum search criteria for a contact 56
ContactManager support for core application searches 58
ClaimCenter support for contact searches 58
Adding the InterpreterSpecialty property to search 59
Adding an address field to Contact search 65
Adding the service state property to search 67
Geocoding and proximity search for vendor contacts 73
Geocoding and batch processing 73
Configuring geocoding for ClaimCenter and ContactManager 75
Proximity search example for vendor contacts 80
PolicyCenter support for Contact searches 81

BillingCenter support for Contact searches....82

5 Securing access to contact information....85

ContactManager contact security....85

ContactManager user roles....86

ContactManager contact subtype and tag permissions....89

ContactManager contact and tag permission check expressions....90

Contact search security configuration parameters in ContactManager....92

Configuring ContactManager contact security....92

PolicyCenter contact security....96

BillingCenter contact security....97

BillingCenter contact and tag permission check expressions....97

Build and view the BillingCenter Security Dictionary....98

Viewing permissions in the BillingCenter Security Dictionary....98

ClaimCenter contact security....99

ClaimCenter contact roles....99

ClaimCenter contact subtype and tag permissions....99

ClaimCenter contact and tag permission check expressions....100

Viewing permissions in the ClaimCenter Security Dictionary....101

Build and view the ClaimCenter Security Dictionary....102

Contact permission search configuration parameters in ClaimCenter....102

Configuring ClaimCenter contact security....102

6 Extending the contact data model....109

Overview of contact entities....109

ABContact data model....109

Contact data model....111

ContactManager and core application contact entity hierarchies....113

General guidelines for extending contacts....115

Deciding whether to create a subtype....115

Refresh applications after contact data model changes....116

Limitations on configuring contact entity extensions....116

Working with contact mapping files....116

Extending the client data model....118

Extending the vendor contact data model....118

Adding a field to a contact subtype....118

Adding a vendor contact subtype....125

Extending contacts with an array....138

Changing the subtype of a Contact instance....152

Changing subtype with a command-prompt utility....152

Restrictions on changing the subtype of a Contact instance....153

Changing the subtype of a contact instance....155

Troubleshooting the change subtype command-prompt utility....157

7 Contact tags....159

Applications and contact tags....159

Add a new contact tag....160

ClaimCenter contact tag generation....160

PolicyCenter contact tag generation....160

BillingCenter contact tag generation....160

Contact tag-based security....160

8 Vendor services....163

Working with vendor services in ContactManager..................164
Importing and exporting vendor service data..................164
Viewing, adding, and removing vendor service data for contacts..................165
Making changes to vendor services..................166
Vendor services data model..................170
Vendor services directory data model..................171
SpecialistService XML syntax..................171
ABContact and SpecialistService entity relationships..................173
Specialist services directory example..................173

9 Vendor document management..................177
Overview of vendor documents..................177
Document storage in the base configuration..................177
Document metadata properties..................178
Working with vendor documents..................179
List vendor documents in ContactManager..................179
Working with the documents card..................179
Create a new vendor document..................180
Uploading documents..................180
Indicate the existence of a hard-copy vendor document..................181
Edit content for a vendor document..................182
Edit metadata properties of a vendor document..................182
Hide a vendor document..................182
Remove a vendor document link..................183
Configuring vendor document management..................183
Document security..................184
Setting document configuration parameters and MIME types..................184
ABContact and Document entity relationships..................185
Working with vendor document PCF files..................186
Vendor document classes..................188
Vendor document management integration..................188
Vendor document integration with a document management system..................188
Vendor document integration with ClaimCenter..................190

10 Linking and synchronizing contacts..................193
Linking a contact..................193
Creating and linking a contact..................193
Linking in ClaimCenter..................195
Linking in PolicyCenter..................196
Linking in BillingCenter..................196
Find duplicates behavior..................197
Synchronizing with ContactManager..................198
ContactManager plugins for broadcasting of contact changes..................198
Synchronizing PolicyCenter and ContactManager contacts..................198
Synchronizing BillingCenter and ContactManager contacts..................200
Synchronizing ClaimCenter and ContactManager contacts..................201

11 ContactManager rules..................207
ContactManager rule sets..................207
EventMessage rule set category..................207
Preupdate rule set category..................210
Validation rule set category..................211

Overview of ContactManager preupdate and validation.......211
Overview of ContactManager validation.......212
Validation graph in ContactManager.......212
Top-level ContactManager entities that trigger validation.......213

12 Configuring personal data destruction in ContactManager.......215
Overview of data destruction.......215
Data destruction configuration parameters.......217
Data destruction web service.......217
PersonalDataDestructionAPI web service methods.......218
Lifecycle of a personal data destruction request.......219
Personal data destruction request entities.......220
Example of a request made with AddressBookUID.......220
Example of a request made with PublicID.......220
Typelists for status of personal destruction workflow.......220
Work queues used in personal data destruction.......221
PersonalDataDestructionController class.......222
Data model configuration for data destruction.......223
DestructionRootPinnable delegate.......223
Do Not Process flag.......223
Display keys for data destruction messages.......223
Obfuscatable delegate.......224
Obfuscator interface.......224
Marking entity fields for obfuscation.......225
ContactManager entity domain graph.......225
Overview of the ContactManager entity domain graph.......225
ContactManager ABContact entity domain graph.......226
ContactManager Data Protection Officer.......229
Data destruction purge configuration.......229
Defining the Destroyer.......230
PersonalDataPurge event.......231
Specifying which objects in the graph can be destroyed.......231
Data obfuscation in ContactManager.......233
Implementing the Obfuscatable delegate in an entity.......233
Implementing the Obfuscator interface in an entity.......233
Personal data obfuscation classes.......234

13 ClaimCenter service provider performance reviews.......237
Service providers.......238
Reviews of service providers.......238
Using service provider performance reviews.......238
Working with reviews.......238
Scheduling service provider performance reviews for processing.......241
Manually processing service provider performance reviews.......243
Data model for service provider performance reviews.......244
ClaimCenter service provider performance review entities.......244
Typelists for service provider performance reviews.......246
ContactManager service provider performance review data model.......246
Configure service provider performance reviews.......247
Export questions and review types.......247
Add a service provider performance review type.......247
Retire a Service Provider Performance Review Type.......248

Import questions and review types....249
Configuring categories for service provider performance reviews.... 249
Configuring review service type and review category typecodes.... 250
Configuring question sets, questions, choices, and filters.... 253
Overview of service provider performance review scoring.... 256
Service provider performance review permissions....257
Service provider performance review plugins....258
ClaimCenter service provider performance review plugin....258
Register a vendor performance review plugin class in ClaimCenter.... 258
ContactManager contact scoring plugin for service providers....259
Register a contact scoring plugin class in ContactManager.... 260

14 Working directly in ContactManager....261
Log in to ContactManager....261
Change your password in ContactManager.... 262
Change your user preferences in ContactManager....262
Preferences worksheet....262
Selecting international settings in ContactManager.... 263
Changing user interface settings in ContactManager.... 263
Changing the screen layout....263
Data entry support for ContactManager input fields....263
ContactManager user interface.... 264
Contacts tab.... 265
Administration tab.... 265
Internal Tools and Server Tools tabs....266
International settings in ContactManager....266
Importing and exporting administrative data.... 267
Import administrative data and vendor service trees 268
Exporting data in the administration tab.... 268
Importing and exporting either with Gosu or from the command prompt.... 269
Using inbound files integration.... 269
Using outbound files integration....270
Managing contact data.... 270
Detecting and merging duplicate contacts....271
Review pending changes to contacts.... 278
Working with work queues and batch processes....279
Vendor services onboarding....281
Initial configuration for vendor services onboarding.... 282
Working with vendor services onboarding.... 285

15 ContactManager integration reference....297
Overview of ContactManager integration....297
ContactManager entities....298
ContactManager link IDs and comparison to other IDs.... 299
ContactManager web services.... 299
Web services provided by ContactManager....300
Support classes for ContactManager web services....301
ABContactAPI web service.... 303
ABVendorEvaluationAPI web service.... 307
ContactManager messaging events.... 308
ContactManager messaging events by entity.... 308
ABContact message safe ordering....310

    ABContact message resynchronization.... 310
ABClientAPI interface.... 310
    Method signatures of the ABClientAPI interface.... 311
    Deleting contacts in ClaimCenter.... 312
ContactMapper class.... 313
    ContactManager ContactMapper class.... 314
    Core application ContactMapper class.... 317
Overview of ContactManager plugins.... 322
    Register a plugin implementation in ContactManager.... 322
    ContactManager plugins.... 323
IFindDuplicatesPlugin plugin interface.... 328
    Duplicate finder classes.... 329
    Query builder classes.... 334
ValidateABContactCreationPlugin plugin interface.... 337
    Contact minimum creation requirements.... 337
    Verify Minimum Criteria work queue.... 338
    Configure the TaxID contact creation requirement.... 338
    Code example for creating ABContact and ABCompany contacts.... 339
    Configuring validation error messages for contact creation.... 339
Testing clock plugin interface.... 339

Support

For assistance, visit the Guidewire Community.

Guidewire customers

`https://community.guidewire.com`

Guidewire partners

`https://partner.guidewire.com`

chapter 1

Managing integrated contacts

Depending on your core application, you can integrate ContactManager to manage vendor contacts or client contacts or both. You typically use your Guidewire core application to work with contacts, although you can log in to ContactManager and work with them there.

See also

- “Client Data Management” on page 11
- “Vendor Data Management” on page 13

Client Data Management

Client Data Management, available as an optional add-on module for Guidewire core applications, integrates and synchronizes client-related information across InsuranceSuite. Client Data Management supports a comprehensive view of the customer across core insurance processes by unifying the customer record. Some Client Data Management capabilities are supported by ContactManager.

In conjunction with Guidewire InsuranceSuite core systems, Client Data Management enables you to manage customer contact data across underwriting, policy administration, billing, and claims processes. If you license the Client Data Management add-on module, you can install ContactManager and integrate it with PolicyCenter and with other InsuranceSuite core applications. With the applications integrated, Client Data Management supports your customer and agent interactions and can enable you to maintain data integrity across different systems.

ContactManager is designed to be your system of record, the central repository for your customer data. This centralized repository stores a unique customer record and synchronizes updates across your policy, billing, and claims systems.

Various types of users, such as billing specialists, underwriters, and agents, can edit customer information in their respective applications without having to navigate to another system. You can add new clients through PolicyCenter, see current client information for policies in ClaimCenter, make updates directly in ClaimCenter, and obtain the latest client data in BillingCenter. To aid data cleanup, duplicate contact detection identifies potential duplicate contacts resulting from creating new contacts and from changing existing contacts, and enables you to merge or override these contacts.

ContactManager’s comprehensive customer view, available in PolicyCenter, enables service representatives to personalize interactions to match customers' needs. The search and auto-fill features support updating of contact records, and contact history tracks which updates have been made and provides visibility into previous customer interactions. Additionally, the PolicyCenter comprehensive view enables underwriters to find a customer’s policy, billing, and claim information in a single place. Underwriters no longer have to search for this information in multiple systems. And, to ensure that the proper controls are in place, user permissions dictate who can update contact information at various points in the lifecycle.

PolicyCenter role in managing client data

With the Client Data Management module, after integrating PolicyCenter with ContactManager, PolicyCenter can store client data in ContactManager. Typically, you create a new client when you open an account or create a policy, or when you add contacts to an account or policy. Additionally, in the PolicyCenter **Contact** tab, you can view information associated with a client, such as personal details like address, date of birth, and phone numbers. On the **Account** tab, you can view contact information such as all accounts, policies, and work orders associated with the client. You can also see claims and billing information if you use ClaimCenter and BillingCenter with PolicyCenter or if you integrate this feature with another claim system or billing system.

Note: While you can create a new client when you open an account or a new policy, the client data is not immediately stored in ContactManager. Contact data is stored in ContactManager only when you bind a policy or when you associate a contact with an account that has a bound policy. Additionally, contacts associated with reinsurance treaties or programs are also stored in ContactManager.

Using the **Contact** tab, you can create new contacts, search for existing contacts, select a recently viewed contact, and change contact information. PolicyCenter additionally enables you to work with contacts in its **Account** and **Policy** screens, where you can add, remove, and update contacts in various roles.

All these contacts are stored locally with the associated account or policy. Contacts that were originally in ContactManager and contacts in accounts with at least one bound policy are linked to a central record in ContactManager. If you make changes to a linked contact, PolicyCenter saves these changes locally and to the central contact record in ContactManager.

ContactManager also notifies PolicyCenter of any changes made to linked contacts outside PolicyCenter, such as a change originating in ClaimCenter when the client makes a claim. These notifications keep a contact’s records synchronized across all accounts, policies, work orders, and so on.

PolicyCenter linked addresses and side effects

PolicyCenter has a feature that enables you to share the same address with more than one contact. When you share an address across contacts in PolicyCenter, the contact addresses continue to be stored separately, but PolicyCenter links them. However, in the other Guidewire applications, contact addresses are not linked. ContactManager stores each contact address as a separate, unlinked address.

If you make a change to one of these addresses from ClaimCenter, BillingCenter, or ContactManager, there is a side effect in the base configurations. When ContactManager sends the changed contact address to PolicyCenter, PolicyCenter applies the change to the other addresses to which the original changed address is linked. PolicyCenter then sends these changed contact addresses back to ContactManager, and the result is that more than one contact gets an address change. There is no indication in ClaimCenter, BillingCenter, or ContactManager of this side effect—the linked addresses are changed automatically. You can configure the applications to behave differently.

If you change a linked address in PolicyCenter, you have the option of linking or unlinking it from other contacts’ addresses. However, since this feature exists only in PolicyCenter, you do not have this option in the other Guidewire applications.

ClaimCenter role in managing client data

ClaimCenter can receive client data from PolicyCenter when ClaimCenter requests policy information during claims processing. ClaimCenter also can receive client data updates from ContactManager when the information for a client stored in ContactManager changes. For example, a client might notify a PolicyCenter customer service representative that their address has changed. The customer service representative updates the client’s address, and PolicyCenter sends the change to ContactManager, which subsequently sends the change to ClaimCenter.

Additionally, a client might notify a ClaimCenter customer service representative that their information has changed, such a change of phone number or address. The ClaimCenter customer service representative can change that information for the client in a claim, and then ClaimCenter can send the change to ContactManager. ContactManager then broadcasts the change to any Guidewire application that is integrated with ContactManager, such as PolicyCenter.

BillingCenter role in managing client data

BillingCenter can receive client data from PolicyCenter when PolicyCenter sends billing information to BillingCenter. BillingCenter also can receive client data updates from ContactManager when the information for a client stored in ContactManager changes. For example, a client might notify a PolicyCenter customer service representative that their address has changed. The customer service representative updates the client's address, and PolicyCenter sends the change to ContactManager, which subsequently sends the change to BillingCenter.

Additionally, BillingCenter can send changes in client data to ContactManager. For example, a customer contacts a billing clerk to report a change of address.

Vendor Data Management

Vendor data management, available only in ClaimCenter, is typically a set of tasks related to adding, modifying, searching for, and deleting vendor contacts. The contacts are typically managed in a contact management system, like Guidewire ContactManager. A vendor contact is a person or company that provides services for claims.

Vendor contacts

A vendor contact is a person or company that provides services for claims. In ClaimCenter, a vendor contact can be a person like a doctor or attorney. Additionally, a vendor contact can be a company, such as a repair shop, a bank, or a hospital. Additionally, a vendor contact can be a specific place or venue for which your company frequently references the geographical location, such as a legal venue like a court house.

Using and configuring vendor data management

Vendor data management is typically a set of tasks related to adding, modifying, searching for, and deleting vendor contacts in a contact management system, like ContactManager. ClaimCenter provides various opportunities for you to create, edit, or search for contacts. For example, when entering a new claim, you can create a new vendor contact or search for a contact, such as an attorney, in the claim creation wizard. You can also create and search for contacts in other areas of the ClaimCenter application where you need to specify a contact for a claim.

If ContactManager is integrated with ClaimCenter, you can define a set of services to be associated with vendor contacts. See "Vendor services" on page 163.

You can configure screens to match company requirements, such as creating and searching for new kinds of contacts, and you can create entire new screens if needed. Or you can perform configurations like validating that an entered postal code is in the correct format.

Overview of ClaimCenter integration with ContactManager

ContactManager is a separate, stand-alone Guidewire application. You can use ContactManager as a central repository for standardizing and managing the contact data for your company. Typically, a company uses ContactManager in conjunction with ClaimCenter to manage vendor contacts that can be used in more than one claim. A claim can also have contacts that are associated only with that one claim, such as a claimant. These contacts can be tracked locally in ClaimCenter.

Centralized vendor data management

As the system of record for vendor contacts, ContactManager enables you to manage and serve vendor contact data centrally. ClaimCenter enables you to search for these centrally-maintained contacts on the **Address Book** tab and from claims. In claims, you can add contacts and edit the contact data as needed, and your changes can be saved in ContactManager.

After you integrate ContactManager and ClaimCenter, users searching for contacts have access not only to contacts stored locally in ClaimCenter, but also to contacts stored in ContactManager. For example, an adjuster working in ClaimCenter can search for an auto shop for a repair and access a list of approved service providers provided by ContactManager.

You can define different users for each Guidewire application. For example, you can have a group of ContactManager users whose primary function is to manage contact data and ensure that it is accurate. These users need not have authorization for ClaimCenter.

A synchronizing facility between the two applications ensures that when a contact changes, regardless of whether the change occurred in ContactManager or ClaimCenter, the change appears in both applications.

Which contacts to store in ContactManager

You need not configure ClaimCenter and ContactManager to store all company contacts in ContactManager. In some cases, it makes sense to centrally manage a contact, and in others it does not. ClaimCenter enables you to manage contacts both centrally in ContactManager and locally in ClaimCenter.

Typically, you store in ContactManager either contacts that you expect to use across multiple claims or contacts that require a single, definitive information source. For example, you would store service providers, vendors, such as doctors, auto repair shops, and inspectors in ContactManager. These contacts require correct tax ID data for reporting and accurate addresses for payments.

Contacts that are specific to a claim are best managed locally in ClaimCenter. These contacts appear only in specific instances and are unlikely to reappear. Additionally, they have no impact on your company's business processes, nor do they have any regulatory requirements. Two examples might be an accident witness on a claim and a bank employee who requests a policy verification.

Locally and centrally managed contacts

When you create a new contact, depending on what kind of contact it is, the contact can be stored in two ways. ClaimCenter can store the contact locally only in ClaimCenter. Or, ClaimCenter can store the contact locally in ClaimCenter and centrally in ContactManager and link the contacts. If you create a new, non-vendor contact directly on a claim, ClaimCenter can store it locally and not in ContactManager. This contact, associated only with the claim, is an *unlinked* contact.

Unlinked contacts are not associated with ContactManager. For an unlinked contact, ClaimCenter does not attempt to determine if a contact associated with one claim appears elsewhere on another claim. Thus, any unlinked contact can be a duplicate of one or more other unlinked contacts associated with different claims. When such duplicates exist, changes to one contact do not propagate to another because they are not related to one another.

Conversely, contacts stored in ContactManager, such as vendor contacts, do propagate changes when stored in ClaimCenter. In ClaimCenter, you can associate a single, centrally-managed contact with any number of claims. ClaimCenter makes a copy of each contact and stores it locally with the claim, and each copy is also linked to the ContactManager contact. Therefore, ClaimCenter can keep the data for all these copies of a contact in sync, even though there are multiple copies stored in ClaimCenter.

You can add a new contact and make the contact centrally managed and available for use with other claims. If the new contact is a vendor, it is sent automatically to ContactManager and does not require any other action in ClaimCenter to become linked.

Note: The contact might require action in ContactManager, however. Under certain circumstances, such as you not being a user with permissions to create contacts, ClaimCenter sends the contact as a pending create. The contact then requires verification from a user in ContactManager, as described later.

For example, from a contact management perspective, a claimant might be a contact that other adjusters do not reuse. Therefore, a claimant might be an unlinked contact, stored locally with the claim but not in ContactManager. Even so, you could add a linked, centrally managed claimant to a claim. For example, you could click **Add an Existing Contact** and find the contact in ContactManager database and then add that contact to the claim.

To link a contact, ClaimCenter and ContactManager define a connection between a specific record in the ClaimCenter database and another record in the ContactManager database. In effect, linking the two contacts declares that though they are in two different databases, they are the same and are to show the same information. Users of ClaimCenter can link contacts from ClaimCenter to ContactManager. ClaimCenter can also automatically link a vendor contact to ContactManager. From ContactManager, users cannot link contacts to ClaimCenter.

After a contact is linked, the contact can be updated from either application, and both applications can get the update. ContactManager notifies ClaimCenter of changes to linked contacts. A linked contact in ClaimCenter whose data has changed in ContactManager is marked as not in sync. To keep contacts in ClaimCenter current, you can copy the data for a linked contact from ContactManager. Copying causes ClaimCenter to copy the latest information from the ContactManager central repository and mark the contact as in sync.

When you update a contact in ClaimCenter that is linked to ContactManager, ClaimCenter sends the change to ContactManager. For example, you can edit a linked contact on the **Contacts** screen of a claim and then update the contact. If you have permission to edit ContactManager contacts, ClaimCenter instructs ContactManager to save the change. If you do not have this permission, ClaimCenter instructs ContactManager to create a pending change, which then has to be reviewed by a ContactManager user.

You can unlink a contact. An unlinked contact in ClaimCenter becomes a claim-specific contact and is no longer centrally managed. However, even though the contact is no longer linked to ContactManager, a contact that you unlink continues to be in the ContactManager database.

See also

- "Linking and synchronizing contacts" on page 193

Deciding whether to use ContactManager for vendor management

In the base configuration, ClaimCenter has a registered address book plugin that is useful only for demonstration purposes. Guidewire also supplies a fully functional Gosu plugin implementation that works with ContactManager and is suitable for production. Do not go into a production environment with the demonstration plugin registered. If you decide not to use the integration with ContactManager but instead want to integrate with another, third-party contact system, you must implement a replacement plugin.

You are not required to use either ContactManager or your own contact plugin in ClaimCenter. Without a contact plugin, ClaimCenter continues to maintain its own local set of contacts in its own database, but its address book functionality changes as follows:

- The **Address Book** tab is present, but attempts to use it for searching result in an error.
- The claim **Parties Involved** screen has reduced functionality. You are unable to view the **Address Book** from the screen or copy a contact or add an existing contact from ContactManager.

The **Address Book** tab, buttons, and other related options are always present in the base application. If you want to remove them, you must edit the related PCF files.

> **Note:** Even without having a contact management application integrated, you can use the **Search** tab to search for local contacts. However, without this integration, you cannot check for duplicate contacts.

See also

- "Client Data Management" on page 11
- "Integrating ContactManager with Guidewire core applications" on page 21
- For information on how to create your own plugin, see the *Integration Guide*

chapter 2

Installing ContactManager

You can install ContactManager in a development environment. The installation process is similar to the process described in the *Installation Guide*.

You can install ContactManager for development by using the Guidewire QuickStart installation setup, or you can use any supported application and database server. Use QuickStart only in a demonstration or development environment.

ContactManager must start with its own application and database server. Do not run ContactManager in the same server as a core application.

For production, you need a dedicated Java Virtual Machine (JVM) for each Guidewire application. Running a Guidewire core application and ContactManager in the same JVM can result in memory conflicts and other problems.

For production, you can install ContactManager with any supported combination of application server and database server. For more information, visit the Guidewire Community and search for knowledge article 1005, "Supported Software Components".

Installing ContactManager with QuickStart for development

A QuickStart installation enables you to immediately use and test the product. The QuickStart installation topics provide only the information necessary to get ContactManager working in the QuickStart environment.

See also

- For general information on QuickStart, see "Installing the QuickStart development environment" in the *Installation Guide*.
- For information on setting the port number for the Jetty server used in QuickStart, see "Configuring QuickStart ports" in the *Installation Guide*.
- For information on configuring the H2 Database Engine used in QuickStart, see "Using the QuickStart database" in the *Installation Guide*.
- For information on using SQL Server or Oracle in the QuickStart environment, see "Using SQL Server or Oracle in a development environment" in the *Installation Guide*.

Install ContactManager with QuickStart

You can install ContactManager and use the provided H2 database for development or demonstration purposes.

Before you begin

These instructions assume that you have installed your Guidewire core application as described in the *Installation Guide*.

Procedure

1. If you have not already done so, create an installation directory for ContactManager.

   This guide uses `ContactManager` as the directory name.
2. Extract the ContactManager Zip file into the ContactManager installation folder.
3. If you are not using a version control system, make a read-only copy of the ContactManager installation folder.

   This copy enables you to recover quickly from accidental changes that can prevent ContactManager from starting.
4. If you are reinstalling ContactManager, drop the QuickStart database created by the previous installation.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb dropDb
   ```
5. At the command prompt, enter the following command:

   ```
   gwb runServer
   ```

   When the server has started, you see the message `***** ContactManager ready *****` in the console messages.
6. Open a browser and enter the URL `http://localhost:8280/ab`, and then log in with user name `su` and password `gw`.

What to do next

"Load sample data for ContactManager" on page 18

Load sample data for ContactManager

ContactManager provides sample data that you can load so you can test application functionality.

Before you begin

Install ContactManager as described at "Install ContactManager with QuickStart" on page 18.

About this task

Sample data can be useful for learning purposes and for completing integration examples, but it is not intended to be used in a production system. Additionally, importing sample data can make changes that you must remove before production.

Procedure

1. Start ContactManager and log in with user name `su` and password `gw`.
   a) At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb runServer
   ```

  b) Open a browser and enter the URL `http://localhost:8280/ab`.
  c) Log in with user name `su` and password `gw`.
2. Press `Alt+Shift+T`.
3. Click the **Internal Tools** tab.
4. Click **AB Sample Data**, and then click the **Load Sample Data** button.
5. Wait to see the completion messages above the button confirming that the sample data was imported.
6. To verify that the data loaded correctly:
   a) On the Options menu ⚙, click **Return to ContactManager**.
   b) Click the **Administration** tab.
   c) In the Sidebar on the left, you see **Default Root Group**. If necessary, click **Default Root Group** to show its contents, and then click **Enigma Fire and Casualty** to see the list of users that was just loaded as sample data.
7. To log in to ContactManager as a user with permission to add, edit, and delete contacts:
   a) On the Options menu ⚙, click **Log Out**.
   b) Log in with user name `aapplegate` and password `gw`.

chapter 3

Integrating ContactManager with Guidewire core applications

You can integrate ContactManager with the Guidewire core applications ClaimCenter, PolicyCenter, and BillingCenter. The Guidewire core applications communicate with ContactManager through SOAP APIs. The applications work with ContactManager through its published web service `ABContactAPI`.

Each Guidewire core application defines a plugin interface for invoking the integration layer to communicate with an external contact management system. In each application, there is a plugin class implementation of this interface—a class that implements this interface—that communicates with ContactManager. The plugin interface and the plugin class implementation are different for each Guidewire core application.

You configure the integration between ContactManager and the Guidewire core applications in Guidewire Studio™.

See also

- "Overview of ContactManager plugins" on page 322

Integrating ContactManager using QuickStart

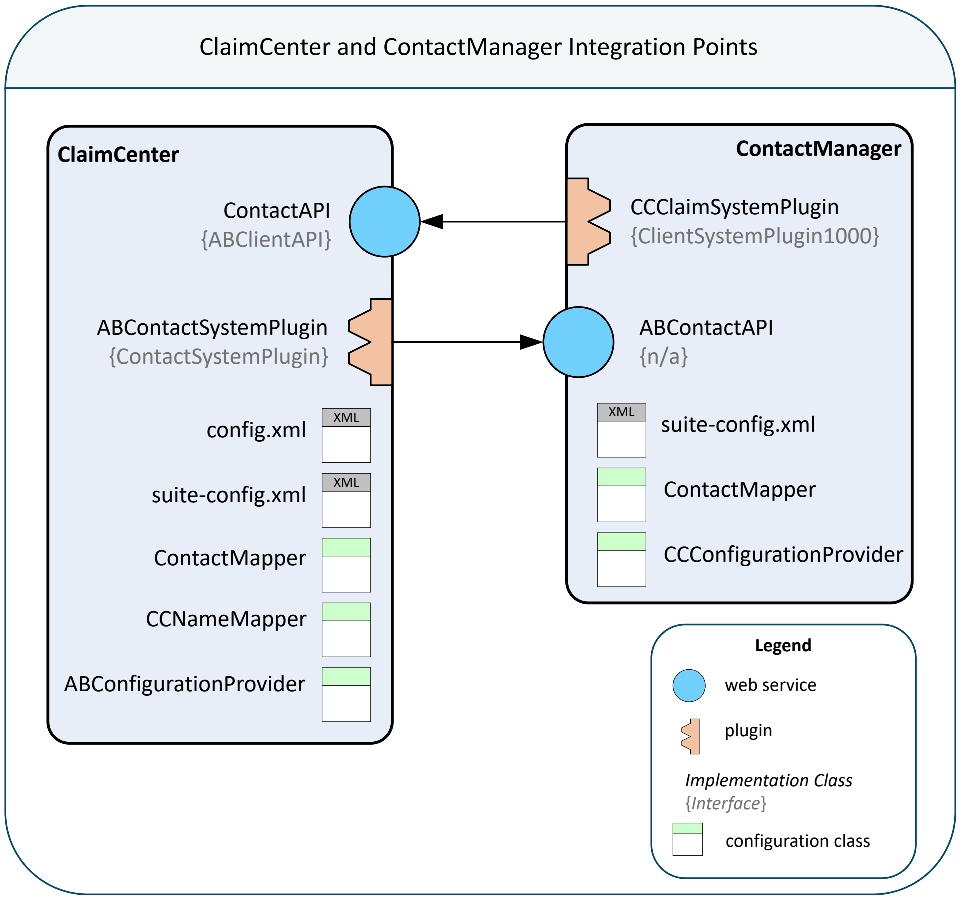
A QuickStart installation enables you to quickly use and test the product. After installing ContactManager, you integrate it with Guidewire core applications in a QuickStart environment.

See also

- "Installing ContactManager with QuickStart for development" on page 17
- For details on the `suite-config.xml` file, see the *Integration Guide*.

Integrating ContactManager with ClaimCenter in QuickStart

The following figure shows the primary integration points used to integrate ClaimCenter and ContactManager.



In general, to get the basic integration working, you edit the `suite-config.xml` files in ClaimCenter and ContactManager and the `config.xml` file in ClaimCenter. You also register the `CCClaimSystemPlugin` plugin implementation in the ContactManager plugin registry `ClaimSystemPlugin.gwp`. Additionally, you register `ABContactSystemPlugin` in the ClaimCenter registry `ContactSystemPlugin.gwp`. All these steps are described in the topics that follow.

If you extend the contact model, you must edit the `ContactMapper` classes in both ContactManager and ClaimCenter, and possibly the `CCNameMapper` class.

IMPORTANT: Guidewire recommends that you change the authentication user and password. To do so, you add authentication users to ContactManager and ClaimCenter and edit the `ABConfigurationProvider` and `CCConfigurationProvider` classes.

If you extend the contact model, you must edit the `ContactMapper` classes in both ContactManager and ClaimCenter, and possibly the `CCNameMapper` class.

Integrating ContactManager with ClaimCenter is a multi-step process:

1. “Integrate ClaimCenter with ContactManager” on page 23
2. “Integrate ContactManager with ClaimCenter” on page 24
3. “Test the integration of ClaimCenter and ContactManager” on page 26
4. “Troubleshooting the ClaimCenter connection with ContactManager” on page 27

See also

- “Installing ContactManager with QuickStart for development” on page 17
- “Configure ClaimCenter-to-ContactManager authentication” on page 42
- “Configure ContactManager-to-ClaimCenter authentication” on page 46

Integrate ClaimCenter with ContactManager

This step is the first in the multi-step process of integrating ClaimCenter and ContactManager.

About this task

For an overview of the integration process, see “Integrating ContactManager with ClaimCenter in QuickStart” on page 21

Procedure

1. At a command prompt, navigate to the ClaimCenter installation folder, and then start Guidewire Studio™ for ClaimCenter by entering the following command:

   ```
   gwb studio
   ```

2. In Guidewire Studio for ClaimCenter, open the **Project** window.
3. Press `Ctrl+Shift+N` and enter `suite-config.xml` to find this file, and then double-click the search result to open the file in the editor.

   This file defines the URLs of the applications in the Guidewire InsuranceSuite. If the entries in this file are commented out, you must remove the comments for the applications you have installed. In the default configuration, the Guidewire applications have the following settings in this file:

   ```
   <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>
   ```

4. Remove the comments around the ContactManager (ab) entry, as follows:

   ```
   <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>
   ```

5. Press `Ctrl+Shift+N` and enter `config.xml` to find this file, and then double-click the search result to open the file in the editor.
6. Add the ContactManager URL to the `ContactSystemURL` configuration parameter.
   For example:

   ```
   <param name="ContactSystemURL" value="http://localhost:8280/ab"/>
   ```

   This URL supports an exit point to ContactManager from the browser in which ClaimCenter is running. For example, the **Edit in ContactManager** button available when editing a contact uses this URL to open ContactManager. If you do not set this URL, the system uses the ContactManager URL defined in `suite-config.xml`. However, in that case, you cannot have separate browser and integration support for the URL.

7. In the **Project** window, expand **configuration** > **config** > **Plugins** > **registry**.
8. Double-click `ContactSystemPlugin.gwp` to open it in the Registry editor.

   By default, the system uses the plugin implementation `gw.plugin.addressbook.demo.DemoContactSystemPlugin`. This Java plugin implementation is provided only for product demonstrations and is not to be used for any other purpose. In the steps that follow, you replace this class with the plugin implementation that connects ClaimCenter with ContactManager.

9. In the Registry editor, click Remove Plugin −.
10. Click Add Plugin + and, in the drop-down menu, choose **Add Gosu Plugin** to register the new plugin implementation.
11. In the **Gosu Class** field, enter the following class:

```
gw.plugin.contact.ab1000.ABContactSystemPlugin
```

12. If necessary, start ContactManager. At a command prompt, navigate to the ContactManager installation folder and enter the following command:

```
gwb runServer
```

13. In the Guidewire Studio for ClaimCenter **Project** window, navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.ab.ab1000.wsc`.
14. Double-click the `ab1000.wsc` web services collection to open the editor, and then, in the editor, click the following resource:

```
${ab}/ws/gw/webservice/ab/ab1000/abcontactapi/ABContactAPI?wsdl
```

The `${ab}` variable in this path corresponds to the `product name="ab"` entry in the `suite-config.xml` file that specifies the URL for ContactManager.

15. With ContactManager running, click **Fetch** to get the latest updates to `ABContactAPI`.
16. Ensure ContactManager sample data is loaded. See "Load sample data for ContactManager" on page 18. If sample data is not loaded and you do not want to load sample data, create the following user in ContactManager:

```
Username = "ClientAppCC"
Password = "gw"
```

In the base configuration, the `ABConfigurationProvider` gosu class defines the `ClientAppCC` user credentials that ClaimCenter uses to authenticate with ContactManager when ClaimCenter makes calls to `ABContactAPI`. This user is included in ContactManager sample data.

IMPORTANT: Guidewire recommends that you change this user name and password in the `ABConfigurationProvider` gosu class and in ContactManager.

17. Close Guidewire Studio for ClaimCenter.

What to do next

"Integrate ContactManager with ClaimCenter" on page 24

Integrate ContactManager with ClaimCenter

To set up the Guidewire ContactManager™ side of integration with Guidewire ClaimCenter™, use Guidewire Studio™ for ContactManager to change suite and plugin registry settings.

Before you begin

Complete "Integrate ClaimCenter with ContactManager" on page 23.

Procedure

1. At a command prompt, navigate to the ContactManager installation folder, and then start Guidewire Studio™ for ContactManager by entering the following command:

```
gwb studio
```

2. In Guidewire Studio for ContactManager, open the **Project** window.

3. Press `Ctrl+Shift+N` and enter `suite-config.xml` to find this file, and then double-click the search result to open the file in the editor.

   This file defines the URLs of the applications in the Guidewire InsuranceSuite. If the entries in this file are commented out, you must remove the comments for the applications you have installed. In the default configuration, the Guidewire applications have the following settings in this file:

   ```
   <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>
   ```

4. Remove the comments around the ClaimCenter (cc) entry, as follows:

   ```
   <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>
   ```

5. Open the **Project** window and expand **configuration** > **config** > **Plugins** > **registry**.
6. Double-click `ClaimSystemPlugin.gwp` to open it in the Plugin Registry editor.

   You need to register a plugin implementation so ContactManager can use it to synchronize address book changes with ClaimCenter. By default, the system uses `gw.plugin.integration.StandAloneClientSystemPlugin`. You must replace this plugin implementation because it does not broadcast `Contact` changes and does not communicate with core applications.

7. In the Registry editor, click Remove Plugin −.
8. Click Add Plugin + and, in the drop-down menu, choose **Add Gosu Plugin** to register the new plugin implementation.
9. In the **Gosu Class** field, enter the following class:

   `gw.plugin.claim.cc1000.CCClaimSystemPlugin`

10. Navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.cc.cc1000.wsc`.
11. Double-click the `cc1000.wsc` web services collection to open the editor, and then, in the editor, click the following resource: `${cc}/ws/gw/webservice/cc/cc1000/contact/ContactAPI?wsdl`.

    ContactManager calls this ClaimCenter webservice when it has `Contact` updates for ClaimCenter. The ClaimCenter web service `ContactAPI` implements the interface `ABClientAPI`. The `${cc}` variable in this path corresponds to the `product name="cc"` entry in the `suite-config.xml` file, which specifies the URL for ClaimCenter.

12. Ensure that the ClaimCenter server is running, and then click **Fetch** to get the latest updates to `ContactAPI`.
13. In the same editor on the **Settings** tab is the configuration provider class. This class, `wsi.remote.gw.webservice.cc.CCConfigurationProvider`, defines the user name and password that ContactManager must use to connect with ClaimCenter.

    IMPORTANT: By default, ContactManager uses the user name `su` and password `gw`. Guidewire recommends that you change this user name and password in ClaimCenter and update `CCConfigurationProvider` in ContactManager.

14. Close Guidewire Studio for ContactManager.
15. To pick up your changes, stop and restart the two servers.

a) Stop ClaimCenter. Open a command prompt in the ClaimCenter installation folder and then enter the following command:

```
gwb stopServer
```

b) Stop ContactManager. Open a command prompt in the ContactManager installation folder and then enter the following command:

```
gwb stopServer
```

c) Start the ContactManager application in the ContactManager installation folder:

```
gwb runServer
```

d) Start the ClaimCenter application in the ClaimCenter installation folder:

```
gwb runServer
```

What to do next

"Test the integration of ClaimCenter and ContactManager" on page 26

Test the integration of ClaimCenter and ContactManager

After setting up integration for ClaimCenter and ContactManager, you can perform some tests to ensure that the integration is working.

Before you begin

Complete "Integrate ContactManager with ClaimCenter" on page 24. Additionally, ensure that both ContactManager and ClaimCenter have started. Finally, if you are using the base configurations of ClaimCenter and ContactManager, ContactManager must have a ClientAppCC user for ClaimCenter to authenticate with. Sample data in ContactManager includes this user.

Procedure

1. When ClaimCenter is ready, open a browser window and enter the ClaimCenter URL. For example:

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

2. Log in as a user who has permissions to view, create, edit, and search for a ContactManager contact in ClaimCenter. For example, log in as the sample user `ssmith` with password `gw`.
3. Open an existing claim and then click **Parties Involved** to open the **Contacts** screen.
4. Click **New Contact** and create a new contact.
5. Give the contact enough data to be able to save it in ContactManager, such as name, address, and tax ID information.
6. Give the contact a role on the claim and click **Update** to save it.
7. When the **Contacts** screen opens, select the new contact. Because you are logged in with a role that permits writing to ContactManager, the contact links automatically to ContactManager. If the link is successful, you see the message, `This contact is linked to the Address Book and is in sync.`
8. Click the **Address Book** tab to search for a contact.
With sample data loaded in ContactManager, you can search for a Vendor (Company) with a name that starts with the letter `a`. That search returns contacts with names like AB Construction and Allendale, Myers & Associates.
9. Open a browser window and enter the following ContactManager URL:

```
http://localhost:8280/ab/
```

10. Log in as a user who can view or edit a contact in ContactManager, such as the sample user `aapplegate` with password `gw`.
11. In ContactManager, click **Search** and verify that you can search for and locate the contact you just created in ClaimCenter.
12. Click the contact in the search results and edit it. For example, change the contact's phone number. Click **Update** to save the changes.
13. In ClaimCenter, open the claim to which you added the new contact, and click **Parties Involved** to open the **Contacts** screen.
14. Select the contact that you added.
    On the **Basics** card, if you have not changed default settings, you see the following message:

    ```
    This contact is linked to the Address Book and is in sync.
    ```
15. Click **View in Address book**.

    ClaimCenter fetches the contact data from ContactManager and displays it in a new screen.

    a) In this screen, you can click **Edit in ContactManager** to open a browser window and connect to the ContactManager server.

       The system looks for the ContactManager URL in the configuration parameter `ContactSystemURL`, or in `suite-config.xml` if that parameter is not defined. You must also have a ContactManager user name to be able to use this feature. Additionally, that user must have a role in ContactManager that enables editing contacts, such as the Contact Manager role.
    b) If necessary, log in to ContactManager.
    c) Edit the contact data directly.
    d) Go back to the ClaimCenter window in your browser.
16. Click **Return to Contacts** near the top of the screen.
    You see the **Contacts** screen again, with your contact on the **Basics** card showing the changes you made in ContactManager.

What to do next

If you encounter errors during testing, see "Troubleshooting the ClaimCenter connection with ContactManager" on page 27.

Troubleshooting the ClaimCenter connection with ContactManager

ClaimCenter error: bad username or password

During testing of the ClaimCenter connection with ContactManager, you might see an error message saying that there was an exception caused by a bad user name or password. If so, you probably need to create the ContactManager user that ClaimCenter uses to communicate with ContactManager. In the base configuration, this user is `ClientAppCC` with password `gw`. This user is available if you import the sample data for ContactManager.

See also

- "Test the integration of ClaimCenter and ContactManager" on page 26
- "Configuring core application authentication with ContactManager" on page 41
- For instructions on how to load sample data for ContactManager, see "Load sample data for ContactManager" on page 18.
- To create a user in ContactManager, see "Configuring ContactManager authentication with core applications" on page 45.

ClaimCenter error: no response from ContactManager

During testing of the ClaimCenter connection with ContactManager, you might not be able to get any response at all from ContactManager. If so, it is possible that the message queue that ClaimCenter uses to communicate with ContactManager is suspended. In that case, you must reactivate the ClaimCenter message queue.

See also

- "Reactivate the ClaimCenter message queue" on page 28

Reactivate the ClaimCenter message queue

If you are not getting a response from ContactManager in ClaimCenter, you can restart the message queue that ClaimCenter uses to communicate with ContactManager.

Before you begin

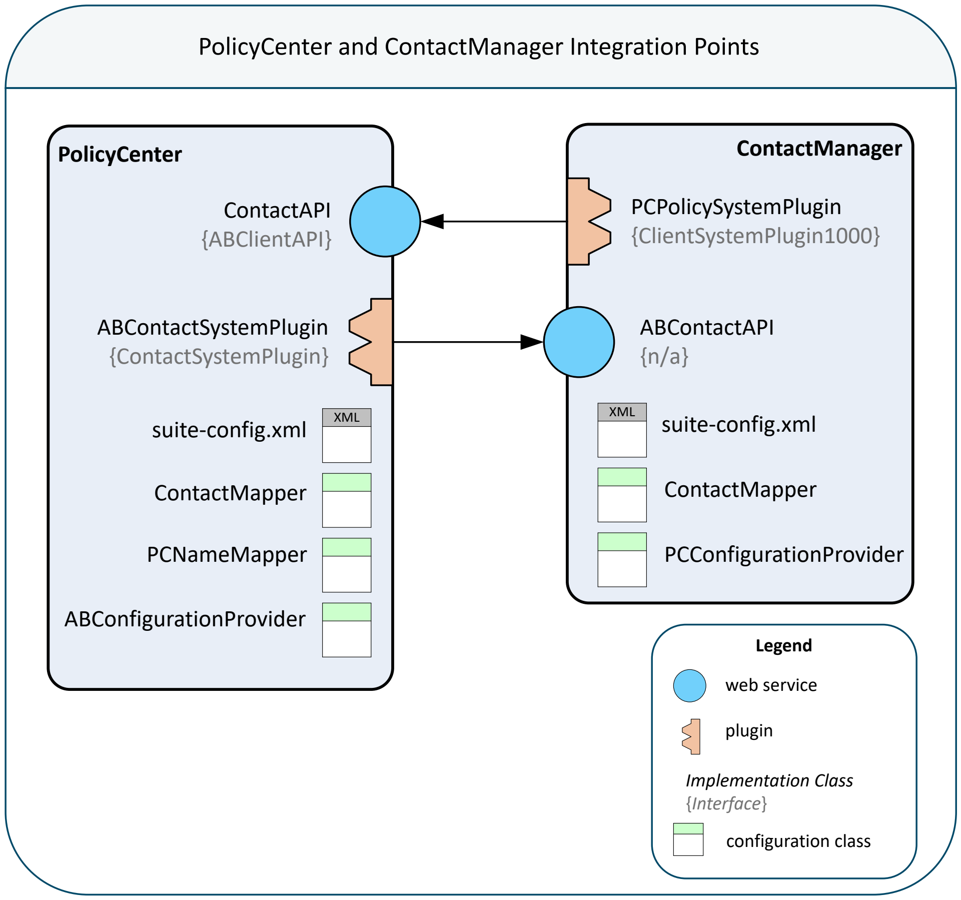
Integrate ContactManager and ClaimCenter.

Procedure

1. Ensure that ClaimCenter and ContactManager are running.
2. Log in to ClaimCenter as a user with administrative privileges. For example, log in with user name `su` and password `gw`.
3. Click the **Administration** tab, and then navigate to **Monitoring** > **Message Queues** in the Sidebar.
4. On the **Message Queues** screen, select the check box next to the **Contact Message Transport** entry.
5. If the **Status** is Suspended, click the **Resume** button at the top of the screen.
6. The **Status** changes to Started, and ClaimCenter can now send messages to ContactManager.

Integrating ContactManager with PolicyCenter in QuickStart

The following figure shows the primary integration points used to integrate PolicyCenter and ContactManager.



In general, to get the basic integration working, you edit the `suite-config.xml` files in PolicyCenter and ContactManager. You also register the `PCPolicySystemPlugin` plugin implementation in the ContactManager plugin registry `PolicySystemPlugin.gwp`. Additionally, you register `ABContactSystemPlugin` in the PolicyCenter registry `ContactSystemPlugin.gwp`.

If you extend the contact model, you must edit the `ContactMapper` classes in both ContactManager and PolicyCenter, and possibly the `PCNameMapper` class.

IMPORTANT: Guidewire recommends that you change the authentication user and password. To do so, you add authentication users to ContactManager and PolicyCenter and edit the `ABConfigurationProvider` and `PCConfigurationProvider` classes.

Integrating ContactManager and PolicyCenter is a multi-step process:

1. “Integrate PolicyCenter with ContactManager” on page 30
2. “Integrate ContactManager with PolicyCenter” on page 32
3. ““Test the Integration of PolicyCenter and ContactManager””
4. “Troubleshooting the PolicyCenter connection with ContactManager” on page 34

See also

- “Configure PolicyCenter-to-ContactManager authentication” on page 43
- “Configure ContactManager-to-PolicyCenter authentication” on page 48

Integrate PolicyCenter with ContactManager

This step is the first in the multi-step process of integrating PolicyCenter and ContactManager.

About this task

This step is the first in the multi-step process of integrating ContactManager and PolicyCenter. For an overview of the integration process, see “Integrating ContactManager with PolicyCenter in QuickStart” on page 28.

Procedure

1. At a command prompt, navigate to the PolicyCenter installation folder, and then start Guidewire Studio™ for PolicyCenter by entering the following command:

   ```
   gwb studio
   ```

2. In Guidewire Studio for PolicyCenter, open the **Project** window.
3. Press `Ctrl+Shift+N` and enter `suite-config.xml` to find this file, and then double-click the search result to open the file in the editor.

   This file defines the URLs of the applications in the Guidewire InsuranceSuite. If the entries in this file are commented out, you must remove the comments for the applications you have installed. In the default configuration, the Guidewire applications have the following settings in this file:

   ```
   <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>
   ```

4. Remove the comments around the ContactManager (ab) entry, as follows:

   ```
   <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>
   ```

5. In the **Project** window, expand **configuration** > **config** > **Plugins** > **registry**.
6. Double-click `ContactSystemPlugin.gwp` to change the plugin implementation used by the system to connect to ContactManager.

   By default, the system uses `gw.plugin.contact.impl.StandAloneContactSystemPlugin`. PolicyCenter uses this plugin when not connected to a contact management system. In the steps that follow, you replace this class with the plugin implementation that connects PolicyCenter with ContactManager.

7. In the Registry editor, click Remove Plugin −.
8. Click Add Plugin + and, in the drop-down menu, choose **Add Gosu Plugin** to register the new plugin implementation.
9. In the **Gosu Class** field, enter the following class:

   `gw.plugin.contact.ab1000.ABContactSystemPlugin`

10. If necessary, start ContactManager.

    At a command prompt, navigate to the ContactManager installation folder and enter the following command:

    ```
    gwb runServer
    ```

11. In the Guidewire Studio for PolicyCenter **Project** window, navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.ab.ab1000.wsc`.

12. Double-click the `ab1000.wsc` web services collection to open the editor, and then, in the editor, click the following resource:

    ```
    ${ab}/ws/gw/webservice/ab/ab1000/abcontactapi/ABContactAPI?wsdl
    ```

    The `${ab}` variable in this path corresponds to the `product name="ab"` entry in the `suite-config.xml` file that specifies the URL for ContactManager.

13. With ContactManager running, click **Fetch** to get the latest updates to `ABContactAPI`.

14. Ensure ContactManager sample data is loaded. See "Load sample data for ContactManager" on page 18. If sample data is not loaded and you do not want to load sample data, create the following user in ContactManager:

    ```
    Username = "ClientAppPC"
    Password = "gw"
    ```

    In the base configuration, the `ABConfigurationProvider` gosu class defines the `ClientAppPC` user credentials that PolicyCenter uses to authenticate with ContactManager when PolicyCenter makes calls to `ABContactAPI`. This user is included in ContactManager sample data.

    IMPORTANT: Guidewire recommends that you change this user name and password in the `ABConfigurationProvider` gosu class and in ContactManager.

15. In the **Project** window, press `Ctrl+N` and enter `ContactMessageTransport`. In the search results, double-click `ContactMessageTransport` to open this Gosu class in the editor.

16. Press `Ctrl+F` and enter `getAdminUserForIntegrationHandling` to find that method in the class. The method is defined as follows:

    ```
    private function getAdminUserForIntegrationHandling() : User {
      return PLDependenciesGateway.getUserFinder().findByCredentialName("admin")
    }
    ```

    This method gets the PolicyCenter user that is assigned an activity if ContactManager throws an unhandled exception during communication with PolicyCenter. In response to the activity, this user reviews the contact's data and makes the needed corrections. After the user updates the contact, PolicyCenter sends it to ContactManager again.

    a) You must designate a user of your own to handle this activity.

       The `getAdminUserForIntegrationHandling` method retrieves the user by login name. In the base configuration, this PolicyCenter user has all permissions. The generic text for the activity is:

       ```
       Subject: Failed to add the contact '{Contact}' to the CMS.
       Description: An unexpected error occurred when adding the contact to the contact management system\:
         {Error Message}
       ```

    b) Assign the user roles with permissions that enable working with contacts.

       The contact and tag permissions in the base configuration are `abcreate`, `ctccreate`, `anytagcreate`, `abdelete`, `ctcdelete`, `anytagdelete`, `abedit`, `ctcedit`, `anytagedit`, `abview`, `abviewsearch`, `ctcview`, and `anytagview`.

    c) When you determine the user who will handle these activities, you can create your own class and copy the code in the `ContactMessageTransport` class into your class.

    d) In your class's `getAdminUserForIntegrationHandling` method, replace the parameter `admin` with the login name of your user.

    e) Navigate to **configuration** > **config** > **Plugins** > **registry** and register your class in the plugin registry `ContactMessageTransport.gwp`.

17. Close Guidewire Studio for PolicyCenter.

What to do next

"Integrate ContactManager with PolicyCenter" on page 32

Integrate ContactManager with PolicyCenter

To set up the Guidewire ContactManager™ side of integration with Guidewire PolicyCenter™, use Guidewire Studio™ for ContactManager to change suite and plugin registry settings.

Before you begin

Complete "Integrate PolicyCenter with ContactManager" on page 30.

Procedure

1. At a command prompt, navigate to the ContactManager installation folder, and then start Guidewire Studio™ for ContactManager by entering the following command:

   ```
   gwb studio
   ```

2. In Guidewire Studio for ContactManager, open the **Project** window.
3. Press `Ctrl+Shift+N` and enter `suite-config.xml` to find this file, and then double-click the search result to open the file in the editor.

   This file defines the URLs of the applications in the Guidewire InsuranceSuite. If the entries in this file are commented out, you must remove the comments for the applications you have installed. In the default configuration, the Guidewire applications have the following settings in this file:

   ```
   <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>
   ```

4. Remove the comments around the PolicyCenter (pc) entry, as follows:

   ```
   <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>
   ```

5. Open the **Project** window and expand **configuration** > **config** > **Plugins** > **registry**.
6. Double-click `PolicySystemPlugin.gwp` to open it in the Plugin Registry editor.

   You need to register a plugin implementation so ContactManager can use it to synchronize address book changes with PolicyCenter. By default, the system uses `gw.plugin.integration.StandAloneClientSystemPlugin`. You must replace this plugin implementation because it does not broadcast `Contact` changes and does not communicate with core applications.

7. In the Registry editor, click Remove Plugin —.
8. Click Add Plugin + and, in the drop-down menu, choose **Add Gosu Plugin** to register the new plugin implementation.
9. In the **Gosu Class** field, enter the following class:

   `gw.plugin.policy.pc1000.PCPolicySystemPlugin`

10. Navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.pc.pc1000.wsc`.
11. Double-click the `pc1000.wsc` web services collection to open the editor, and then, in the editor, click the following resource:

    ```
    ${pc}/ws/gw/webservice/pc/pc1000/contact/ContactAPI?wsdl
    ```

ContactManager calls this PolicyCenter webservice when it has `Contact` updates for PolicyCenter. The PolicyCenter web service `ContactAPI` implements the interface `ABClientAPI`. The `${pc}` variable in this path corresponds to the `product name="pc"` entry in the `suite-config.xml` file, which specifies the URL for PolicyCenter.

12. Ensure that the PolicyCenter server is running, and then click **Fetch** to get the latest updates to `ContactAPI`.
13. At the bottom of the window is the **Settings** tab. The default setting in this tab is the configuration provider class. This class, `wsi.remote.gw.webservice.pc.PCConfigurationProvider`, defines the user name and password that ContactManager uses to connect with PolicyCenter.

IMPORTANT: In the base configuration, ContactManager uses the user name `su` and password `gw`. Guidewire recommends that you change this user name and password in the `PCConfigurationProvider` class and in ContactManager.

14. When the update finishes, close Guidewire Studio for ContactManager.
15. To pick up your changes, stop and restart ContactManager and PolicyCenter.
    a) Stop PolicyCenter.

       Open a command prompt in the PolicyCenter installation folder and then enter the following command:

       ```
       gwb stopServer
       ```
    b) Stop ContactManager.

       Open a command prompt in the ContactManager installation folder and then enter the following command:

       ```
       gwb stopServer
       ```
    c) Start the ContactManager application in the ContactManager installation folder:

       ```
       gwb runServer
       ```
    d) Start the PolicyCenter application in the PolicyCenter installation folder:

       ```
       gwb runServer
       ```

What to do next

“Test the integration of PolicyCenter and ContactManager” on page 33

Test the integration of PolicyCenter and ContactManager

After setting up integration for PolicyCenter and ContactManager, you can perform some tests to ensure that the integration is working.

Before you begin

You must complete “Integrate ContactManager with PolicyCenter” on page 32 before continuing with this step. Additionally, ensure that both ContactManager and PolicyCenter have started. Finally, if you are using the base configurations of PolicyCenter and ContactManager, ContactManager must have a ClientAppPC user for PolicyCenter to authenticate with. Sample data in ContactManager includes this user.

About this task

Verify that the two applications are integrated.

Procedure

1. When PolicyCenter is ready, open a browser window and enter the following PolicyCenter URL:

   ```
   http://localhost:8180/pc/
   ```

2. Log in as a user who can create a contact in PolicyCenter, such as the sample user `aapplegate` with password `gw`.
3. Open an existing account from the **Account** tab.
4. Click **Contacts** in the left Sidebar menu.
5. At the top of the **Account File Contacts** screen, click **Create New Contact**.
6. Click **Additional Interest** > **From Address Book**.
7. Choose **Person** for the type and search for a contact that is stored in ContactManager.

   With sample data loaded in ContactManager, you could search for the person William Andy.
8. In the search results, look for a contact that has its **External** column set to **Yes**. That contact is stored only in ContactManager.

   If a contact is stored in both PolicyCenter and ContactManager, its **External** column is set to **No**. If you do not see any contacts that are external, either ContactManager has not found a contact stored only in ContactManager or it has suspended its PolicyCenter message queue.

   - If you are sure the contact is stored only in ContactManager, the contact's tag might not be set to Client. Any contact created in PolicyCenter and stored in ContactManager automatically gets a Client tag, but you have to add the tag manually if you create the contact directly in ContactManager.
   - An additional possibility is that PolicyCenter has stopped the message queue it uses for ContactManager.
9. Click **Select** for the person found in ContactManager.
   You see that contact added to the list of account file contacts in the role Additional Interest.
10. Open a browser window and enter the following ContactManager URL:

    `http://localhost:8280/ab/`
11. Log in as a user who can view or edit a contact in ContactManager, such as the sample user `aapplegate` with password `gw`.
12. In ContactManager, click **Search** and find the contact you added to the account in PolicyCenter.
13. Edit that contact and change the address, and then update the contact.
14. In PolicyCenter, go to the same **Contact** screen for the account to which you added the contact, and then click the contact.
15. Verify that the contact's address has changed to the one you specified in ContactManager.

What to do next

If you encounter errors during testing, see "Troubleshooting the PolicyCenter connection with ContactManager" on page 34.

Troubleshooting the PolicyCenter connection with ContactManager

PolicyCenter connection with ContactManager not working

You must be working with either a policy that is in force or an account that has at least one policy in force for contacts to be saved in ContactManager. If PolicyCenter appears not to be working with ContactManager and there is no error message, check the policy or account status.

See also

- “Test the integration of PolicyCenter and ContactManager” on page 33

PolicyCenter error: bad username or password

You might see an error message saying that there was an exception caused by a bad user name or password. If so, you probably need to create the ContactManager user that PolicyCenter uses to communicate with ContactManager. In the base configuration, this user is ClientAppPC with password gw. This user is available if you import the sample data for ContactManager.

See also

- To create a user in ContactManager, see “Configuring ContactManager authentication with core applications” on page 45.
- “Configuring core application authentication with ContactManager” on page 41
- “Load sample data for ContactManager” on page 18

PolicyCenter error: no response from ContactManager

If you are not able to get any response at all from ContactManager, it is possible that the message queue that PolicyCenter uses to communicate with ContactManager has been suspended.

See also

- “Reactivate the PolicyCenter message queue” on page 35

Reactivate the PolicyCenter message queue

If you are not getting a response from ContactManager in PolicyCenter, you can restart the message queue that PolicyCenter uses to communicate with ContactManager.

Before you begin

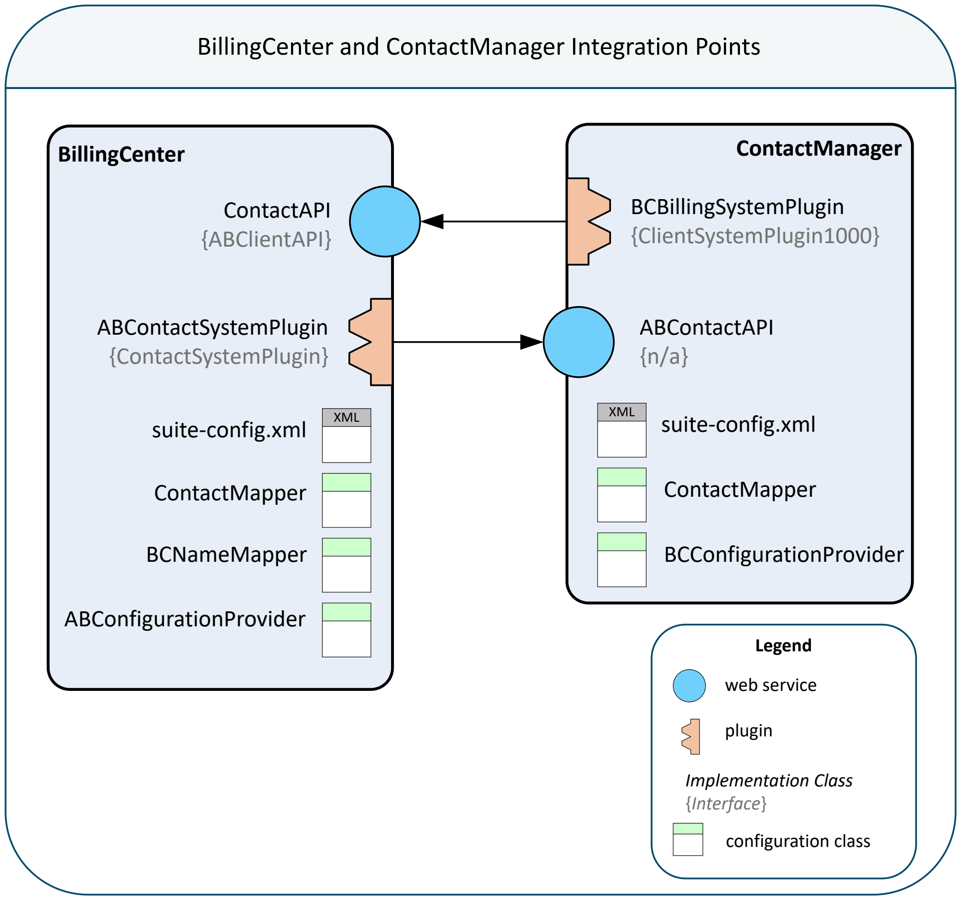
Integrate ContactManager and PolicyCenter.

Procedure

1. If they are not already running, start PolicyCenter and ContactManager.
2. Log in to PolicyCenter as a user with administrative privileges. For example, log in with user name `su` and password `gw`.
3. Click the **Administration** tab, and then navigate to **Monitoring** > **Message Queues** in the Sidebar on the left.
4. On the **Message Queues** screen, select the **ContactMessageTransport** entry.
5. If the **Status** is Suspended, click the **Resume** button at the top of the screen.
6. The **Status** changes to Started, and PolicyCenter can now send messages to ContactManager.

Integrating ContactManager with BillingCenter in QuickStart

The following figure shows the primary integration points used to integrate BillingCenter and ContactManager.



In general, to get the basic integration working, you edit the `suite-config.xml` files in BillingCenter and ContactManager. You also register the `BCBillingSystemPlugin` plugin implementation in the ContactManager plugin registry `BillingSystemPlugin.gwp`. Additionally, you register `ABContactSystemPlugin` in the BillingCenter registry `ContactSystemPlugin.gwp`.

If you extend the contact model, you edit the `ContactMapper` classes in both ContactManager and BillingCenter, and possibly the `BCNameMapper` class.

IMPORTANT: Guidewire recommends that you change the authentication user and password. To do so, you add authentication users to ContactManager and BillingCenter and edit the `ABConfigurationProvider` and `BCConfigurationProvider` classes.

Detailed steps for integrating BillingCenter and ContactManager are a multi-step process:

1. "Integrate BillingCenter with ContactManager" on page 37
2. "Integrate ContactManager with BillingCenter" on page 38
3. "Test the integration of BillingCenter and ContactManager" on page 40
4. "Troubleshooting the BillingCenter connection with ContactManager" on page 41

See also

- "Configure BillingCenter-to-ContactManager authentication" on page 44
- "Configure ContactManager-to-BillingCenter authentication" on page 51
- "Extending the client data model" on page 118

Integrate BillingCenter with ContactManager

This step is the first in the multi-step process of integrating BillingCenter and ContactManager.

About this task

For an overview of the integration process, see “Integrating ContactManager with BillingCenter in QuickStart” on page 35.

Procedure

1. At a command prompt, navigate to the BillingCenter installation folder, and then start Guidewire Studio™ for BillingCenter by entering the following command:

   ```
   gwb studio
   ```

2. In Guidewire Studio for BillingCenter, open the **Project** window.
3. Press `Ctrl+Shift+N` and enter `suite-config.xml` to find this file, and then double-click the search result to open the file in the editor.

   This file defines the URLs of the applications in the Guidewire InsuranceSuite. If the entries in this file are commented out, you must remove the comments for the applications you have installed. In the default configuration, the Guidewire applications have the following settings in this file:

   ```
   <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>
   ```

4. Remove the comments around the ContactManager (ab) entry, as follows:

   ```
   <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>
   ```

5. In the **Project** window, expand **configuration** > **config** > **Plugins** > **registry**.
6. Double-click `ContactSystemPlugin.gwp` to open it in the Registry editor.

   By default, the system uses the plugin implementation `gw.plugin.contact.impl.StandAloneContactSystemPlugin`. BillingCenter uses this Gosu plugin if it is not integrated with a contact management system. In the steps that follow, you replace this class with the plugin implementation that connects BillingCenter with ContactManager.

7. In the Registry editor, click Remove Plugin −.
8. Click Add Plugin + and, in the drop-down menu, choose **Add Gosu Plugin** to register the new plugin implementation.
9. In the **Gosu Class** field, enter the following class:

   ```
   gw.plugin.contact.ab1000.ABContactSystemPlugin
   ```

10. If necessary, start ContactManager.

    At a command prompt, navigate to the ContactManager installation folder and enter the following command:

    ```
    gwb runServer
    ```

11. In the Guidewire Studio for BillingCenter **Project** window, navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.ab.ab1000.wsc`.

12. Double-click the `ab1000.wsc` web services collection to open the editor, and then, in the editor, click the following resource:

    ```
    ${ab}/ws/gw/webservice/ab/ab1000/abcontactapi/ABContactAPI?wsdl
    ```

    The `${ab}` variable in this path corresponds to the `product name="ab"` entry in the `suite-config.xml` file that specifies the URL for ContactManager.

13. With ContactManager running, click **Fetch** to get the latest updates to `ABContactAPI`.
14. In the same editor, look at the **Settings** tab.

    There is a setting for the configuration provider `wsi.remote.gw.webservice.ab.ABConfigurationProvider`. This class defines the user name and password that BillingCenter uses to authenticate with ContactManager when BillingCenter makes calls to `ABContactAPI`. In the base configuration, `ABConfigurationProvider` defines the following:

    ```
    config.Guidewire.Authentication.Username = "su"
    config.Guidewire.Authentication.Password = "gw"
    ```

    Note: Guidewire recommends that you change this user name and password in the `ABConfigurationProvider` class and in ContactManager.

15. When the update finishes, close Guidewire Studio for BillingCenter.

What to do next

The next step is “Integrate ContactManager with BillingCenter” on page 38.

Integrate ContactManager with BillingCenter

To set up the ContactManager side of integration with BillingCenter, use Guidewire Studio for ContactManager to change suite and plugin registry settings.

Before you begin

Complete “Integrate BillingCenter with ContactManager” on page 37.

Procedure

1. At a command prompt, navigate to the ContactManager installation folder, and then start Guidewire Studio™ for ContactManager by entering the following command:

    ```
    gwb studio
    ```

2. In Guidewire Studio for ContactManager, open the **Project** window.
3. Press `Ctrl+Shift+N` and enter `suite-config.xml` to find this file, and then double-click the search result to open the file in the editor.

    This file defines the URLs of the applications in the Guidewire InsuranceSuite. If the entries in this file are commented out, you must remove the comments for the applications you have installed. In the default configuration, the Guidewire applications have the following settings in this file:

    ```
    <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>
    ```

4. Remove the comments around the BillingCenter (bc) entry, as follows:

    ```
    <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>
```

5. Open the **Project** window and navigate to **configuration** > **config** > **Plugins** > **registry**.
6. Double-click `BillingSystemPlugin.gwp` to open it in the Registry editor.

   You need to register a plugin implementation so ContactManager can use it to synchronize address book changes with BillingCenter. By default, the system uses `gw.plugin.integration.StandAloneClientSystemPlugin`. You must replace this plugin implementation because it does not broadcast `Contact` changes and does not communicate with core applications.
7. In the Registry editor, click Remove Plugin —.
8. Click Add Plugin + and, in the drop-down menu, choose **Add Gosu Plugin** to register the new plugin implementation.
9. In the **Gosu Class** field, enter the following class:

   ```
   gw.plugin.billing.bc1000.BCBillingSystemPlugin
   ```
10. Navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.bc.bc1000.wsc`.
11. Double-click the `bc1000.wsc` web services collection to open the editor, and then, in the editor, click the following resource:

    ```
    ${bc}/ws/gw/webservice/bc/bc1000/contact/ContactAPI?wsdl
    ```

    ContactManager calls this BillingCenter web service when it has `Contact` updates for BillingCenter. The BillingCenter web service `ContactAPI` implements the interface `ABClientAPI`. The `${bc}` variable in this path corresponds to the `product name="bc"` entry in the `suite-config.xml` file, which specifies the URL for BillingCenter.
12. Ensure that BillingCenter is running, and then click **Fetch** to get the latest updates to `ContactAPI`.
13. In the same editor on the **Settings** tab is the configuration provider class `wsi.remote.gw.webservice.bc.BCConfigurationProvider`.

    This class defines the user name and password that ContactManager uses to connect with BillingCenter.

    IMPORTANT: In the base configuration, ContactManager uses the user name `su` and password `gw`. Guidewire recommends that you change this user name in the `BCConfigurationProvider` class and in ContactManager.
14. When the update finishes, close Guidewire Studio for ContactManager.
15. To pick up your changes, stop and restart ContactManager and BillingCenter.

    a) Stop BillingCenter.

       Open a command prompt in the BillingCenter installation folder and then enter the following command:

       ```
       gwb stopServer
       ```

    b) Stop ContactManager.

       Open a command prompt in the ContactManager installation folder and then enter the following command:

       ```
       gwb stopServer
       ```

    c) Start the ContactManager application in the ContactManager installation folder:

       ```
       gwb runServer
       ```

d) Start the BillingCenter application in the BillingCenter installation folder:

```
gwb runServer
```

What to do next

"Test the integration of BillingCenter and ContactManager" on page 40

Test the integration of BillingCenter and ContactManager

After setting up integration for BillingCenter and ContactManager, you can perform some tests to ensure that the integration is working.

Before you begin

Complete "Integrate ContactManager with BillingCenter" on page 38. Additionally, ensure that both ContactManager and BillingCenter have started.

About this task

Verify that the two applications are integrated. If you are using the base configurations of BillingCenter and ContactManager, ContactManager provides the su user for BillingCenter to authenticate with.

Procedure

1. When BillingCenter is ready, open a browser window and enter the following BillingCenter URL:

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

2. Log in as a user who can create a contact in BillingCenter, such as the sample user `aapplegate` with password `gw`.
3. Open an existing account from the **Account** tab.
4. Click **Contacts** in the left Sidebar under **Actions**.
5. On the **Contacts** screen, click the **Edit** button.
6. Click **Add Existing Contact**, and search for a company you know is in ContactManager and has a Client tag, such as the sample company Albertson's.

   If a contact is stored in both BillingCenter and ContactManager, its **External** column is set to **No**. If you do not see any contacts that are external, either ContactManager has not found contacts stored only in ContactManager or it has suspended its BillingCenter message queue.

   - If you are sure the contact is stored only in ContactManager, the contact's tag might not be set to Client. Any contact created in BillingCenter and stored in ContactManager does have a Client tag.
   - An additional possibility is that BillingCenter has stopped the message queue it uses for ContactManager.
7. Click **Select** next to the company name.
8. On the **Contacts** screen, select the company name.
9. Below, on the **Contact Info** card, change contact data, like the **Address 1** field, and then click **Update** to save the change.
10. Open a browser window and enter the following ContactManager URL:

    ```
    http://localhost:8280/ab/
    ```

11. Log in as a user who can view or edit a contact in ContactManager, such as the sample user `aapplegate` with password `gw`.
12. In ContactManager, click **Search** and find the contact you changed in BillingCenter.

13. Verify that the data you changed for the contact in BillingCenter was also changed for the contact in ContactManager.

What to do next

If you encounter errors during testing, see "Troubleshooting the BillingCenter connection with ContactManager" on page 41.

Troubleshooting the BillingCenter connection with ContactManager

BillingCenter error: bad username or password error

You might see an error message saying that there was an exception caused by a bad user name or password. If so, you probably need to create the ContactManager user that BillingCenter uses to communicate with ContactManager. In the base configuration, this user is `su` with password `gw`.

See also

- "Configuring ContactManager authentication with core applications" on page 45
- "Configuring core application authentication with ContactManager" on page 41
- "Load sample data for ContactManager" on page 18

BillingCenter error: no response from ContactManager

If you are not able to get any response at all from ContactManager, it is possible that the message queue that BillingCenter uses to communicate with ContactManager has been suspended.

See also

- "Reactivate the BillingCenter message queue" on page 41

Reactivate the BillingCenter message queue

If you are not getting a response from ContactManager in BillingCenter, you can restart the message queue that BillingCenter uses to communicate with ContactManager.

Before you begin

Integrate ContactManager and BillingCenter.

Procedure

1. If they are not already running, start BillingCenter and ContactManager.
2. Log in to BillingCenter as a user with administrative privileges. For example, log in with user name `su` and password `gw`.
3. Click the **Administration** tab, and then navigate to **Monitoring** > **Message Queues** in the sidebar on the left.
4. On the **Message Queues** screen, select the **Contact Message Transport** entry.
5. If the **Status** is Suspended, click the **Resume** button at the top of the screen.
6. The **Status** changes to Started, and BillingCenter can now send messages to ContactManager.

Configuring core application authentication with ContactManager

When you integrate a Guidewire core application with ContactManager, you register a plugin implementation with the core application's plugin registry. For example, to integrate ContactManager 10.0 with a core application of the same release, you register `gw.plugin.contact.ab1000.ABContactSystemPlugin` with the core application's `ContactSystemPlugin.gwp` registry.

Each core application defines a user name and password that it uses to authenticate with ContactManager when the application uses its plugin to call `ABContactAPI` methods. The definition is in the class `wsi.remote.gw.webservice.ab.ABConfigurationProvider`. For security purposes, Guidewire recommends that you define your own user names and passwords.

- ClaimCenter defines the two authentication parameters, `username` and `password`, as `ClientAppCC` and `gw`.
- PolicyCenter defines the two authentication parameters, `username` and `password`, as `ClientAppPC` and `gw`.
- BillingCenter defines the two authentication parameters, `username` and `password`, as `su` and `gw`.

The base ContactManager application's sample code contains users with names and passwords that match the default settings in the core applications. The sample code has users with names `ClientAppCC`, `ClientAppPC`, and `su`, all with password `gw`.

Guidewire recommends that you change the `username` and `password` that each application uses to authenticate with ContactManager when making calls to `ABContactAPI`. The process, which is similar for all applications, follows:

1. Set up a user and password in ContactManager to match the user name and password you define in the core application.
2. Define the user name and password in the core application's `ABConfigurationProvider` class.

Configure ClaimCenter-to-ContactManager authentication

Guidewire recommends that you change the user name and password that ClaimCenter uses to authenticate with ContactManager.

Procedure

1. Start ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb runServer
   ```

2. Log in as a user who can create new ContactManager users.
   For example, log in with user name `su` and password `gw`.
3. Click the **Administration** tab, and then navigate to **Actions** > **New User**.
4. Enter the following values:

Name	Value
First Name	`CC`
Last Name	`NewAuthUser`
Username	`CCNewAuthUser`
Password	`Xc8899Lm`
Confirm Password	`Xc8899Lm`
Active	`Yes`
Locked	`No`

5. Under **Roles**, click **Add**.
6. Click the **Name** field and choose **Client Application**.
7. Click **Update**.
8. Start Guidewire Studio for ClaimCenter.

At a command prompt, navigate to the ClaimCenter installation folder and enter the following command:

```
gwb studio
```

9. Open the **Project** window.
10. Press `Ctrl+N` and enter `ABConfigurationProvider`, and then click the class name in the search results to open it in the editor.
11. Change the `Username` and `Password` definition in the `configure` method.
For example, change the values to `CCNewAuthUser` and `Xc8899Lm`, as follows:

```
override function configure( serviceName : QName, portName : QName, config : WsdlConfig )  {
  config.Guidewire.Authentication.Username = "CCNewAuthUser"
  config.Guidewire.Authentication.Password = "Xc8899Lm"
}
```

12. In the **Project** window, navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.ab.ab1000.wsc`.
13. Double-click the `ab1000.wsc` web resource collection to open the editor.
14. In the editor, click `${ab}/ws/gw/webservice/ab/ab1000/abcontactapi/ABContactAPI?wsdl`.
15. With ContactManager still running, click **Fetch** .
16. When the update finishes, stop the ClaimCenter server if it is running, and then restart it to pick up this change.
17. Log in to ClaimCenter as a user with permission to work with the Address Book, such as user `ssmith` with password `gw`.
18. Verify that you can use the **Address Book** tab to search for ContactManager contacts, such as the sample contact William Weeks.

Configure PolicyCenter-to-ContactManager authentication

Guidewire recommends that you change the user name and password that PolicyCenter uses to authenticate with ContactManager.

Procedure

1. Start ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

```
gwb runServer
```

2. Log in as a user who can create new ContactManager users.
For example, log in with user name `su` and password `gw`.
3. Click the **Administration** tab, and then navigate to **Actions** > **New User**.
4. Enter the following values:

Name	Value
First Name	`PC`
Last Name	`NewAuthUser`
Username	`PCNewAuthUser`
Password	`Xc8899Lm`
Confirm Password	`Xc8899Lm`
Active	`Yes`

Name	Value
Locked	No

5. Under **Roles**, click **Add**.
6. Click the **Name** field and choose **Client Application**.
7. Click **Update**.
8. Start Guidewire Studio for PolicyCenter.

   At a command prompt, navigate to the PolicyCenter installation folder and enter the following command:

   ```
   gwb studio
   ```
9. Press `Ctrl+N` and enter `ABConfigurationProvider`, and then click the class name in the search results to open it in the editor.
10. Change the `Username` and `Password` definition in the `configure` method.
    For example, change the values to `PCNewAuthUser` and `Xc8899Lm`, as follows:

    ```
    override function configure( serviceName : QName, portName : QName, config : WsdlConfig )  {
      config.Guidewire.Authentication.Username = "PCNewAuthUser"
      config.Guidewire.Authentication.Password = "Xc8899Lm"
    }
    ```
11. In the **Project** window, navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.ab.ab1000.wsc`.
12. Double-click the `ab1000.wsc` web services collection to open the editor.
13. In the editor, click `${ab}/ws/gw/webservice/ab/ab1000/abcontactapi/ABContactAPI?wsdl`.
14. With ContactManager still running, click **Fetch**.
15. When the update finishes, stop the PolicyCenter server if it is running, and then restart it to pick up this change.
16. Log in to PolicyCenter as a user with permission to search for and change contacts, such as user name `aapplegate` with password `gw`.
17. Verify that you can use the **Contact** tab to search for ContactManager contacts. You can be sure that a contact is stored in ContactManager if its **External** field has the value **Yes**. For example, search for the person Adam Hinds, a client contact in the ContactManager sample data.

Configure BillingCenter-to-ContactManager authentication

Guidewire recommends that you change the user name and password that BillingCenter uses to authenticate with ContactManager.

Procedure

1. Start ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb runServer
   ```
2. Log in as a user who can create new ContactManager users.
   For example, log in with user name `su` and password `gw`.
3. Click the **Administration** tab, and then navigate to **Actions** > **New User**.
4. Enter the following values:

Name	Value
First Name	BC

Name	Value
Last Name	`NewAuthUser`
Username	`BCNewAuthUser`
Password	`Xc8899Lm`
Confirm Password	`Xc8899Lm`
Active	`Yes`
Locked	`No`

5. Under **Roles**, click **Add**.
6. Click the **Name** field and choose **Client Application**.
7. Click **Update**.
8. Start Guidewire Studio for BillingCenter.

   At a command prompt, navigate to the BillingCenter installation folder and enter the following command:

   ```
   gwb studio
   ```

9. Press `Ctrl+N` and enter `ABConfigurationProvider`, and then click the class name in the search results to open it in the editor.
10. Change the `Username` and `Password` definition in the `configure` method.
    For example, change the values to `BCNewAuthUser` and `Xc8899Lm`, as follows:

    ```
    override function configure( serviceName : QName, portName : QName, config : WsdlConfig )  {
      config.Guidewire.Authentication.Username = "BCNewAuthUser"
      config.Guidewire.Authentication.Password = "Xc8899Lm"
    }
    ```

11. In the **Project** window, navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.ab.ab1000.wsc`.
12. Double-click the `ab1000.wsc` web services collection to open the editor.
13. In the editor, click `${ab}/ws/gw/webservice/ab/ab1000/abcontactapi/ABContactAPI?wsdl`.
14. With ContactManager still running, click **Fetch** .

    For more information on the web service editor, see the *Configuration Guide*.
15. When the update finishes, stop BillingCenter server if it is running, and then restart it to pick up this change.
16. Log in to BillingCenter as a user with permission to search for and change contacts, such as user name `su` with password `gw`.
17. Verify that you can use the **Contact** tab to search for ContactManager contacts. You can be sure that a contact is stored in ContactManager if its **External** field has the value **Yes**. For example, search for the person Adam Hinds, a client contact in the ContactManager sample data.

Configuring ContactManager authentication with core applications

You can change the user names and passwords that ContactManager uses to send `ABContact` updates to the core applications. In the base configuration, ContactManager defines this user as `su`, an unrestricted user that, by default, can perform any action in a core application. Guidewire recommends for security reasons that you change this default setting to a user with permission only to create, update, and delete contacts.

Overview of ContactManager authentication configuration

Initially, to set up ContactManager to send `ABContact` updates to a Guidewire core application, you open Guidewire Studio for ContactManager and register a plugin with the core application's plugin registry. For example:

- To communicate with ClaimCenter 10.0, you register `gw.plugin.claim.cc1000.CCClaimSystemPlugin` with the plugin registry `ClaimSystemPlugin.gwp`.
- To communicate with PolicyCenter 10.0, you register `gw.plugin.policy.pc1000.PCPolicySystemPlugin` with the plugin registry `PolicySystemPlugin.gwp`.
- To communicate with BillingCenter 10.0, you register `gw.plugin.billing.bc1000.BCBilllingSystemPlugin` with the plugin registry `BillingSystemPlugin.gwp`.

ContactManager uses these plugins to make calls to core application web services that implement `ABClientAPI`.

ContactManager defines core application authorization for each web service in a configuration provider class, which is associated with the web service in a web service collection. To see the web services that ContactManager uses with a core application, you open the editor for the web service collection.

ContactManager provides the following web service collections for each core application. Each web service collection enables you to open and edit a configuration provider class, where you can define the user name and password that ContactManager uses with the core application.

ClaimCenter

`cc1000.wsc` uses `wsi.remote.gw.webservice.cc.CCConfigurationProvider`.

PolicyCenter

`pc1000.wsc` uses `wsi.remote.gw.webservice.pc.PCConfigurationProvider`.

BillingCenter

`bc1000.wsc` uses `wsi.remote.gw.webservice.bc.BCConfigurationProvider`.

In general, you change the user name and password as follows:

1. Set up a user with a name and password in the core application that matches the user name and password you define in the configuration provider class in ContactManager.
2. Give the user a role with limited permissions, with at least the permissions to create, edit, and delete contacts and the permission SOAP administration.
3. In ContactManager, define the user name and password in the configuration provider class for the core application web service.

Configure ContactManager-to-ClaimCenter authentication

Guidewire recommends that you change the user name and password that ContactManager uses to authenticate with ClaimCenter.

Procedure

1. Start the ClaimCenter server.

   At a command prompt, navigate to the ClaimCenter installation folder and enter the following command:

   ```
   gwb runServer
   ```

2. Log in as a user who can create new ClaimCenter users and roles.
   For example, log in with user name `su` and password `gw`.
3. Create a user role with all the permissions that enable working with contacts whose contact information is stored in ContactManager. In the base configuration, the minimum required permission codes are `abcreate`, `abcreatepref`, `anytagcreate`, `ctccreate`, `abdelete`, `abdeletepref`, `anytagdelete`, `abedit`, `abeditpref`, `anytagedit`, `ctcedit`, and `soapadmin`.

a) Click the **Administration** tab, and then navigate to **Actions** > **Roles**.

b) On the **Roles** screen, click **Add Role**.

c) For **Name** enter `Contact Data Admin`.

d) For **Type**, choose **User Role**.

e) For **Description** enter `Permissions for virtual user required by ContactManager for updating contacts`.

f) Beneath the **Description** field, click **Add**.

g) Click the **Permission** field and choose **Create address book contacts** from the list.

h) Repeat the process for the following permissions. For each permission, click **Add** and choose the permission from the drop-down list:

- **Create address book preferred vendors**
- **Create contact with any tag**
- **Create local contacts**
- **Delete address book contacts**
- **Delete address book preferred vendors**
- **Delete contact with any tag**
- **Edit address book contacts**
- **Edit address book preferred vendors**
- **Edit contact with any tag**
- **Edit local contacts**
- **SOAP administration**

i) Click **Update** to create the new Contact Data Admin role.

4. Choose **Actions** > **New User**.
5. Enter the following values for a new user named `CMUpdateCC`:

Name	Value
First Name	`CMUpdate`
Last Name	`CC`
Username	`CMUpdateCC`
Password	`Xc8899Lm`
Confirm Password	`Xc8899Lm`
Active	`Yes`
Locked	`No`

6. Under **Roles**, click **Add**.
7. Click the **Name** field and choose `Contact Data Admin`.

   If this role has more permissions than you would like, you can create a new role and assign the new user to that role. The new role must have at least the following permissions: `abcreate`, `abdelete`, `abedit`, `abview`, `abcreatepref`, `abdeletepref`, `abeditpref`, `abviewsearch`, `anytagcreate`, `anytagdelete`, `anytagedit`, `anytagview`, `ctccreate`, `ctcedit`, `ctcview`, and `soapadmin`. For more information on setting up role-based security, see the *Application Guide*.
8. Click **Update**.
9. Start Guidewire Studio for ContactManager.

At a command prompt, navigate to the ContactManager installation folder and enter the following command:

```
gwb studio
```

10. In the **Project** window, navigate to **configuration** > **gsrc**, and then navigate to `wsi.remote.gw.webservice.cc.CCConfigurationProvider`.
11. Double-click `CCConfigurationProvider` to open it in the editor.
12. Change the `Username` and `Password` definition in the `configure` method and then save your changes.
    For example, define `Username` and `Password` to be `CMUpdateCC` and `Xc8899Lm`, as follows:

    ```
    override function configure( serviceName : QName,
                                 portName : QName,
                                 config : WsdlConfig )  {
      config.Guidewire.Authentication.Username = "CMUpdateCC"
      config.Guidewire.Authentication.Password = "Xc8899Lm"
    }
    ```

13. In the editor, the `cc1000.wsc` web services collection is at the same level as the `CCConfigurationProvider` class. Double-click `cc1000.wsc`.

    Alternatively, you can press `Ctrl+Shift+N` and enter `cc1000`, and then double-click `cc1000.wsc` in the search results.
14. In the editor, select the following resource:

    ```
    ${cc}/ws/gw/webservice/cc/cc1000/contact/ContactAPI?wsdl
    ```

15. Ensure that the **Settings** tab has the **Setting Type** for **ConfigurationProvider** set to `wsi.remote.gw.webservice.cc.CCConfigurationProvider`, the class you just edited.
16. With ClaimCenter still running, above the **Settings** tab, click **Fetch** to update the web service's WSDL file.
17. When the update finishes, stop the ContactManager server and then restart it to pick up this change.
18. Log in to ClaimCenter as a user with permission to work with the Address Book, such as user `ssmith` with password `gw`.
19. Add to a claim a contact that is stored in ContactManager.
    For example, open a claim, click **Parties Involved**, and on the **Contacts** screen, click **Add Existing Contact**. Search for a contact that you know is stored in ContactManager, like the sample contact Stephen Marshall. Select the contact and give the contact a role on the claim, and then click **Update**.
20. Log in to ContactManager as a user who can work with contacts.
    For example, log in as the sample user `aapplegate` with password `gw`.
21. Make a change to the contact that you added to the claim, such as a different phone number, and save it.
22. Back in ClaimCenter, on the **Contact** screen, select a different contact from the one you updated.
23. Select the contact that you added.
    On the **Basics** card for that contact is the message, **This contact is linked to the Address Book but is out of sync**.
24. Click **Copy from Address Book** to update the contact.

Configure ContactManager-to-PolicyCenter authentication

Guidewire recommends that you change the user name and password that ContactManager uses to authenticate with PolicyCenter.

Procedure

1. Start the PolicyCenter server.

   At a command prompt, navigate to the PolicyCenter installation folder and enter the following command:

   ```
   gwb runServer
   ```

2. Log in as a user who can create new PolicyCenter users and roles.
   For example, log in with user name `su` and password `gw`.
3. Create a user role with all the permissions that enable working with clients whose contact information is stored in ContactManager. The permission codes in the base configuration are `abcreate`, `ctccreate`, `anytagcreate`, `abdelete`, `anytagdelete`, `abedit`, `anytagedit`, `ctcedit`, and `soapadmin`.
   a) Click the **Administration** tab, and then navigate to **Actions** > **Roles**.
   b) On the **Roles** screen, click **New Role**.
   c) For **Name** enter `Client Data Admin`.
   d) For **Type**, choose **User Role**.
   e) For **Description** enter `Permissions for virtual user required by ContactManager for updating contacts`.
   f) Under **Permissions**, click **Add**.
   g) Click the **Permission** field and choose **Create address book contacts** from the list.
   h) Repeat the process for the following permissions. For each permission, click **Add** and choose the permission from the drop-down list:
      - **Create contact with any tag**
      - **Create local contacts**
      - **Delete address book contacts**
      - **Delete contact with any tag**
      - **Edit address book contacts**
      - **Edit contact with any tag**
      - **Edit local contacts**
      - **SOAP administration**
   i) Click **Update** to create the new Client Data Admin role.
4. On the **Administration** tab, navigate to **Actions** > **New User**.
5. Enter the following values for a new user named `CMUpdatePC`:

Name	Value
First Name	`CMUpdate`
Last Name	`PC`
Username	`CMUpdatePC`
Password	`Xc8899Lm`
Confirm Password	`Xc8899Lm`
Active	`Yes`
Locked	`No`
Vacation Status	`At work`
User Type	`Other`
Primary Phone	`Work`
Work Phone	`800-555-5555`

   Primary Phone and **Work Phone** are required for a new user, but they do not require real values for this particular user.
6. Click the **Roles** tab, and then click **Add**.

7. Click the **Name** field and choose `Client Data Admin`.
8. Click **Update**.
9. Start Guidewire Studio for ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb studio
   ```
10. In the **Project** window, navigate to **configuration** > **gsrc**, and then navigate to `wsi.remote.gw.webservice.pc.PCConfigurationProvider`.
11. Double-click `PCConfigurationProvider` to open it in the editor.
12. Change the `Username` and `Password` definition in the `configure` method.
    For example, define `Username` and `Password` to be `CMUpdatepc` and `Xc8899Lm`, as follows:

    ```
    override function configure( serviceName : QName,
                                 portName : QName,
                                 config : WsdlConfig )  {
      config.Guidewire.Authentication.Username = "CMUpdatePC"
      config.Guidewire.Authentication.Password = "Xc8899Lm"
    }
    ```
13. In the **Project** window, the `pc1000.wsc` web services collection is at the same level as the `PCConfigurationProvider` class. Double-click `pc1000.wsc`.

    Alternatively, you can press `Ctrl+Shift+N` and enter `pc1000`, and then double-click `pc1000.wsc` in the search results.
14. In the editor, select the following resource:

    ```
    ${pc}/ws/gw/webservice/pc/pc1000/contact/ContactAPI?wsdl
    ```
15. Ensure that the **Settings** tab has the **Setting Type** for **ConfigurationProvider** set to `wsi.remote.gw.webservice.pc.PCConfigurationProvider`, the class you just edited.
16. With PolicyCenter still running, above the **Settings** tab, click **Fetch** to update the web service's WSDL file.
17. When the update finishes, stop the ContactManager server and restart it to pick up this change.
18. Log in to PolicyCenter as a user with permission to work with the **Contact** tab and with accounts, such as the user `aapplegate` with password `gw`.
19. Add a contact stored in ContactManager to an active account. For example:
    a) Open an active account, such as the sample data account S000212121 named Armstrong Cleaners, and, in the Sidebar on the left under **Actions**, click **Contacts**.
       The **Account File Contacts** screen opens.
    b) Click **Create New Contact** > **Additional Insured** > **From Address Book**.
    c) Search for a contact who has the Client tag set in ContactManager, such as the sample `Person` contact Mark Stone.
    d) Click **Select** for a contact that the search results list says is stored in ContactManager.

       The value of the contact's **External** field must be **Yes**.
    e) PolicyCenter adds the selected contact to the list of contacts on the **Account File Contacts** screen.
20. Log in to ContactManager as a user who can work with contacts.
    For example, log in as the sample user `aapplegate` with password `gw`.
21. Make a change to the contact that you added to the account, such as entering a different primary phone number, and then click **Update** to save the change.
22. Back in PolicyCenter, on the **Account File Contacts** screen, select a different contact from the one you added.

23. Select the contact that you added previously.
PolicyCenter updates the added contact’s data.
24. Ensure that the data has changed for that contact.

Configure ContactManager-to-BillingCenter authentication

Guidewire recommends that you change the user name and password that ContactManager uses to authenticate with BillingCenter.

Procedure

1. Start the BillingCenter server.

   At a command prompt, navigate to the BillingCenter installation folder and enter the following command:

   ```
   gwb runServer
   ```

2. Log in as a user who can create new BillingCenter users.
For example, log in with user name `su` and password `gw`.
3. Create a user role with all the permissions that enable working with clients whose contact information is stored in ContactManager. The permission codes are `acctcntcreate`, `acctcntdelete`, `acctcntedit, plcycntcreate`, `plcycntdelete, plcycntedit`, `prodcntcreate`, `prodcntdelete`, `prodcntedit`, and `soapadmin`.
   a) Click the **Administration** tab, and then navigate to **Actions** > **Roles**.
   b) On the **Roles** screen, click **New Role**.
   c) For **Name** enter `Client Data Admin`.
   d) For **Description** enter `Permissions for virtual user required by ContactManager for updating contacts`.
   e) Below the **Description** field, click **Add**.
   f) Click the **Permission** field and choose **Create account contact** from the list.
   g) Repeat the process for the following permissions. For each permission, click **Add** and choose the permission from the drop-down list:
      - **Create policy contact**
      - **Create producer contact**
      - **Delete account contact**
      - **Delete policy contact**
      - **Delete producer contact**
      - **Edit account contact**
      - **Edit policy contact**
      - **Edit producer contact**
      - **SOAP administration**
   h) Click **Update** to create the new Client Data Admin role.
4. On the **Administration** tab, navigate to **Actions** > **New User**.
5. Enter the following values for a new user named `CMUpdateBC`:

Name	Value
First Name	`CMUpdate`
Last Name	`BC`

Name	Value
Username	`CMUpdateBC`
Password	`Xc8899Lm`
Confirm Password	`Xc8899Lm`

6. Click **Add User Role**.
7. Click the **Name** list and choose `Client Data Admin`.
8. Click **Next**. and add some fake data for the required fields **Address**, **City**, **Country**, **Primary Phone**, and the telephone number for the type of primary phone you selected.
9. Click **Next**.
10. Click **Finish**.
11. Start Guidewire Studio for ContactManager.

    At a command prompt, navigate to the ContactManager installation folder and enter the following command:

    ```
    gwb studio
    ```

12. In the **Project** window, navigate to **configuration** > **gsrc**, and then navigate to `wsi.remote.gw.webservice.bc.BCConfigurationProvider`.
13. Double-click `BCConfigurationProvider` to open it in the editor.
14. Change the `Username` and `Password` definition in the `configure` method.
    For example, define `Username` and `Password` to be `CMUpdateBC` and `Xc8899Lm`, as follows:

    ```
    override function configure( serviceName : QName,
                                 portName : QName,
                                 config : WsdlConfig )  {
      config.Guidewire.Authentication.Username = "CMUpdateBC"
      config.Guidewire.Authentication.Password = "Xc8899Lm"
    }
    ```

15. In the **Project** window, the `bc1000.wsc` web services collection is at the same level as the `BCConfigurationProvider` class. Double-click `bc1000.wsc`.

    Alternatively, you can press `Ctrl+Shift+N` and enter `bc1000`, and then double-click `bc1000.wsc` in the search results.

16. In the editor, select the following resource:

    ```
    ${bc}/ws/gw/webservice/bc/bc1000/contact/ContactAPI?wsdl
    ```

17. Ensure that the **Settings** tab has the **Setting Type** for **ConfigurationProvider** set to `wsi.remote.gw.webservice.bc.BCConfigurationProvider`, the class you just edited.
18. With BillingCenter still running, above the **Settings** tab, click **Fetch** to update the web service's WSDL file.
19. When the update finishes, stop the ContactManager server and then restart it to pick up this change.
20. Log in to BillingCenter as a user with permission to work with the **Contacts** screen for accounts, such as the user `su` with password `gw`.
21. Add a contact stored in ContactManager to an active account. For example:

    a) Open an active account, such as the sample account named Standard Account, and, in the Sidebar on the left under **Actions**, click **Contacts**. The **Contacts** screen opens.

    b) Click **Edit,** and then click **Add Existing Contact**.

    c) Search for a contact that has the Client tag set in ContactManager, such as the sample `Person` contact Stan Newton. If necessary, log in to ContactManager and add a Client tag to a contact.

    d) Click **Select** for a contact that the search results list says is stored in ContactManager.

The value of the contact’s **External** field must be **Yes**.

e) BillingCenter adds the selected contact to the list of contacts on the **Contacts Edit** screen.

f) Click **Update** to add the contact to the account.

22. Log in to ContactManager as a user who can work with contacts.
For example, log in as the sample user `aapplegate` with password `gw`.

23. Make a change to the contact that you added to the account, such as entering a different address, and click **Update** to save the change.

24. Back in BillingCenter, on the **Contacts** screen, select the contact you added and ensure that the data has changed for that contact.

chapter 4

Searching for contacts

ContactManager supports search for contacts both in its own user interface and searches initiated externally through web services, such as from the Guidewire core applications. There are configuration points for contact search both in ContactManager and in the core applications.

Overview of contact search

Guidewire core applications store contacts locally, and, if integrated with ContactManager, a core application can also store contacts centrally in the ContactManager database. PolicyCenter and BillingCenter support searching for contacts both locally and in an external contact management application, like ContactManager. ClaimCenter supports searching for contacts only in an external contact management application, like ContactManager.

For example, in ClaimCenter, if you have added a contact to a claim, you can see a list of contacts by navigating to the **Parties Involved** > **Contacts** screen. All the contacts on the list are stored locally with the claim. Contacts can also be stored in ContactManager. If you click a contact in the list, you see its detailed information below the list. If the contact is stored in ContactManager, at the top of the **Basics** card is a message starting with the text `This contact is linked to the Address Book`.

- In ClaimCenter, you can search for a contact in the Address Book by clicking the **Address Book** tab and entering your search criteria. All contact data returned by this search comes from contacts stored in ContactManager. You can also search for contacts when creating existing contacts for claims. For example, with a claim open, you can navigate to the **Parties Involved** > **Contacts** screen and click **Add Existing Contact**.
- In PolicyCenter, you can search for a contact by clicking the **Contact** tab and then entering your search criteria. Alternatively, you can click the **Search** tab menu and choose **Contact**. Additionally, you can open the **Contacts** screen for an account or policy and select a contact. Then you can click **Add Existing Contact**, choose a contact type, and then choose **From Address Book** to open the contact search screen.
- In BillingCenter, you can search for a contact by clicking the **Search** tab and choosing **Contacts**, and then entering your search criteria. Additionally, if you edit the **Contacts** screen for an account or policy period, clicking **Add Existing Contact** opens the contact search screen.

Searching for contacts in PolicyCenter and BillingCenter returns information both about locally stored contacts and about contacts stored in an external contact management system, such as ContactManager. The list of search results has an **External** field that provides the following information:

- If the value in this field is `Yes`, the contact is not local, but is stored in an external contact management system.
- If the value in this field is `No`, the contact is stored locally and might also be stored in an external contact management system.

In all cases, searching for contacts returns a subset of information about the contact, which you see listed in the set of returned contacts.

IMPORTANT: Limiting the fields retrieved by a search improves performance. In the base configuration, ContactManager filters search results to reduce the size of the objects returned to the calling application. To avoid performance issues, ensure that your searches are optimized if possible to return only the parts of objects that you need and not the full objects.

The following topics describe how contact search works between the core applications and ContactManager:

- "ContactManager support for contact searches" on page 56
- "ClaimCenter support for contact searches" on page 58
- "PolicyCenter support for Contact searches" on page 81
- "BillingCenter support for Contact searches" on page 82

ContactManager support for contact searches

ContactManager supports contact search that is initiated both locally, from its own user interface, and externally, through web services. External searches most commonly originate from Guidewire core applications.

Local ContactManager search criteria

If you run ContactManager and log in as a user who has permission to search for and view contacts, you can search for contacts by clicking the **Contacts** tab. The criteria you use for searches on this screen are defined in the ContactManager `search-config.xml` file and in the entity `ABContactSearchCriteria`.

IMPORTANT: There are restrictions on adding new `CriteriaDef` elements. See the *Configuration Guide*. Additionally, there are limitations on how you can use the `ArrayCriterion` element. See the *Configuration Guide*.

See also

- For an example, see "Adding the InterpreterSpecialty property to search" on page 59.
- For complete information on `search-config.xml`, see the *Configuration Guide*

Defining minimum search criteria for a contact

There is a plugin class in the base configuration of ContactManager in which you can define the minimum criteria required to find a contact, `gw.plugin.contact.ValidateABContactSearchCriteriaPluginImpl`. Settings in this class affect both local searches and searches from Guidewire core applications. You can add new, required search fields by editing this class. Because the class is written in Gosu, you have considerable flexibility in how you specify which fields to require for a search.

The following code snippet from the parent class, `ValidateABContactSearchCriteriaPluginBase`, shows the default search criteria defined for an `ABPerson` entity, which you can override:

```
protected function abPersonCanSearch(searchCriteria : ABContactSearchCriteria) : boolean {
  if (searchCriteria.FirstName != null or searchCriteria.FirstNameKanji != null) {
    if (searchCriteria.Keyword == null and searchCriteria.KeywordKanji == null) {
      return false
    }
  }
  if (searchCriteria.Keyword == null
      and searchCriteria.KeywordKanji == null
      and searchCriteria.FirstName == null
      and searchCriteria.FirstNameKanji == null
      and searchCriteria.TaxID == null
      and satisfiesNoLocaleSpecificCriteriaRequirements(searchCriteria)
      and searchCriteria.Address.PostalCode == null
      and not searchCriteria.isValidProximitySearch()) {
    return false
  }
  return true
}
```

Effect of locale and country on minimum search criteria

Different locales have different requirements for searches. ContactManager determines the locale for the name fields of a search and uses the country specified to determine the other fields required.

The following code snippet from `ValidateABContactSearchCriteriaPluginBase.validateCanSearch` shows the code that picks up the locale and the country that the user specified in the search:

```
var country = searchCriteria.Address.Country ?: gw.api.admin.BaseAdminUtil.getDefaultCountry()
if (searchSpec != null && searchSpec.Locale == null)
  searchSpec.Locale = LocaleType.get(PLConfigParameters.DefaultApplicationLocale.Value)
}
```

Note: If the country or the locale or both are not passed in, the method uses default values.

The `validateCanSearch` method throws an exception based on the contact subtype if the search criteria are not met:

```
var exception:ContactSubtypeSpecificException = null
if (searchCriteria.SpecialistServices != null and
    searchCriteria.SpecialistServices.Count > 0) {
  exception = new ContactSearchWithServiceCriteriaException(
    searchCriteria.ContactSubtypeType, country, searchSpec.Locale)
} else {
  exception = new TooLooseContactSearchCriteriaException(
    searchCriteria.ContactSubtypeType, country, searchSpec.Locale)
}
em = exception.Message
```

The message sent uses the locale and country to send the appropriate search criteria back to the calling application.

Note: In the class `ValidateABContactSearchCriteriaPluginImpl`, you can override the `satisfiesNoLocaleSpecificCriteriaRequirements` method to provide locale-specific search criteria. Your override can determine if the locale information can pass validation. For example, the base implementation requires a combination of City and State, but some locales might not use states or might have different requirements altogether.

See also

- “Minimum search criteria error messages” on page 57

Minimum search criteria error messages

The `ValidateABContactSearchCriteriaPluginImpl` class extends `ValidateABContactSearchCriteriaPluginBase`. This class uses the `TooLooseContactSearchCriteriaException` message definitions to pass locale and country specific error messages to the calling application. In the base configuration, these messages are display keys in the `display.properties` file.

To see this file, open Guidewire Studio for ContactManager. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.

In the file, if you press `Ctrl+F` and search for `TooLooseContactSearchCriteriaException`, you see the set of display keys defined for this exception. For example:

```
Java.TooLooseContactSearchCriteriaException.ABCompany.AU.ja_JP =
  Please specify one of the following\: Name (phonetic), Name, Tax ID, Postcode, or City and State
```

The display keys are grouped first by entity type, `ABCompany`, `ABContact`, `ABPerson`, and `ABPlace`. In each set, there are display keys for the countries Australia (`AU`), Canada (`CA`), Germany (`DE`), France (`FR`), Great Britain (`GB`), Japan (`JP`), and the United States (`US`). For each country, there is a display key for one locale, `ja_JP`.

Note: You can add display keys for additional countries, locales, and entity types. If you add additional country or locale display keys, be sure to add them for all entity types, which in the base configuration are `ABCompany`, `ABContact`, `ABPerson`, and `ABPlace`.

This classification of messages enables ContactManager to return different contact search error messages based on country and locale. For example, a ClaimCenter user searches for a `MedicalCareOrg`, a `Company` subtype, located in the United States. The user has previously set the locale to `ja_JP` (Japan). ContactManager determines that there are not enough search criteria to find a company in that country and locale, and sends an exception message defined by the following display key:

```
Java.TooLooseContactSearchCriteriaException.ABCompany.US.ja_JP =
  Please specify one of the following\: Name (phonetic), Name, Tax ID, Zip Code, or City and State
```

Note: Each entity has one display key that does not specify a country or locale. This display key is a default in case the country or locale is not specified. Additionally, the display keys that have only a country and no locale are the default display keys for all locales that are not defined. In the base configuration, the only locale for which a message is defined is `ja_JP`.

ContactManager support for core application searches

The ContactManager web service `gw.webservice.ab.ab1000.abcontactapi.ABContactAPI` provides methods that the Guidewire core applications call to search for, create, retrieve, update, and delete contacts. Additionally, this class provides a method for finding vendor contacts by associated service. The method that supports search is `ABContactAPI.searchContact`. This class is read-only and cannot be directly edited.

You configure search in other classes and in a configuration parameter.

See also

- "ABContactAPI web service" on page 303

Configuring Contact search criteria and search results in ContactManager

The Gosu class `ABContactAPISearchCriteria` specifies the contact search criteria that the Guidewire core applications can use. The package for this class is `gw.webservice.ab.ab1000.abcontactapi`. You can edit this class to add new search criteria for core application contact searches.

You can also define the fields included in the search results that ContactManager sends back to a Guidewire core application. To add a field to the search results, add code for the field to the Gosu class `gw.webservice.ab.ab1000.abcontactapi.ABContactAPISearchResult`.

These classes, `ABContactAPISearchCriteria` and `ABContactAPISearchResult`, are classes used as parameters in the web service API. After changing either of these classes, restart ContactManager to regenerate these web services. Then start Guidewire Studio™ for the Guidewire core application, navigate to the `ab1000.wsc` web collection, and fetch updates for the ContactManager web service `ABContactAPI`.

See also

- "Add search support in ContactManager for Guidewire core applications" on page 60

Limiting the number of service elements specified in a Contact search

If there are too many services specified in a contact search, the search can fail. To limit this number, there is a ContactManager configuration parameter, `MaxNumberServicesInSearchQuery`, that you can set in `config.xml`. This configuration parameter defaults to a value of 20. If a contact search specifies more services than the number set in this parameter, ContactManager sends an error message to the core application.

If `MaxNumberServicesInSearchQuery` is set to too large a number, ContactManager can experience SQL errors. In that case, ContactManager returns an error message to the core application with a note to check the logs in ContactManager for the details of the error.

ClaimCenter support for contact searches

There are two files that define the contact search criteria that ClaimCenter sends to ContactManager, the entity `ContactSearchCriteria.etx` and the Gosu class `ContactSearchMapper`. The entity defines the search criteria that appear in the search screens, and the Gosu class maps the search criteria to ContactManager. ClaimCenter uses these criteria when it calls the ContactManager `ABContactAPI` method `searchContacts`. ClaimCenter calls this method in the plugin `ABContactSystemPlugin`, which implements `ContactSystemPlugin`.

There is a third Gosu class, `ContactSearchResultMapper`, that `ABContactSystemPlugin` uses to map fields sent from ContactManager search results so that ClaimCenter can display them in the lists of search results.

Note: ClaimCenter displays `Contact` entities in the search results. Therefore, the result mapper maps from ContactManager results to `Contact` entities.

The files are:

`ContactSearchCriteria.etx`

Defines the search criteria that are shown to the user in the search screens. If you add entries to `ContactSearchCriteria`, you must also add them to `ContactSearchMapper`. ClaimCenter uses both files.

`gw.plugin.addressbook.ab1000.ContactSearchMapper`

Defines how search criteria map to ContactManager `ABContact` search criteria.

`gw.plugin.addressbook.ab1000.ContactSearchResultMapper`

Defines the search results received from ContactManager to display in the search results screen.

See also

These files work in concert with the ContactManager search files described at “ContactManager support for core application searches” on page 58.

Adding the InterpreterSpecialty property to search

The entity `ABInterpreter` is a subtype of `ABPersonVendor` created in an example. It has one field, `InterpreterSpecialty`. Adding `InterpreterSpecialty` to search is a multi-step process that is split into a series of topics.

Before starting this process, complete the example “Adding a vendor contact subtype” on page 125.

IMPORTANT: To improve search performance, Guidewire recommends that you add an index in the ContactManager entity definition for every field used in the search. The `ABInterpreter` example does define an index.

Add search support to the ContactManager user interface

Make changes to the ContactManager user interface to support searching for the `InterpreterSpecialty` field.

About this task

This step is the first in the multi-step process described at “Adding the InterpreterSpecialty property to search” on page 59.

In this step, you make changes that affect only the ContactManager user interface. You add `InterpreterSpecialty` to `ABContactSearchCriteria.etx`. Then you add a `CriteriaDef` element to the ContactManager `search-config.xml` file to specify where the `InterpreterSpecialty` column is located.

Procedure

1. In Guidewire Studio for ContactManager, in the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Entity**, and then double-click `ABContactSearchCriteria.etx`.
2. In the editor, at the top of the **Element** hierarchy, click **nonPersistentEntity (extension)**.
3. Click the drop-down list next to ✚ and choose **column**.

   You use the **column** element rather than **typekey** because the data type is `varchar` and is not a typekey.

4. To the new column, add the following values that support searches for `IntepreterSpecialty`.

Name	Value
name	`InterpreterSpecialty`
type	`varchar`
nullok	`true`
desc	`Interpreter language specialties`

5. Click the drop-down list next to ✚ and choose **params**.
6. Enter the following values to specify the size of this `varchar` column:

Name	Value
name	`size`
value	`30`

7. In the **Project** window, navigate to **configuration** > **config** > **search** and double-click `search-config.xml` to edit the file.
8. Increment the `version` attribute in the `SearchConfig` element at the top of the file.
   In the following example, the original version number was `1` and has been changed to `2`.

```
<SearchConfig
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="search-config.xsd"
  version="2">
```

9. Find the following comment:

```
<!-- Specific fields in ABContact subtypes -->
```

10. After the last `CriteriaDef` element in this group, add a new `CriteriaDef` element for the target entity `ABInterpreter` that tests `IntepreterSpecialty` for equality:

```
<CriteriaDef entity="ABContactSearchCriteria" targetEntity="ABInterpreter">
  <Criterion property="InterpreterSpecialty" matchType="eq"/>
</CriteriaDef>
```

What to do next

"Add search support in ContactManager for Guidewire core applications" on page 60

See also

- For a complete description of `search-config.xml`, see the *Configuration Guide*

Add search support in ContactManager for Guidewire core applications

Add code for the new `InterpreterSpecialty` field to the Gosu class `ABContactAPISearchCriteria`. This Gosu object is populated by the core application to pass a search criterion to ContactManager.

Before you begin

Complete "Add search support to the ContactManager user interface" on page 59.

About this task

For this example, the new field will be added to the search results that ContactManager sends back to the Guidewire core application. To add the field, you enter code for the field in the Gosu class `ABContactAPISearchResult`.

Procedure

1. Add an entry for `InterpreterSpecialty` to the class `ABContactAPISearchCriteria`.
   a) In Guidewire Studio for ContactManager, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.webservice.ab.ab1000.abcontactapi`.
   b) Double-click `ABContactAPISearchCriteria` to open the class in the editor.
   c) Add the following variable definition to the list of variables at the beginning of the class.

   ```
   public var InterpreterSpecialty : String
   ```

   d) In the method `toSearchCriteria`, before `:AllTagsRequired`, add the following entry:

   ```
   :InterpreterSpecialty = this.InterpreterSpecialty,
   ```

2. If you want the field to be added to the search results sent back to a Guidewire core application, add an entry for `InterpreterSpecialty` to the class `ABContactAPISearchResult`.
   a) Press `Ctrl+N` and enter `ABContactAPISearchResult`.
   b) Click the class with package `gw.webservice.ab.ab1000.abcontactapi` in the search results to open the class in the editor.
   c) Add the following variable definition to the list of variables at the beginning of the class.

   ```
   public var InterpreterSpecialty : String
   ```

   d) In the constructor `construct(contact : ABContact)`, find the `if` statement block for `ABPerson`:

   ```
   if (contact typeis ABPerson) {
   ```

   e) After that `if` statement block, add a new `if` statement for `ABInterpreter`, as follows:

   ```
       // Search results for ABInterpreter
       if (contact typeis ABInterpreter) {
         this.InterpreterSpecialty = contact.InterpreterSpecialty
       }
   ```

What to do next

"Configure the address book search interface in ContactManager" on page 61

Configure the address book search interface in ContactManager

In this step of the search configuration example, you add the `InterpreterSpecialty` search field to the PCF files used in searching for address book contacts in ContactManager.

Before you begin

Complete "Add search support in ContactManager for Guidewire core applications" on page 60.

Procedure

1. If necessary, open Guidewire Studio for ContactManager.
2. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
3. Press `CTRL+F` and enter `Web.ContactSearch.IncludePendingCreates` to find this entry.
4. Add a line and enter the following display key:

   ```
   Web.ContactSearch.InterpreterSpecialty = Interpreter Specialty
   ```

5. Press `Ctrl+Shift+N` and enter `ContactSearchDV` to find this PCF file, and then double-click the file in the search results to open it in the editor.

6. Select the `ContactSearchDV` detail view panel.
7. Click the **Code** tab at the bottom of the window and find the following line of code:

```
function isDoctor(c : ABContactSearchCriteria) : boolean {
    return entity.ABDoctor.Type.isAssignableFrom(c.ContactSubtypeType )}
```

8. Insert a line after that one and enter on one line the following code:

```
function isInterpreter(c : ABContactSearchCriteria) : boolean {
return entity.ABInterpreter.Type.isAssignableFrom(c.ContactSubtypeType )}
```

9. In the `InputColumn` on the left, locate the `InputSet` containing the `Organization Name` input.
10. From the **Toolbox** on the right, drag an `InputSet` widget and drop it under the `InputSet` containing the `Organization Name` input.
11. Click the new `InputSet` and set the following property:

visible	`isInterpreter(SearchCriteria)`

12. Add an `Input` widget to the `InputSet` and set the following properties:

editable	`true`
id	`InterpreterSpecialty`
label	`displaykey.Web.ContactSearch.InterpreterSpecialty`
value	`SearchCriteria.InterpreterSpecialty`

What to do next

“Restart ContactManager and refresh web services in ClaimCenter” on page 62

Restart ContactManager and refresh web services in ClaimCenter

After you configure search for ContactManager, you must stop and restart ContactManager to force regeneration of the SOAP web services.

Before you begin

Complete “Configure the address book search interface in ContactManager” on page 61.

About this task

Before restarting ContactManager, you regenerate the ContactManager *Data Dictionary* to ensure that your changes to the entity `ABContactSearchCriteria` are valid. If they are, then you restart ContactManager and, in Guidewire Studio for ClaimCenter, you refresh the ContactManager web services.

Procedure

1. Open a command prompt in the ContactManager installation folder and then enter the following command:

```
gwb stopServer
```

2. Regenerate the dictionaries:

```
gwb genDataDictionary
```

3. If the dictionary regenerates without errors, start the ContactManager application:

```
gwb runServer
```

4. In Guidewire Studio for ClaimCenter, refresh the ContactManager plugin.

a) Press `Ctrl+Shift+N` and enter `ab1000.wsc` to find that web service collection, and then double-click the file in the search results to open it in the editor.

b) In the editor, select the following resource:

```
${ab}/ws/gw/webservice/ab/ab1000/abcontactapi/ABContactAPI?wsdl
```

c) Click **Fetch** to update the WSDL for this web service.

What to do next

"Add ContactManager search capability in ClaimCenter" on page 63

Add ContactManager search capability in ClaimCenter

Enable ClaimCenter to search for ContactManager contacts by `InterpreterSpecialty`.

Before you begin

Complete "Restart ContactManager and refresh web services in ClaimCenter" on page 62.

About this task

In this step you set search criteria and search results in ClaimCenter that work with ContactManager. The topic uses the new `Contact` subtype named `Interpreter` that has one field, `InterpreterSpecialty`.

Procedure

1. To support search on the `InterpreterSpecialty` field, add it to the entity `ContactSearchCriteria.etx`.

   a) In Guidewire Studio for ContactManager, in the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Entity**, and then double-click `ContactSearchCriteria.etx`.

   b) In the editor, at the top of the **Element** hierarchy, select **nonPersistentEntity (extension)**.

   c) Click the drop-down list next to ✚ and choose **column**.

   You use the **column** element because the data type is `varchar` and is not a typekey.

   d) To the new column, add the following values that support searches for `IntepreterSpecialty`:

Name	Value
name	`InterpreterSpecialty`
type	`varchar`
nullok	`true`
desc	`Interpreter language specialties`

   e) Click the drop-down list next to ✚ and choose **params**.

   f) Enter the following values to specify the size of this `varchar` column:

Name	Value
name	`size`
value	`30`

2. Add an entry for `InterpreterSpecialty` to the Gosu class `ContactSearchMapper` so you can use this field when you search for a contact.

   a) In Guidewire Studio for ClaimCenter, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.ab1000`.

   b) Double-click `ContactSearchMapper` to open it in the editor.

c) Press `CTRL+F` and enter `convertToABContactAPISearchCriteria` to find this method.

d) Find the following line of code:

```
searchCriteriaInfo.FirstNameKanji = searchCriteria.FirstNameKanji
```

e) After that line, add the following code:

```
searchCriteriaInfo.InterpreterSpecialty = searchCriteria.InterpreterSpecialty
```

3. Add an entry for `InterpreterSpecialty` to the Gosu class `ContactSearchResultMapper` so you can see this field in the search results that come back from ContactManager.

   a) In the same **Project** window folder, `gw.plugin.contact.ab1000`, double-click the class `ContactSearchResultMapper` to open it in the editor.

   b) Press `Ctrl+F` and enter `populateContactFromSearchResult` to find that method, and then in the method enter the following statement:

```
if (contact typeis Interpreter) {
  contact.InterpreterSpecialty = searchResult.InterpreterSpecialty
}
```

What to do next

“Configure the address book search interface in ClaimCenter” on page 64

Configure the address book search interface in ClaimCenter

In this step of the search configuration example, you add the `InterpreterSpecialty` search field to the PCF files used in searching for Address Book contacts in ClaimCenter.

Before you begin

Complete “Add ContactManager search capability in ClaimCenter” on page 63.

Procedure

1. If necessary, open Guidewire Studio for ClaimCenter.
2. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
3. Press `Ctrl+F` and enter the search text `Web.ContactSearch.IncludeSpecialistServices`, and then insert a new line after the one found by the search.
4. Enter the following display key on the new line:

   `Web.ContactSearch.InterpreterSpecialty = Interpreter Specialty`
5. Press `Ctrl+Shift+N` and enter `AddressBookSearchDV` to find this PCF file, and then double-click the file in the search results to open it in the editor.
6. In the `InputColumn` on the left, locate the `InputSet` containing the Law Firm Specialty `TypeKeyInput`, and then copy the input set and paste the copy below it.
7. Select the new `InputSet` and set the following property:

visible	`searchCriteria.isSearchFor(entity.Interpreter)`

8. Right-click the `TypeKeyInput` widget inside the `InputSet` and click **Change element type**.
9. Click `Input` in the list of element types, and then click **OK**.
10. Set the following properties for the input widget:

editable	`true`
id	`InterpreterSpecialty`
label	`displaykey.Web.ContactSearch.InterpreterSpecialty`
value	`searchCriteria.InterpreterSpecialty`

What to do next

"Test the InterpreterSpecialty search extensions" on page 65

Test the InterpreterSpecialty search extensions

Before you begin

Complete "Configure the address book search interface in ClaimCenter" on page 64.

Procedure

1. If the ContactManager server is running, stop and then restart ContactManager.
2. Open a command prompt in the ClaimCenter installation folder and then enter the following command:

   ```
   gwb stopServer
   ```

3. Regenerate the ClaimCenter *Data Dictionary* to test your changes to `ContactSearchCriteria`. At the command prompt, enter:

   ```
   gwb genDataDictionary
   ```

4. If the data dictionary regenerates successfully, start the ClaimCenter application:

   ```
   gwb runServer
   ```

5. Log in to ContactManager as a user who can create and edit contacts, such as user `aapplegate` with password `gw`.
6. Click **Search** on the left, and then choose **Interpreter** in the **Contact Type** list.
7. Search for one of the interpreters you created during your earlier tests.
   a) Do a search for that interpreter and use the interpreter specialty as a search criterion to see if the search returns that interpreter.
   b) Search for the same interpreter and use a specialty that is not a specialty for that interpreter to see if you get zero returns.
8. Log in to ClaimCenter and use the Address Book to search for an interpreter. Use the same search strategies as you did for ContactManager.

Adding an address field to Contact search

You can add an `Address` field to `Contact` search. The field can be either a normal address field or a denormalized address field.

Add a normal address field to search

You can add a normal address field, one that is not denormalized, to ClaimCenter search for ContactManager contacts.

About this task

The process for adding `Address` fields that are not denormalized fields to search is similar to the process for adding the `InterpreterSpecialty` property to search. However, the ClaimCenter part of the process is different enough that you might want to refer to the steps for adding a denormalized address field, starting with "step 6".

See also

- “Adding the InterpreterSpecialty property to search” on page 59
- For the ContactManager part of the process, see “Adding the InterpreterSpecialty property to search” on page 59.

Add a denormalized address field to search

Adding a denormalized field to search can improve the speed of searching.

About this task

If you want to search on a denormalized address field, add fields to the `Address` entity and the `ABContact` entity to give ContactManager an actual field to search for. On the ContactManager side, the field you set up for search is the field you add to `ABContact`. On the core application side, the field you search for is the `Address` field, as it would be for a regular, non-denormalized `Address` field search. The following steps show what you do differently for denormalized `Address` searches:

Procedure

1. Add a column for the denormalized field to the `Address` entity and set `SupportsLinguisticSearch` to `true`.
2. Add a `searchColumn` to the `ABContact` entity that has the same name as the column added to `Address`, plus `Denorm`.
   For example, if the `Address` column is named `County`, the `ABContact` column is named `CountyDenorm`. The `sourceColumn` is the same as the name for this search column, such as `CountyDenorm`, and the `sourceForeignKey` is `PrimaryAddress`.
3. The next steps are similar to those starting at “Add search support to the ContactManager user interface” on page 59. In general, for the ContactManager part of the process, use the `ABContact` search column as your field.
   For example:
   a) Add a column to `ABContactSearchCriteria.etx`, using the name of the `ABContact` search column.
   b) Add to `search-config.xml` a `<Criterion>` for the column you just added to `ABContactSearchCriteria.etx`. Add it to the following `<CriteriaDef>` for `ABContact`:

      ```
      <CriteriaDef entity="ABContactSearchCriteria" targetEntity="ABContact">
      ```

      The new criterion must use the name of the column in `ABContactSearchCriteria.etx` and can have any match type you prefer.
4. The next steps are similar to those starting at “Add search support in ContactManager for Guidewire core applications” on page 60. In general, for the ContactManager part of the process, use the `ABContact` search column as your field.
   For example:
   a) Add an entry for your `ABContact` search field to the class `ABContactAPISearchCriteria`.
   b) If you want the field to be added to the search results sent back to a Guidewire core application, add the `ABContact` search field to the class `ABContactAPISearchResult`.
5. Refresh the web services. See “Restart ContactManager and refresh web services in ClaimCenter” on page 62.
6. Go through the steps for the ClaimCenter part of the configuration, but use the `Address` field you want to have in search. This part of the configuration starts at “Add ContactManager search capability in ClaimCenter” on page 63.
   a) The entity `ContactSearchCriteria.etx` has a foreign key to `Address`, so any field on `Address` can be used in contact search, and there is no need to update this file.
   b) Add an entry for the `Address` field to `ContactSearchMapper` in the following `if` statement after the first line of code in that statement:

      ```
          if (searchCriteria.Address != null) {
            var address = new ABContactAPIAddressSearch()
      ```

For example, for `Address.County`, add the following new line of code:

```
address.County = searchCriteria.Address.County
```

c) The class `ContactSearchResultMapper` has a lot of the `Address` fields already in it. Check to see if a field you want to make visible for the address part of search might already be in the file. If you add a new field to `Address` that you want to search for, you must add an entry for the new `Address` field to this file.

7. Continue with the user interface configuration instructions, but use them as only a guideline for adding `Address` fields and do not follow them exactly. `Address` fields are located in different parts of the screens from `Contact` and `ABContact` fields. See "Configure the address book search interface in ContactManager" on page 61.

Adding the service state property to search

Adding a service state property to search is a multi-step process that is split into a series of topics. The service state property example uses the `ContactServiceState` entity and `ServiceState` property from the example in "Extending contacts with an array" on page 138.

In this example, you make the new `ServiceState` property searchable in both ClaimCenter and ContactManager. The property names must match in both applications. Any object you add to the search system must have indexes declared for its fields in ContactManager to make the search reasonably fast. In the example, `ContactServiceState` does have indexes declared for each of its fields. Without indexes, searching for the object can be slow.

Overview of adding search capability in ContactManager

Before starting the example of adding the ServiceState property to ContactManager and ClaimCenter search, be sure you understand how to use an `<ArrayCriterion>` element as described in this topic. Additionally, see "Adding the service state property to search" on page 67.

There are two primary aspects to adding search capability in ContactManager:

Adding search support to the ContactManager search screens

You add the `ServiceState` typekey to `ABContactSearchCriteria.etx`. Then you add an `<ArrayCriterion>` element to the ContactManager `search-config.xml` file to specify where the `ServiceState` column is located. There are restrictions on using `<ArrayCriterion>`:

- For any `<CriteriaDef>` element in the `search-config.xml` file, you must have only one `<ArrayCriterion>` subelement. If you have more than one `<ArrayCriterion>` defined for an entity, a search can return duplicate results for that entity. For example, you have two `<ArrayCriterion>` properties for `ABContact`. A successful search returns the same contact twice, once for each matched property.
- If you want to search for multiple fields in an array entity, create a new field that combines the fields.
- Additionally, the system requires unique values for the field that you are searching on. For example, `City` and `State` are two fields on an array entity, and the combination of the two fields is what makes the search unique. In this case, you must create a third field on the entity that concatenates the two values, and then create the array search criterion on that third field.

Adding search support for Guidewire core applications in the Gosu class `ABContactAPISearchCriteria`

You can have the search field show in the search results that ContactManager sends back to the Guidewire core application. To add the field to the search results, add code for the field to the Gosu class `ABContactAPISearchResult`.

These classes provide parameters used in web service calls, and changing them changes the web service WSDL. After changing one of these classes, you must restart ContactManager to regenerate the web APIs. Then, in Guidewire Studio for the Guidewire core application, you must refresh the ContactManager web APIs.

The first step in this example is "Enable ServiceState support for the ContactManager search screens" on page 68.

See also

- For detailed descriptions of `ArrayCriterion` and `search-config.xml`, see the *Configuration Guide*.

Enable ServiceState support for the ContactManager search screens

In the first step of this search example, you add support for the `ServiceState` property to enable its use in the ContactManager search screens.

Before you begin

Before you start this step, see the description of `<ArrayCriterion>` in "Overview of adding search capability in ContactManager" on page 67.

Procedure

1. In Guidewire Studio for ContactManager, in the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Entity** and then double-click `ABContactSearchCriteria.etx` to open this entity in the editor.
2. Right-click **nonPersistentEntity (extension)** at the top of the **Element** hierarchy and click **Add new**, and then choose **typekey** from the drop-down list.
3. Enter the following values:

Name	Value
name	`ServiceState`
typelist	`State`
desc	`State where contact provides services`

4. In the **Project** window, navigate to **configuration** > **config** > **search** and double-click `search-config.xml` to edit the file.
5. Increment the `version` attribute in the `SearchConfig` element at the top of the file.
   In the following example, the original version number was `1` and has been changed to `2`.

```
<SearchConfig
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="search-config.xsd"
  version="2">
```

6. Press `Ctrl+F` and enter `ABContact` to find the `<CriteriaDef>` element for the target entity `ABContact`.
7. Add a new `<ArrayCriterion>` element to the end of this `<CriteriaDef>` element, as follows:

```
<!-- Search by ABContact Fields -->
  <CriteriaDef entity="ABContactSearchCriteria" targetEntity="ABContact">
    <Criterion property="TaxID" matchType="eq"/>
    <Criterion property="VendorType" matchType="eq"/>
    <Criterion property="VendorAvailability" matchType="eq"/>
    <Criterion property="Keyword" matchType="startsWith"/>
    <Criterion property="KeywordKanji" matchType="startsWithCaseSensitive"/>
    <Criterion property="Score" matchType="ge"/>
    <!-- Some additional search criteria are defined in code -->
    <ArrayCriterion property="ServiceState"
                    targetProperty="ContactServiceArea"
                    arrayMemberProperty="ServiceState" />
  </CriteriaDef>
```

The attributes of the new element are:

`property`
   The `name` attribute for the column in the `ABContactSearchCriteria` entity

`targetProperty`
   The `name` of the array on the base entity `ABContact`

`arrayMemberProperty`
   The `name` of the column on the extension array entity `ContactServiceState`

What to do next

"Add ServiceState search support in ContactManager for core applications" on page 69

Add ServiceState search support in ContactManager for core applications

To support searching for contacts by the `ServiceState` property extension, add it to the ContactManager web API that core applications use for contact search.

Before you begin

Complete "Enable ServiceState support for the ContactManager search screens" on page 68. Also, it might be useful to review the information on the web API class `ABContactAPISearchCriteria` in "Overview of adding search capability in ContactManager" on page 67.

About this task

In this step of the search example, you add code for the `ServiceState` field to the Gosu class `ABContactAPISearchCriteria`. This web API class enables a Guidewire core application's **Search** screen to show the field to the user and pass its value to ContactManager.

Procedure

1. In Guidewire Studio for ContactManager, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.webservice.ab.ab1000.abcontactapi`.
2. Double-click `ABContactAPISearchCriteria` to open it in the editor.
3. Add the following variable definition to the list of variables at the beginning of the class:

   `public var ServiceState : typekey.State`
4. In the method `toSearchCriteria`, add a comma after the line starting with `:AllTagsRequired` and add a new entry for `ServiceState`:

```
:AllTagsRequired = this.AllTagsRequired,
:ServiceState = this.ServiceState
```

What to do next

"Configure ContactManager search screens for ServiceState" on page 69

Configure ContactManager search screens for ServiceState

Enable ContactManager users to search for contacts by `ServiceState` extension.

Before you begin

Complete "Add ServiceState search support in ContactManager for core applications" on page 69.

About this task

Add the `ServiceState` search field to the PCF files used in searching for contacts in ContactManager.

Procedure

1. In Guidewire Studio for ContactManager, open the **Project** window.
2. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
3. Press `Ctrl+F` and enter the search text `Web.ContactSearch.Person.LastName` to find this line in the file, and then add a new line after it.

4. Add the following new entry:

```
Web.ContactSearch.ServiceState = Service State
```

5. In the **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **search**, and then click `ContactSearchDV` to edit this PCF file.
6. In the `InputColumn` on the left, locate the `InputSet` containing the Organization Name input, then add an `InputSet` widget under it.
7. Click the new `InputSet` and set the following property:

visible	`isCompany(SearchCriteria)`

8. Add an `Input` widget to the `InputSet` and set the following properties:

editable	`true`
id	`ServiceState`
label	`displaykey.Web.ContactSearch.ServiceState`
value	`SearchCriteria.ServiceState`

What to do next

"Regenerate and refresh ContactManager web services" on page 70

Regenerate and refresh ContactManager web services

At this point in this multi-step search example, ClaimCenter does not have the web service changes you made in ContactManager. To get them, restart the ContactManager server to regenerate the web services, and then, in Guidewire Studio for ClaimCenter, refresh the ContactManager web service WSDL.

Before you begin

Complete "Configure ContactManager search screens for ServiceState" on page 69.

Procedure

1. Open a command prompt in the ContactManager installation folder and then enter the following command:

```
gwb stopServer
```

2. Start the ContactManager application:

```
gwb runServer
```

3. After ContactManager starts up, in Guidewire Studio for ClaimCenter, refresh the ContactManager plugin.
   a) In the **Project** window, press `Ctrl+Shift+N` and enter `ab1000.wsc`. Then click `ab1000.wsc` in the search results to open this web service collection in the editor.
   b) In the editor, select the following resource:

   ```
   ${ab}/ab/ws/gw/webservice/ab/ab1000/abcontactapi/ABContactAPI?wsdl
   ```

   c) Click **Fetch** to update the WSDL.

What to do next

"Add ServiceState search capability in ClaimCenter" on page 71

Add ServiceState search capability in ClaimCenter

Before you begin

Complete "Regenerate and refresh ContactManager web services" on page 70.

Procedure

1. Using Guidewire Studio for ClaimCenter, in the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Entity**.
2. Click `ContactSearchCriteria.etx` to open the entity extension in an editor.
3. Right-click **nonPersistentEntity (extension)** at the top of the **Element** hierarchy and click **Add new**, and then choose **typekey** from the drop-down list.
4. Enter the following values:

Name	Value
name	`ServiceState`
typelist	`State`
desc	`State where contact provides services`

5. Add an entry for `ServiceState` to the Gosu class `ContactSearchMapper` so you can use this field when you search for a contact.
   a) In Guidewire Studio for ClaimCenter, press `Ctrl+N` and enter `ContactSearchMapper`, and then in the search results click `ContactSearchMapper(gw.plugin.contact.ab1000)` to open this class in the editor.
   b) Press `Ctrl+F` and enter `convertToABContactAPISearchCriteria` to find this method in the class.
   c) Insert a new line after the following line of code:

   ```
   searchCriteriaInfo.PreferredVendors = searchCriteria.PreferredVendors
   ```

   d) Enter the following code for `ServiceState`:

   ```
   searchCriteriaInfo.ServiceState = searchCriteria.ServiceState.Code
   ```

What to do next

"Configure ClaimCenter search screens for ServiceState" on page 71

Configure ClaimCenter search screens for ServiceState

Enable ClaimCenter users to search for contacts by `ServiceState` property extension.

Before you begin

Before you start this step, complete the step "Add ServiceState search capability in ClaimCenter" on page 71.

About this task

Add the `ServiceState` search field to the PCF files used in searching for contacts in ClaimCenter.

Procedure

1. If necessary, open Guidewire Studio for ClaimCenter.
2. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
3. Press `Ctrl+F` and enter `Web.AddressBook.Search.SearchType` to find this line in the file, and then add a new line after it.

4. Add the following new entry:

   `Web.AddressBook.Search.ServiceState = Service State`

5. In the **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **addressbook** and double-click `AddressBookSearchDV` to edit this PCF file.
6. In the `InputColumn` on the left, locate the `InputSet` containing the Tax ID input, then add an `InputSet` widget under it.
7. Click the new `InputSet` and set the following property:

visible	`searchCriteria.isSearchFor(entity.Company)`

8. Add an `Input` widget to the `InputSet` and set the following properties:

editable	`true`
id	`ServiceState`
label	`displaykey.Web.AddressBook.Search.ServiceState`
value	`searchCriteria.ServiceState`

What to do next

"Test the ServiceState search extensions" on page 72

Test the ServiceState search extensions

Before you begin

Complete "Configure ClaimCenter search screens for ServiceState" on page 71.

Procedure

1. Shut down and restart both ClaimCenter and ContactManager.
   a) At a command prompt open in the ContactManager installation folder, enter the following command:
   ```
   gwb stopServer
   ```
   b) Start the ContactManager application:
   ```
   gwb runServer
   ```
   c) At a command prompt open in the ContactManager installation folder, enter the following command:
   ```
   gwb stopServer
   ```
   d) Start the ClaimCenter application:
   ```
   gwb runServer
   ```
2. Log in to ContactManager and click **Search** on the left, then choose **Company** in the **Contact Type** list.
3. Search for one of the companies you created during your earlier tests.
   a) If you have not done so already, create a new company and specify a service state. Then do a search for that company with the **Service State** specified in the search to see if the search returns that company.
   b) Search for the same company, first specifying a state that is not in the list of states for that company. See if you get zero returns.
   c) Do a search that includes the first letter of that company name, but no state specified.
   d) Do the same search with a state specified to see if the search results narrow to the companies with that value specified.

4. Log in to ClaimCenter and use the Address Book to search for a company that you know is stored in ContactManager. Use the same search strategies as you did for ContactManager.

Geocoding and proximity search for vendor contacts

ClaimCenter and ContactManager support *geocoding*, which enables the assignment of latitude and longitude to an address. By assigning latitude and longitude, the system is able to pinpoint an address as a location and specify its geographic coordinates. The application can then use these geographic coordinates to present data like the distance between two addresses or all the addresses in a certain radius.

You can configure geocoding to work with ClaimCenter and ContactManager.

See also

- For information on geocoding plugin integration, see the *Integration Guide*.

Geocoding and batch processing

The geocoding feature uses the Guidewire work queue infrastructure for asynchronous searches. For example, ClaimCenter Geocode batch processing searches asynchronously for new addresses at the times specified in the `scheduler-config.xml` file, as described at “Schedule geocoding” on page 78.

ClaimCenter Geocode batch processing geocodes all addresses in the database that have not yet been geocoded. ContactManager AB Geocode batch processing does the same thing.

Configuring the geocoding work queues

The ClaimCenter `work-queue.xml` file configures one `Geocode` worker to geocode addresses.

```
<work-queue
    workQueueClass="com.guidewire.pl.domain.geodata.geocode.GeocodeWorkQueue"
    progressinterval="600000">
    <worker instances="1" batchsize="100"/>
</work-queue>
```

The ContactManager `work-queue.xml` file configures one `ABGeocode` worker to perform geocode processing on addresses.

```
<work-queue
    progressinterval="600000"
    workQueueClass="com.guidewire.ab.domain.geodata.geocode.ABGeocodeWorkQueue">
    <worker batchsize="100" instances="1"/>
</work-queue>
```

You can modify these configurations, but first verify that you have a good understanding of the work queue infrastructure. For example:

- You can have more than one instance of worker.
- One concept is how `progressinterval` is used with `batchsize` by the system. The default setting of `progressinterval` is 600,000 milliseconds, or 10 minutes. This setting is the amount of time that ClaimCenter or ContactManager allots for a worker to process `batchsize` geocode work items. If the time a worker has held a batch of items exceeds the `progressinterval`, the work items become *orphans*. ClaimCenter reassigns orphan work items to a new worker instance. Therefore, you need to set the `progressinterval` larger than the longest time required to process a work item, multiplied by the `batchsize`.

You must restart ClaimCenter and ContactManager after changing any of these files.

IMPORTANT: If you have many new `User` or `Contact` objects in ClaimCenter or many new `ABContact` objects in ContactManager, processing these objects can be a system intensive operation. In this case, run the `Geocode` batch process when you anticipate that system use will be low, such as late at night.

See also

- For information on scheduling the batch process, see “Schedule geocoding” on page 78
- For information on work queues and the work queue scheduler, see the *Administration Guide*

BatchGeocode and GeocodeStatus properties in ContactManager

AB Geocode batch processing in ContactManager determines which addresses to geocode by checking the `BatchGeocode` and `GeocodeStatus` properties. ContactManager handles these properties as follows:

- The `Address` property `BatchGeocode` indicates whether or not an address is to be geocoded. The default value is `false`, meaning that addresses by default are not geocoded.
- There is a ContactManager rule that sets vendor contacts to be automatically geocoded. The rule—Set Vendor’s Primary Address BatchGeocode—is a Preupdate rule in the ABContactPreupdate rule set. This rule sets the `BatchGeocode` property of the primary address of an `ABContact` to `true` if the contact is tagged as a vendor.
- Geocode batch processing picks up addresses for which `BatchGeocode` is set to `true` and `GeocodeStatus` is set to `none`. This combination means that the address needs to be geocoded and has not been geocoded yet.

If you are adding many new contacts, especially into ContactManager, tune this parameter to match your expected daily load of new addresses.

See also

- “Schedule geocoding” on page 78

Run Geocode batch processing manually in ClaimCenter

About this task

You can run Geocode batch processing manually as a system administrator on the **Server Tools** screen.

Procedure

1. Log in as an administrator.
   For example, enter username `su` and password `gw`.
2. Press `Alt+Shift+T` to access the **Server Tools** screen.
   The screen opens with the **Server Tools** tab selected and the **Batch Process Info** item selected in the Sidebar.
3. Locate **Geocode Writer** and click its **Run** button.

Run Geocode batch processing manually in ContactManager

About this task

You can run Geocode batch processing manually as a system administrator on the **Server Tools** screen.

Procedure

1. Log in as an administrator.
   For example, username `su` and password `gw`.
2. Press `Alt+Shift+T` to access the **Server Tools** tab.
3. In the sidebar, click **Batch Process Info**.
4. Locate **AB Geocode Writer** and click its **Run** button.

What to do next

See also

- “Configuring the geocoding work queues” on page 73

Configuring geocoding for ClaimCenter and ContactManager

Configuring geocoding involves activating the geocoding plugin, setting `config.xml` parameters, and enabling the work queue and scheduler to run batch geocoding of addresses.

Configuring geocoding is a multi-step process.

IMPORTANT: If your environment includes a ContactManager integration, you must perform these steps in both ClaimCenter and ContactManager for proper functioning of the geocoding feature.

1. "Activate geocoding in Bing Maps" on page 75
2. "Enable and use the Geocoding plugin" on page 75
3. "Set geocoding configuration parameters in ClaimCenter" on page 76
4. "Set geocoding configuration parameters in ContactManager" on page 77
5. "Schedule geocoding" on page 78
6. "Log geocoding information" on page 79
7. "Stop and restart ClaimCenter and ContactManager" on page 79

Activate geocoding in Bing Maps

To use the geocoding plugin, your company must set up its own account, login, and application key with Bing Maps.

Procedure

1. Go to `http://www.bingmapsportal.com`, where you can set up a Bing Maps account and obtain an application key.

   When you create a key, the application name is arbitrary, and no application URL is required.
2. Register the geocoding plugin class that implements `GeocodePlugin` plugin interface.

   In the base configuration, this plugin implementation class is `gw.plugin.geocode.impl.BingMapsPluginRest`.

   Note: By default, the `GeocodePlugin` plugin interface uses the Microsoft Bing Maps web service and is disabled.

What to do next

"Enable and use the Geocoding plugin" on page 75

Enable and use the Geocoding plugin

For ClaimCenter and ContactManager to be able to use geocoding, you must enable the `GeocodePlugin` in both applications.

Before you begin

Complete "Activate geocoding in Bing Maps" on page 75.

About this task

The instructions for enabling the Geocoding plugin are the same for both ContactManager and ClaimCenter.

Procedure

1. Start Guidewire Studio™ for ClaimCenter and perform the steps that follow, and then start Guidewire Studio™ for ContactManager and perform the steps that follow.
2. In the **Project** window, navigate to **configuration** > **config** > **Plugins** > **registry** and then double-click `GeocodePlugin.gwp` to open this registry file in the editor.

3. Clear the **Disabled** check box to enable the plugin.
4. In the **Parameters** section, set the Geocode plugin parameters.

Parameter	Description
`applicationKey`	The application key that you obtained from Bing Maps. See "Activate geocoding in Bing Maps" on page 75.
`geocodeDirectionsCulture`	The locale for geocoded addresses and routing instructions returned from Bing Maps. For example, use the locale code `ja-JP` for addresses and instructions for Japan. The base application plugin implementation uses `en-US` if you do not specify a value. For a current list of codes that Bing Maps supports, see http://msdn.microsoft.com/en-us/library/cc981048.aspx.
`imageryCulture`	The language for map imagery. For example, use the language code `ja` for maps labeled in Japanese. The base application plugin implementation uses en if you do not specify a value. For a current list of codes that Bing Maps supports, see http://msdn.microsoft.com/en-us/library/cc981048.aspx.
`mapURLHeight`	Width of maps, in pixels. The base application plugin implementation uses `500` if you do not specify a value.
`mapURLWidth`	Height of maps, in pixels. The base application plugin implementation uses `500` if you do not specify a value.

What to do next

"Set geocoding configuration parameters in ClaimCenter" on page 76

Set geocoding configuration parameters in ClaimCenter

To enable geocoding features in the ClaimCenter user interface, you must set parameters in the `config.xml` file.

Before you begin

Complete "Enable and use the Geocoding plugin" on page 75.

About this task

If you have ContactManager integrated with ClaimCenter, you must consider both applications when you set some of these configuration parameters.

Procedure

1. Open Guidewire Studio™ for ClaimCenter.
2. In the **Project** window, navigate to **configuration** > **config**.
3. Double-click `config.xml` to open this file in an editor.
4. In the editor, press `Ctrl+F` and then enter `Geocoding` to find that section of the configuration parameters.
5. Set the following parameters to enable geocoding.

Parameter	Description
`UseGeocodingInPrimaryApp`	Set this parameter to `true` to enable geographical data and proximity search on application screens in `ClaimCenter`. The setting affects screens like **Assignment** and **User** search windows. This parameter must be `true` to perform assignment by proximity. The default setting is `false`.
`UseGeocodingInAddressBook`	Set this parameter to `true` to enable geocoding on ClaimCenter **Address Book** screens if ClaimCenter is integrated with ContactManager. The default setting is `false`.

Parameter	Description
`UseMetricDistancesByDefault`	Use kilometers in the user interface. The default setting, `false`, specifies that miles are to be used.
`ProximitySearchOrdinalMaxDistance`	A distance that provides an approximate bound to improve performance of an ordinal (nearest *n*) proximity search. The search can return results that are farther away than the distance specified. The default setting is `300`. The actual unit value of the distance is miles or kilometers depending on how you have set `UseMetricDistancesByDefault`. The setting for this parameter must be the same in ContactManager and ClaimCenter. This parameter has no effect on *radius* (within *n* miles or kilometers) proximity searches or walking-the-group-tree-based proximity assignment.
`ProximityRadiusSearchDefaultMaxResultCount`	Maximum number of results to return when performing a radius (*n* miles or kilometers) search from ClaimCenter. • ClaimCenter passes the value of this parameter to ContactManager when it makes a call for a proximity search. • The default value in the base configuration is `1000`. • This parameter has no effect on ordinal (nearest *n*) proximity searches. • This parameter applies only to searches originating from ClaimCenter and does not have to match the value of the corresponding parameter in the ContactManager `config.xml` file.

What to do next

"Set geocoding configuration parameters in ContactManager" on page 77

Set geocoding configuration parameters in ContactManager

To enable geocoding features in the ContactManager user interface, you must set parameters in the `config.xml` file.

Before you begin

Complete "Set geocoding configuration parameters in ClaimCenter" on page 76.

Procedure

1. In Guidewire Studio for ContactManager, in the **Project** window, navigate to **configuration** > **config**.
2. Navigate to the bottom of that node, and then double-click `config.xml` to open the file in an editor.
3. In the editor, press `Ctrl+F` and then enter `Geocoding` to find that section of the configuration parameters.
4. Set the following parameters to enable geocoding.

Parameter	Description
`UseGeocodingInAddressBook`	Set this parameter to `true` to enable geocoding for contacts storied in ContactManager. The default is `false`. The setting for this parameter must be the same in ContactManager and ClaimCenter.

Parameter	Description
`UseMetricDistancesByDefault`	Use kilometers in the user interface. The default, `false`, is to use miles. The setting for this parameter must be the same in ContactManager and ClaimCenter.
`ProximitySearchOrdinalMaxDistance`	Maximum distance to use when performing an ordinal (nearest *n*) proximity search. The default is `300`. The actual unit value of the distance is miles or kilometers depending on how you have set `UseMetricDistancesByDefault`. This parameter has no effect on radius (*n* miles or kilometers) searches or proximity assignment based on walking the group tree. The setting for this parameter must be the same in ContactManager and ClaimCenter.
`ProximityRadiusSearchDefaultMaxResultCount`	Maximum number of results to return when performing a radius (*n* miles or kilometers) search from the ContactManager search screen. • The default value in the base configuration is `1000`. • This parameter has no effect on ordinal (nearest *n*) proximity searches. • This parameter applies only to searches originating from ContactManager and does not have to match the value of the corresponding parameter in the ClaimCenter `config.xml` file.

What to do next

"Schedule geocoding" on page 78

Schedule geocoding

If you enable geocoding, you must also schedule the work queues that geocode addresses that have not been geocoded yet.

Before you begin

Complete "Set geocoding configuration parameters in ContactManager" on page 77.

About this task

When you schedule the geocoding work queue, use the following guidelines:

- Schedule `Geocode` and `ABGeocode` batch processing with enough time between runs to fully process the work items in the work queues. If you find duplicate work items in the work queues for the same address ID, make the time between runs a longer interval.
- Geocoding a large number of addresses can require considerable application resources. Set up your implementation to do its batch geocoding of addresses at a time when general access to your server is restricted.
- The `ABGeocode` batch process in ContactManager uses the settings described at "BatchGeocode and GeocodeStatus properties in ContactManager" on page 74.
- See also "Geocoding and batch processing" on page 73.

Procedure

1. In Guidewire Studio for ClaimCenter, press `Ctrl+Shift+N` and enter `scheduler-config.xml`, and then double-click `scheduler-config.xml` in the search results to open it in the editor.
2. In the editor, press `CTRL+F` and enter `Geocode` to find the following entry:

The following XML code shows the entry in the base configuration of ClaimCenter, including the `<!--` and `-->` comment tags:

```
<!--  New addresses searched for geocoding at 1:30 am  -->
<!--
  <ProcessSchedule process="Geocode">
    <CronSchedule hours="1" minutes="30"/>
  </ProcessSchedule>
-->
```

This entry, when uncommented, instructs the batch process to start searching for new addresses to geocode every night at 1:30 AM.

3. Remove the opening `<!--` and closing `-->` comments to activate this entry the next time you start the application.
4. If ClaimCenter is integrated with ContactManager, you must also uncomment the `ABGeocode` entry in the ContactManager `scheduler-config.xml` file. In Guidewire Studio for ContactManager, follow the same procedure you did for ClaimCenter to open the file and find the entry.
   The following XML code shows the entry in the base configuration of ContactManager, including the `<!--` and `-->` comment tags:

```
<!--   <ProcessSchedule process="ABGeocode">
         <CronSchedule minutes="0"/>
       </ProcessSchedule> -->
```

   This entry, when uncommented, instructs the work queue to start searching for new addresses to geocode every hour on the hour.
5. Remove the opening `<!--` and closing `-->` comments to activate this entry the next time you start the application.

What to do next

"Log geocoding information" on page 79

Log geocoding information

Logging information for the geocoding plugins enables you to review results of the geocoding work queues.

Before you begin

Complete "Schedule geocoding" on page 78.

About this task

Perform the following tasks in both Guidewire Studio for ClaimCenter and Guidewire Studio for ContactManager.

Procedure

1. Navigate in the **Project** window to **configuration** > **config** > **logging** and then double-click `log4j2.xml`.
2. Configure the sections of this file that add details of geocoding plugin activity to the log.

What to do next

"Stop and restart ClaimCenter and ContactManager" on page 79

Stop and restart ClaimCenter and ContactManager

After making configuration and logging file changes for geocoding, you must stop the Guidewire applications that you configured if they are running, and then rebuild and redeploy them.

Before you begin

Complete "Log geocoding information" on page 79.

About this task

Following are the steps for stopping and starting both ClaimCenter and ContactManager when they are running in development mode:

Procedure

1. Stop ClaimCenter.

   Open a command prompt in the ClaimCenter installation folder and then enter the following command:

   ```
   gwb stopServer
   ```

2. Stop ContactManager.

   Open a command prompt in the ContactManager installation folder and then enter the following command:

   ```
   gwb stopServer
   ```

3. Start the ContactManager application in the ContactManager installation folder:

   ```
   gwb runServer
   ```

4. Start the ClaimCenter application in the ClaimCenter installation folder:

   ```
   gwb runServer
   ```

Proximity search example for vendor contacts

Your search results are returned from ContactManager ordered by distance, closest to farthest away. In the base configuration, this order of search results is the only one available.

The system sorts the results of a proximity search, but it does not display driving directions with the search results. To obtain driving directions, you must select a return address and click the **Return Driving Directions** button. Then additional columns for driving distances and times appear, as well as links to display the directions for the requested search results.

Note: You cannot sort driving direction results.

Request distanced-based proximity search

This example shows you how to request a distance-based proximity search from the ClaimCenter user interface and describes what the system does in response to the search request.

Before you begin

To walk through this example in ClaimCenter, you must have done the following:

- Integrated ContactManager with ClaimCenter as described in “Integrating ContactManager with Guidewire core applications” on page 21.
- Loaded sample data as described at “Load sample data for ContactManager” on page 18.
- Set up geocoding for both applications as described in “Configuring geocoding for ClaimCenter and ContactManager” on page 75.
- Run `Geocode` batch processing for ClaimCenter and `ABGeocode` batch processing for ContactManager as described in:
  - “Run Geocode batch processing manually in ClaimCenter” on page 74
  - “Run Geocode batch processing manually in ContactManager” on page 74

About this task

In this example, you request the closest auto repair shops within 25 miles of San Mateo, California, USA.

Procedure

1. Log in to ClaimCenter.
2. Click the **Address Book** tab.
3. On the **Search Address Book** screen, choose the following settings:

Type	`Auto Repair Shop`
Search Radius	`25 miles`
City (under Search Radius)	`San Mateo`
ZIP Code	`94404`

4. Click **Search**.
5. ClaimCenter determines the center of the proximity search by synchronously geocoding this one address.
6. The system applies any filters that you set in the search. For example, if under **Location** you set the **City** to `Burlingame` and the **State** to `California`, the search eliminates every address that does not have a city set to Burlingame. Any address without a latitude and longitude is not considered.
7. ClaimCenter calculates the distance of the remaining results from the search center. If you limited the range as shown previously, ClaimCenter discards any results that are too far away.
8. ClaimCenter sorts the remaining results in ascending order of proximity, from closest to farthest away, and returns them in the **Search Results**.

PolicyCenter support for Contact searches

PolicyCenter enables you to search both for locally stored contacts and for contacts stored externally in ContactManager.

To configure `Contact` search in PolicyCenter, you edit the following files in Guidewire Studio for PolicyCenter:

- `search-config.xml` – Defines search criteria for addresses.
- `gw.plugin.contact.ab1000.ABContactSystemPlugin` – The plugin implementation used to call ContactManager web services. This plugin implementation has a `searchContacts` method that calls the ContactManager `ABContactAPI.searchContact` method.
- `gw.plugin.contact.ab1000.ABContactAPISearchCriteriaEnhancement` – The class that defines additional PolicyCenter search criteria to be used with ContactManager contact searches. The class enhances the ContactManager web service `ABContactAPISearchCriteria`, adding a `sync` method that is used by `ABContactSystemPlugin.searchContacts`.
- `gw.plugin.contact.ab1000.ABContactAPISearchResultEnhancement` – The class that defines the search result fields that PolicyCenter displays in the search results returned from ContactManager.
- `ContactSearchCriteria.eix` – You can extend this entity by creating a `ContactSearchCriteria.etx` file. This entity defines the search criteria that are shown to the user in the search screens. If you add search criteria to `ContactSearchCriteria.etx`, you must also add them to `ABContactAPISearchCriteriaEnhancement`. PolicyCenter uses both files.
- `gw.plugin.contact.impl.ContactSearchCriteriaEnhancement` – The class in which the contact search method for both internal and external contact search is defined. The primary contact search method in this class is `performSearch`.
- `ContactSearchScreen.pcf` – This PCF file displays the contact search criteria. It also instantiates an instance of the `ContactSearchCriteria` entity and uses it as a parameter to its `doSearch` method defined in the **Code** tab. That method calls the method `ContactSearchCriteriaEnhancement.performSearch`.

In the base configuration, PolicyCenter uses the `Address` fields defined in `search-config.xml` to search only for address fields. PolicyCenter does not use this file to configure searching for `Contact` fields.

To search for contacts stored in ContactManager, PolicyCenter uses the plugin implementation `gw.plugin.contact.ab1000.ABContactSystemPlugin`. This class's `searchContacts` method instantiates an `ABContactAPISearchCriteria` object and calls its `sync` method, which in turn calls the ContactManager web service method `ABContactAPI.searchContact`. The PolicyCenter `searchContacts` method returns data from the `ABContactAPISearchResult` Gosu objects returned by the ContactManager `searchContact` method. PolicyCenter shows these returned objects as contacts and their fields in the search results.

One class where you can customize this contact search behavior is `gw.plugin.contact.ab1000.ABContactAPISearchCriteriaEnhancement`. This class extends the search criteria defined in the ContactManager web service `ABContactAPISearchCriteria`. For example, the `sync` method specifically searches for the client tag. If you want to search for other tags, you can modify the code of this method to do so.

A corresponding class in which you can customize contact search behavior is `gw.plugin.contact.ab1000.ABContactAPISearchResultEnhancement`. This class defines extensions to the search results returned by ContactManager. It enhances the ContactManager web service `ABContactAPISearchResult`.

See also

- For information on the ContactManager files involved in an external contact search, see "ContactManager support for core application searches" on page 58.

BillingCenter support for Contact searches

BillingCenter enables you to search both for locally stored contacts and for contacts stored externally in ContactManager.

To configure `Contact` search in BillingCenter, you edit the following files in Guidewire Studio for BillingCenter:

- `search-config.xml` – Defines search criteria for addresses.
- `gw.plugin.contact.ab1000.ABContactSystemPlugin` – The plugin implementation used to call ContactManager web services. This plugin implementation has a `searchContacts` method that calls the ContactManager `ABContactAPI.searchContact` method.
- `gw.plugin.contact.ab1000.ABContactSearchCriteriaInfoEnhancement` – The class that defines additional BillingCenter search criteria to be used with ContactManager contact searches. The class enhances the ContactManager web service `ABContactAPISearchCriteria`, adding a `sync` method that is used by `ABContactSystemPlugin.searchContacts`.
- `gw.plugin.contact.ab1000.ContactResultFromSearch` – The class that defines the search result fields that BillingCenter displays in the search results returned from ContactManager.
- `ContactSearchCriteria.eix` – You can extend this entity by creating a `ContactSearchCriteria.etx` file. `ContactSearchCriteria` defines the search criteria that are shown to the user in the search screens. If you add search criteria to `ContactSearchCriteria.etx`, you must also add them to `ABContactSearchCriteriaInfoEnhancement`. BillingCenter uses both files.
- `gw.plugin.contact.impl.ContactSearchCriteriaEnhancement` – The class in which the contact search method for both internal and external contact search is defined. The primary contact search method in this class is `performSearch`.
- `ContactSearchScreen.pcf` – This PCF file contains the display view in which the contact search criteria are shown. `ContactSearchScreen.pcf` also instantiates an instance of the `ContactSearchCriteria` entity and uses it as a parameter to its `doSearch` method defined in the **Code** tab. That method calls the method `ContactSearchCriteriaEnhancement.performSearch`.
- `ContactSearchDV.pcf` – This PCF file contains the contact search criteria shown in the search screen. This file is where you add new search criteria.

In the base configuration, BillingCenter uses the `Address` fields defined in `search-config.xml` to search for address fields. However, to maintain performance, any search that uses the `Address` field must also include at least one of the following fields: `Company Name`, `First Name`, or `Last Name`. These minimally-required search criteria are defined in the `isReasonablyConstrainedForSearch` method in the `ContactCriteria` class.

To search for contacts stored in ContactManager, BillingCenter uses the plugin implementation `gw.plugin.contact.ab1000.ABContactSystemPlugin`. This class's `searchContacts` method instantiates an `ABContactAPISearchCriteria` object and calls its `sync` method, which in turn calls the ContactManager web service `ABContactAPI.searchContact`. The BillingCenter `searchContacts` method returns data from the `ABContactAPISearchResult` Gosu objects returned by the ContactManager `searchContact` method. BillingCenter shows these returned objects as contacts with their fields in the search results.

One class in which you can customize this contact search behavior is `gw.plugin.contact.ab1000.ABContactSearchCriteriaInfoEnhancement`. This class enhances the search criteria defined in the ContactManager web service `ABContactAPISearchCriteria`. For example, the `sync` method of this class specifically searches for the Client tag. If you want to search for other tags, you can modify the code of this method to do so.

A corresponding class where you can customize contact search behavior is `gw.plugin.contact.ab1000.ContactResultFromSearch`. This class defines extensions to the search results returned by ContactManager. It extends the ContactManager web service `ABContactAPISearchResult`.

See also

- For information on the ContactManager files involved in an external contact search, see "ContactManager support for core application searches" on page 58.

chapter 5

Securing access to contact information

You can grant secure access to the information associated with contacts, both in ContactManager and in the core applications.

A fundamental component of enforcing security on shared contacts is checking permissions for users of the Guidewire core application. Contact security can be enforced both by a core application and by ContactManager as users create, update, and delete contacts. Additionally, users of ContactManager with appropriate permissions can merge duplicate contacts and review pending contact updates and creation.

Each Guidewire application has its own rules and security setup.

See also

- "ContactManager contact security" on page 85
- "PolicyCenter contact security" on page 96
- "Security: Roles, Permissions, and the Community Model" in the *PolicyCenter Application Guide*
- "Authority Profiles" in the *PolicyCenter Application Guide*
- "BillingCenter contact security" on page 97
- "Security" in the *BillingCenter Application Guide*
- "ClaimCenter contact security" on page 99
- "Configuring ClaimCenter contact security" on page 102

ContactManager contact security

ContactManager is built on the same platform as the Guidewire core applications and provides roles and permissions that you can configure to control access to contact data. Additionally, the permissions have a set of permission check expressions you can configure to limit access to specific screens and widgets.

Role-based security defines what actions a user of a Guidewire application is allowed to perform. This type of security includes defining permissions, grouping related permissions into roles, and assigning these roles to users based on the work they perform in the Guidewire application.

For most purposes, other than merging duplicate contacts and reviewing pending contact changes, you typically access ContactManager data through a core application, like ClaimCenter, PolicyCenter, or BillingCenter. Those applications provide contact security to control which users of the application can access contact data.

You can set up ContactManager users to access contact data directly in ContactManager. The base configuration provides a role for this purpose, the ContactManager role.

The primary tasks that users have to perform in ContactManager are:

Merging duplicate contacts
See “Detecting and merging duplicate contacts” on page 271.

Reviewing pending contact changes
See “Review pending changes to contacts” on page 278.

See also

- “ContactManager user roles” on page 86
- “PolicyCenter contact security” on page 96
- “BillingCenter contact security” on page 97
- “ClaimCenter contact security” on page 99

ContactManager user roles

A role is a collection of permissions. By grouping permissions into roles, you can define the authority of a user of ContactManager by assigning the user a few roles rather than a larger list of permissions. A user can have multiple roles and must have at least one role.

The permissions used for contact access in the base configuration of ContactManager are `abcreate`, `abcreatepref`, `abdelete`, `abdeletepref`, `abedit`, `abeditpref`, `abview`, `abviewpending`, `abviewmerge`, `abviewpending`, `abviewsearch`, `anytagcreate`, `anytagdelete`, `anytagedit`, and `anytagview`.

These permissions are described in more detail in the topic, “ContactManager contact subtype and tag permissions” on page 89.

You can create additional permissions for certain contacts, contact subtypes, or tags. For example, you can create a permission for working with client contacts and another for working with vendor contacts.

To add permissions to roles and assign roles to users, use the ContactManager **Roles** screen. Log in as a user with administration privileges and click the **Administration** tab, and then navigate in the sidebar to **Users & Security** > **Roles**. See “Configuring ContactManager contact security” on page 92.

The base configuration of ContactManager provides the following roles, each of which has a set of default permissions. It is likely that you will add your own roles and permissions as well.

Role	Permissions	Description
Client Application	• Client Application – `clientapp` • Create address book contacts – `abcreate` • Create address book preferred vendors – `abcreatepref` • Add documents to a contact – `doccreate` • Create contact with any tag – `anytagcreate` • Delete address book contacts – `abdelete` • Delete address book preferred vendors – `abdeletepref` • Delete contact with any tag – `anytagdelete` • Edit address book contacts – `abedit` • Edit address book preferred vendors – `abeditpref` • Edit contact with any tag – `anytagedit` • Edit documents – `docedit` • Edit user language – `usereditlang` • Remove documents from a contact – `docdelete`	This role is used by core applications to communicate with ContactManager. It contains permissions that allow the core applications to perform specific tasks.

Role	Permissions	Description
	• View address book contact search screens – `abviewsearch` • View address book contacts – `abview` • View contact with any tag – `anytagview` • View documents – `docview`	
ContactManager	• Create address book contacts – `abcreate` • Create address book preferred vendors – `abcreatepref` • Create contact with any tag – `anytagcreate` • Add documents to a contact – `doccreate` • Delete address book contacts – `abdelete` • Delete address book preferred vendors – `abdeletepref` • Delete contact with any tag – `anytagdelete` • Edit address book contacts – `abedit` • Edit address book preferred vendors – `abeditpref` • Edit contact with any tag – `anytagedit` • Edit documents – `docedit` • Edit user language – `usereditlang` • View address book contact search screens – `abviewsearch` • View address book contacts – `abview` • View contact with any tag – `anytagview` • View documents – `docview` • View merge – `abviewmerge` • View pending – `abviewpending`	User with full permission to create, edit, and delete contacts.
Contact Subtype Changer	• Change Contact Subtype – `changecontactsubtype` • SOAP administration – `soapadmin`	User with permissions to change the subtype of a contact instance.
Contact Viewer	• Edit user language – `usereditlang` • View address book contact search screens – `abviewsearch` • View address book contacts – `abview` • View contact with any tag – `anytagview`	User with view-only permissions for contacts
Data Protection Officer	• Create groups – `groupcreate` • Delete groups – `groupdelete` • Edit groups – `groupedit` • Edit obfuscated user contact – `editobfuscatedusercontact` • Edit user language – `usereditlang` • Edit users – `useredit` • Grant roles to users – `usergrantroles` • Request Contact Destruction – `requestcontactdestruction` • View all users – `userviewall`	User who can respond to failures in contact purging or obfuscation with corrections

Role	Permissions	Description
	• View group tree – `grouptreeview` • View groups – `groupview` • View user – `userview`	
Rule Admin	• Administer rules – `ruleadmin` • Edit user language – `usereditlang`	Rule administrator
Tools View	• View BatchProcess tools screen – `toolsBatchProcessview` • View Cache Info screen – `toolsCacheinfoview` • View Cluster tools screen – `toolsClusterview` • View Info tools screen – `toolsInfoview` • View Log tools screen – `toolsLogview` • View ManagementBeans tools screen – `toolsJMXBeansview` • View Profiler tools screen – `toolsProfilerview` • View StartablePlugin tools screen – `toolsPluginview` • View WorkQueue tools screen – `toolsWorkQueueview`	User with permission to work on the **Server Tools** screens. To access **Server Tools**, press `Alt+Shift+T` and click **Server Tools**.
User Admin	• Always access debug tools – `usereditattrs` • Create groups – `groupcreate` • Create users – `usercreate` • Delete groups – `groupdelete` • Delete users – `userdelete` • Edit groups – `groupedit` • Edit user language – `usereditlang` • Edit users – `useredit` • Grant roles to users – `usergrantroles` • Manage attributes – `attrmanage` • Manage holidays – `holidaymanage` • Manage regions – `regionmanage` • Manage roles – `rolemanage` • Manage script parameters – `scrprmmanage` • Manage security zones – `seczonemanage` • Resync message – `resyncmessage` • Retry message – `retrymessage` • Skip message – `skipmessage` • SOAP administration – `soapadmin` • View attributes – `attrview` • View event messages – `eventmessageview` • View group tree – `grouptreeview` • View groups – `groupview` • View holidays – `holidayview`	User who handles administration of ContactManager users.

Role	Permissions	Description
	• View regions – `regionview` • View roles – `roleview` • View script parameters – `scrprmview` • View user – `userview`	

See also

- "Document security" on page 184

ContactManager contact subtype and tag permissions

The Guidewire core applications and ContactManager provide contact subtype and tag permissions that you can use to control access to contacts. The `SystemPermissionType` typelist lists all the subtype and tag permissions in ContactManager.

The following table lists the subtype and tag permissions provided in the base configuration of ContactManager for contacts:

Code	Permission Description
`abview`	View the details of contact entries in ContactManager
`abviewsearch`	View ContactManager contact search screens
`anytagcreate`	Create a new contact regardless of which contact tag it requires
`anytagdelete`	Delete a contact that has any contact tag
`anytagedit`	Edit a contact that has any contact tag
`anytagview`	See a contact that has any contact tag
`abcreate`	Create a new contact in ContactManager
`abcreatepref`	Create a new preferred vendor in ContactManager
`abdelete`	Delete an existing contact from the address book
`abdeletepref`	Delete an existing preferred vendor address book entry
`abedit`	Edit an existing contact in ContactManager
`abeditpref`	Edit an existing preferred vendor in ContactManager
`abviewmerge`	Review and merge duplicate contacts in the **Merge Contacts** screens
`abviewpending`	Review and approve or disapprove pending contacts in the **Pending Changes** screens

The system uses role-based security for these permissions. As described in the previous topic, to implement role-based security, a system administrator associates permissions with roles in the system and assigns roles to users. For each role assigned, the user acquires the permissions associated with that role. For example, a role associated with the `abcreate` and `anytagcreate` permissions enables the user who has this role to create any type of contact.

The contact and tag permissions supplied in the base configuration apply across all contact subtypes and tags. If you create a permission that applies to a contact subtype, that permission also applies to all the subtypes of that contact subtype.

ContactManager enables you to restrict permissions according to contact subtype or tag. For example, you can enable a user with `abcreate` permission to create only `PersonVendor` contacts, but not `CompanyVendor` contacts. You configure contact and tag permissions through the `SystemPermissionType` typelist and the `security-config.xml` resource.

> **Note:** If you create a set of tag permissions for a specific tag, these permissions enable access to contacts that have only that tag. For example you create a set of Vendor tag permissions and a user has a role with only those tag permissions. That user will not be able to work with a contact that has both Claim Party and Vendor tags. You could also create a set of tag permissions for Claim Party tags. In that case, a user with both Vendor and Claim Party tag permissions would be able to work with contacts that have both Vendor and Claim Party tags.

For examples showing how to create and use permissions for a contact subtype and a contact tag, see “Configuring ContactManager contact security” on page 92.

ContactManager contact and tag permission check expressions

In a page configuration file (PCF), you can control permissions on specific widgets with Gosu expressions that determine if a user has permission to perform an operation.

For example, in the ContactManager `NewContact.pcf` file, the `canVisit` attribute is set to:

```
perm.ABContact.create(ContactType) and
ContactTagType.userHasCreatePermissionForAtLeastOneContactTagType()
```

These expressions control access to this page. They allow access only to users who have both subtype permission to create the contact and at least one tag permission to create a tag.

Note: To see this PCF file, in Guidewire Studio™ for ContactManager, press `Ctrl+Shift+N` and enter `NewContact`, and then double-click `NewContact.pcf (configuration\config\web\pcf\contacts)` in the list of objects that the system finds.

Permission check expression parameters in ContactManager

Some Gosu permission check expressions require a parameter, and some do not.

For example, the ContactManager tag create permission check expression is the same as the `ABContact` create permission check expression: `perm.ABContact.create`. When this expression has a `ContactTagType` argument, the expression verifies that the user has permission to create a contact with this tag. The `ContactTagType` arguments for this expression in the base configuration support the following enumeration constants:

- `ContactTagType.TC_CLIENT`
- `ContactTagType.TC_VENDOR`
- `ContactTagType.TC_CLAIMPARTY`

For example, the expression `perm.ABContact.create(ContactTagType.TC_CLIENT)` verifies that the user has permission to create a new contact with the Client tag. If the user has `anytagcreate` permission, which grants permission for creating contacts with all contact tag types, this permission check returns `true`.

The contact and tag create permission expression can take an `ABContact` type or subtype parameter or a `ContactTagType` typecode specified as an enumeration constant. The contact and tag delete, edit, and view permission expressions take an `ABContact` instance as a parameter. All these expressions check for both contact and tag permissions, so no additional tag permission check expression is needed. These expressions are:

- `perm.ABContact.create`
- `perm.ABContact.delete`
- `perm.ABContact.edit`
- `perm.ABContact.view`

The following table lists the ContactManager contact and tag permission check expressions:

ContactManager Expression	Description
`perm.ABContact.create`	Takes an input parameter that is either an `ABContact` type or subtype or a `ContactTagType` typecode specified as an enumeration constant. Depending on the parameter, verifies that the user has the permission to create either a contact with the specified tag or a contact of the specified type.
`perm.ABContact.createpreferred`	Does not take an input parameter. Verifies that the user has permission to add the preferred contact in the address book.

ContactManager Expression	Description
`perm.ABContact.delete`	Takes an `ABContact` instance as an input parameter. Determines the subtype and contact tag or tags from the instance, and then verifies that the user has the contact and tag permissions to delete the contact.
`perm.ABContact.deletepreferred`	Does not take an input parameter. Verifies that the user has permission to delete the preferred vendor from the address book.
`perm.ABContact.edit`	Takes an `ABContact` instance as an input parameter. Determines the subtype and contact tag or tags from the instance, and then verifies that the user has the contact and tag permissions to edit the contact.
`perm.ABContact.editpreferred`	Does not take an input parameter. Verifies that the user has permission to edit the preferred vendor in the address book.
`perm.ABContact.view`	Takes an `ABContact` instance as an input parameter. Determines the subtype and contact tag or tags from the instance, and then verifies that the user has the contact and tag permissions to view the contact.
`perm.ABContact.viewmerge`	Does not take an input parameter. Verifies that the user has permission to merge contacts in the address book.
`perm.ABContact.viewpending`	Does not take an input parameter. Verifies that the user has permission to view pending contacts in the address book.
`perm.ABContact.viewsearch`	Does not take an input parameter. Verifies that the user has permission to search for contacts in the address book.

Viewing permissions in the ContactManager Security Dictionary

You can use the *Security Dictionary* to get information on the application permission keys. For example, after opening the *Security Dictionary*, click the **System Permissions** filter at the top of the left pane. You see all the permissions listed on the left by code name. If you click the permission code **abedit**, you see that it has the related application permission key **ABContact edit**, which has the associated Gosu check expression `perm.ABContact.edit`. You can filter the list by application permission keys, pages, system permissions, and roles.

Using the *Security Dictionary*, you can determine the following:

- The system permission related to an application permission key
- PCF files and widgets that use an application permission key
- Roles, application permission keys, PCF pages, and widgets that use a system permission
- Gosu application permission expressions called from each PCF page
- Permissions assigned to each role

See also

- "Build and view the ContactManager Security Dictionary" on page 91

Build and view the ContactManager Security Dictionary

Procedure

1. At a command prompt, run the following command from the ContactManager installation folder:

   ```
   gwb genDataDictionary
   ```

   This command builds the *Security Dictionary* in the following location:

   `ContactManager/build/dictionary/security/index.html`

2. To see the *Security Dictionary*, navigate in a web browser to the location of the dictionary. For example, open:

   `file:///C:/ContactManager/build/dictionary/security/index.html`

Contact search security configuration parameters in ContactManager

There are two parameters you can set in the ContactManager `config.xml` file to configure security for contact searches, `RestrictSearchesToPermittedItems` and `RestrictContactPotentialMatchToPermittedItems`. You edit this file in Guidewire Studio for ContactManager.

You can use the `RestrictSearchesToPermittedItems` configuration parameter to control the interaction between the permissions `abviewsearch` and `abview`. The `abviewsearch` permission determines which users can search for contacts. However, not all users with `abviewsearch` permission also have `abview` permission. The `abview` permission enables users to view the contact's detailed information.

If `RestrictSearchesToPermittedItems` is `false`, in response to a search the system returns all contacts that match the search criteria. If this parameter is `true`, the system returns only contacts for which the user has view permissions. This setting also interacts with contact and tag permissions. For example, if a user can view only the `Person` subtype with any tag and `RestrictSearchesToPermittedItems` is `true`, the system returns only contacts of the `Person` subtype.

Additionally, you can set the `RestrictContactPotentialMatchToPermittedItems` parameter. This parameter controls the security of potential search results. If the parameter is `true`, only the potential matches for which the user has view permissions are returned.

Configuring ContactManager contact security

Use the `SystemPermissionType` typelist and the `security-config.xml` configuration file to define new ContactManager contact security permissions. These resources enable you to create more finely grained system permissions for specific `Contact` subtypes and tags.

Create permissions for an ABContact subtype in ContactManager

Procedure

1. If necessary, start Guidewire Studio for ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb studio
   ```

2. In Guidewire Studio for ContactManager, press `Ctrl+Shift+N` and search for `SystemPermissionType.ttx`.
3. Double-click `SystemPermissionType.ttx` in the search results to open this typelist in an editor.
4. Add one or more new typecodes to `SystemPermissionType.ttx`.
   For example, for each of the following contact permission typecodes, right-click an existing typecode and choose **Add new** > **typecode**. Then enter the information for the new typecode.

Code	Name	Description
`abpersoncreate`	Create an `ABPerson` instance	Permission to create an instance of an `ABPerson` subtype
`abpersonview`	View an `ABPerson` instance	Permission to view an instance of an `ABPerson` subtype
`abpersonedit`	Edit an `ABPerson` instance	Permission to edit an instance of an `ABPerson` subtype
`abpersondelete`	Delete an `ABPerson` instance	Permission to delete an instance of an `ABPerson` subtype

5. In the **Project** window, navigate to **configuration** > **config** > **security** and then double-click `security-config.xml`.
6. Associate the new permissions with an `ABContact` subtype in the `security-config.xml` file.
   For example, map the new typecodes to the `ABPerson` contact subtype as follows:

```
<ContactPermissions>
  <ContactSubtypeAccessProfile entity="ABPerson">
    <ContactCreatePermission permission="abpersoncreate"/>
    <ContactViewPermission permission="abpersonview"/>
    <ContactEditPermission permission="abpersonedit"/>
    <ContactDeletePermission permission="abpersondelete"/>
```

```
  </ContactSubtypeAccessProfile>
</ContactPermissions>
```

This entry might be the only `ContactPermissions` entry in the file. By default, the file contains a set of `StaticHandler` entries for system permissions.

7. Stop the ContactManager server, optionally regenerate the data and security dictionaries, and then restart ContactManager, as follows:
   a) At a command prompt open in the ContactManager installation folder, enter the following command:

      ```
      gwb stopServer
      ```

   b) At a command prompt open in the ContactManager installation folder, optionally regenerate the data and security dictionaries:

      ```
      gwb genDataDictionary
      ```

      Regenerating the *Security Dictionary* is a good way to find out if there is a problem with the new definitions.
   c) Restart ContactManager:

      ```
      gwb runServer
      ```

8. Add the new permissions to a new user role.
   a) Log in to ContactManager as a user that has the User Admin role.
      For example, user name `su` with password `gw`.
   b) Click the **Administration** tab.
   c) In the sidebar, click **Users & Security** > **Roles**.
   d) In the **Roles** screen, click **Add Role**.
   e) For **Name** enter `ABPerson Manager`.
   f) Add the following set of permissions to the role.

      For each of the following permissions, click **Add** and then click in the new field. Then choose a permission from the drop-down list:

      - **Create an ABPerson instance**
      - **Create contact with any tag**
      - **Delete an ABPerson instance**
      - **Delete contact with any tag**
      - **Edit an ABPerson instance**
      - **Edit contact with any tag**
      - **View address book contact search pages**
      - **View an ABPerson instance**
      - **View contact with any tag**
   g) Click **Update** to add the new permissions to the role.
9. Assign the role to one or more users. You have a list of users if you loaded sample data, as described at “Load sample data for ContactManager” on page 18.
   a) In the sidebar, navigate to **Users & Security** > **Roles**.
   b) On the **Roles** screen, click **ABPerson Manager**.
   c) Click the **Users** tab and click **Edit**, and then click **Add**.

d) On the **Search Users** screen, enter `S` in the **Last Name** field and click **Search**.

e) If you have the sample data loaded, Steve Smith is one of the **Search Results**. Click click **Select** next to the name.

f) Click **Update** to add Steve Smith to the **ABPerson Manager** role.

With the ABPerson Manager role, Steve Smith now has permission to create, edit, delete, and view a person or any `ABPerson` subtype. In addition, Steve has permission to create, edit, delete, or view a contact with any tag. These permissions are required for Steve to be able to use the menus, screens, and widgets associated with creating, editing, or deleting a person.

10. Log in as `ssmith` with password `gw` and test to see if this user can create a new `ABPerson` or `ABPerson` subtype and edit and delete contacts of these types. Also, verify that searching for contacts shows only `ABPerson` or `ABPerson` subtypes.

Creating client tag permissions in ContactManager

The base application comes with a set of tag permissions that permit a user to create, delete, edit, and view contacts with any tag. You can add permissions that control access to contact tags at a more granular level.

Tag permissions can cause more restriction on access to contacts than you might intend. A set of permissions for a single tag apply exclusively to that tag. If you want to control access to contact tags at this more granular level, create a set of tag permissions for every contact tag in your system. Then create roles using tag permissions and assign the roles to users based on the tags you expect each subset of contacts to have.

For example, a user is in a role that has only the Client tag permissions and no other tag permissions. That user cannot see contacts that have additional tags, even if those contacts have the Client tag. That user was previously able to see a contact with the Client tag, but now can no longer see that same contact. The reason is that the contact filed a claim, and ClaimCenter automatically tagged the contact with the Claim Party tag. Since the user's role does not have the Claim Party tag permissions, that user can no longer see the Client contact. Adding the Claim Party tag permissions to the role enables the user to see and work with contacts who have both tags, Client and Claim Party, and no other tags. However, with only these two sets of tag permissions, this user would not be able to see a contact that is tagged both as a Client and a Vendor.

See also

- "Create new claim party and vendor permissions and associated role" on page 94
- "ContactManager contact subtype and tag permissions" on page 89.

Create new claim party and vendor permissions and associated role

About this task

In this example, you create a set of permissions that control access to contacts that have the Client tag. Because ClaimCenter assigns the Claim Party tag automatically for participants in a claim, this example also creates permissions for Claim Party tags. You then create a role that includes both sets of tags and assign it to a user.

Procedure

1. Start Guidewire Studio for ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **extensions** > **Typelist** and then double-click `SystemPermissionType.ttx` to open this typelist in an editor.

3. For each of the following contact tag permission typecodes, right-click an existing typecode and choose **Add new** > **typecode**. Then enter the information for the new typecode.

New Code	Name	Description
`claimpartytagcreate`	Create contact with Claim Party tag	Permission to create a contact with a Claim Party tag
`claimpartytagdelete`	Delete contact with Claim Party tag	Permission to delete a contact with a Claim Party tag
`claimpartytagedit`	Edit contact with Claim Party tag	Permission to edit a contact with a Claim Party tag
`claimpartytagview`	View contact with Claim Party tag	Permission to view a contact with a Claim Party tag
`clienttagcreate`	Create contact with Client tag	Permission to create a contact with a Client tag
`clienttagdelete`	Delete contact with Client tag	Permission to delete a contact with a Client tag
`clienttagedit`	Edit contact with Client tag	Permission to edit a contact with a Client tag
`clienttagview`	View contact with Client tag	Permission to view a contact with a Client tag

4. In the **Project** window, navigate to **configuration** > **config** > **security** and double click `security-config.xml`.
5. Associate the new permissions with the Client and Claim Party tags in the `security-config.xml` file.
   - If you previously added other contact permissions, you already have a `ContactPermissions` element. In that case, add the two `ContactTagAccessProfile` elements to the existing `ContactPermissions` element.
   - If these contact permissions are the first ones you are adding, add the following new typecodes:

```
<ContactPermissions>
  <ContactTagAccessProfile tag="ClaimParty">
    <ContactCreatePermission permission="claimpartytagcreate"/>
    <ContactDeletePermission permission="claimpartytagdelete"/>
    <ContactEditPermission permission="claimpartytagedit"/>
    <ContactViewPermission permission="claimpartytagview"/>
  </ContactTagAccessProfile>
  <ContactTagAccessProfile tag="Client">
    <ContactCreatePermission permission="clienttagcreate"/>
    <ContactDeletePermission permission="clienttagdelete"/>
    <ContactEditPermission permission="clienttagedit"/>
    <ContactViewPermission permission="clienttagview"/>
  </ContactTagAccessProfile>
</ContactPermissions>
```

6. Stop the ContactManager server, regenerate the data and security dictionaries, and then restart ContactManager, as follows:
   a) If ContactManager is running, open a command prompt in the ContactManager installation folder and then enter the following command:

   ```
   gwb stopServer
   ```

   b) To ensure that your new permissions are correctly formatted, at a command prompt, navigate to the ContactManager installation folder and then regenerate the data and security dictionaries:

   ```
   gwb genDataDictionary
   ```

   c) Restart ContactManager:

   ```
   gwb runServer
   ```

7. Add one or more new permissions to a user role. For example:
   a) Log in to ContactManager as a user that has the User Admin role.

   For example, user name `su` with password `gw`.

   b) Click the **Administration** tab.
   c) In the sidebar, click **Users and Security** > **Roles**.
   d) In the **Roles** screen, click **Add Role**.
   e) For **Name** enter `Client ContactManager`.
   f) Add the following set of permission to the role.

For each of the following permissions, click **Add** and then click in the new field. Then choose a permission from the drop-down list:

- **Create address book contacts**
- **Create contact with Claim Party tag**
- **Create contact with Client tag**
- **Delete address book contacts**
- **Delete contact with Claim Party tag**
- **Delete contact with Client tag**
- **Edit address book contacts**
- **Edit contact with Claim Party tag**
- **Edit contact with Client tag**
- **View address book contact search pages**
- **View address book contacts**
- **View contact with Claim Party tag**
- **View contact with Client tag**

g) Click **Update** to add the new permissions to the role.

8. Click **Actions** > **New User**.

9. Enter the following values for the new user:

Field	Value
First name	`Pat`
Last name	`Hu`
Username	`phu`
Password	`gw`

10. Under **Roles**, click **Add**.

11. Click the empty **Name** field and choose **Client ContactManager** from the list.

12. Click **Update** to save the new user.

13. Log in as `phu` with password `gw` and ensure that this user can edit and delete contacts that have the Client tag, the Claim Party tag, and both tags.

The sample data has contacts with all these tags set. You can also log in as another user, like `su`, and create users with various tags for testing. To load sample data, see “Load sample data for ContactManager” on page 18.

14. Search for contacts with **Tag** specified as `Vendor` and verify that no contacts are returned.

15. Create a user and verify that you can assign only Claim Party and Client tags.

PolicyCenter contact security

The primary discussion of PolicyCenter security is in "Security: Roles, Permissions, and the Community Model" in the *PolicyCenter Application Guide*.

Additionally, there is an example that uses PolicyCenter contact security in “Configure ContactManager-to-PolicyCenter authentication” on page 48.

There are *ab* and *anytag* contact permissions, such as `abedit`, `abcreate`, `anytagedit`, and `anytagcreate`, that are part of a general contact security infrastructure that supports all Guidewire applications. This infrastructure also provides permission check expressions like `perm.Contact.createab` and `perm.Contact.editab`. You can configure and extend these permissions and expressions in PolicyCenter.

The following ClaimCenter topics describe how these permissions and expressions work in ContactManager and ClaimCenter:

- “ClaimCenter contact subtype and tag permissions” on page 99
- “ClaimCenter contact and tag permission check expressions” on page 100

BillingCenter contact security

BillingCenter provides contact permissions to secure access to contact related tasks and screens.

BillingCenter contact-related permissions

As described at “Security” in the *BillingCenter Application Guide*, BillingCenter uses roles and permissions to limit tasks that users can perform.

For tasks related to contacts, such as editing data for an account contact, BillingCenter provides the following permissions:

Name	Code	Description
Create account contact	`acctcntcreate`	Permission to add a new contact to an account
Create policy contact	`plcycntcreate`	Permission to add a new contact to a policy period
Create producer contact	`prodcntcreate`	Permission to add a new contact to a producer
Delete account contact	`acctcntdelete`	Permission to remove a contact from an account
Delete policy contact	`plcycntdelete`	Permission to remove a contact from a policy period
Delete producer contact	`prodcntdelete`	Permission to remove a contact from a producer
Edit account contact	`acctcntedit`	Permission to edit information on an account contact
Edit policy contact	`plcycntedit`	Permission to edit information on an existing policy period contact
Edit producer contact	`prodcntedit`	Permission to edit information on an existing producer contact
View account contacts screen	`acctcontview`	Permission to view **Accounts** > **Contacts** screen
View policy contacts screen	`plcycontview`	Permission to view **Policies** > **Contacts** screen
View producer contacts screen	`prodcontview`	Permission to view **Producers** > **Contacts** screen

Additionally, there are *ab* and *anytag* contact permissions, such as `abedit`, `abcreate`, `anytagedit`, and `anytagcreate` that are part of a general contact security infrastructure that supports all Guidewire applications.

See also

- “ContactManager contact subtype and tag permissions” on page 89
- “Configure ContactManager-to-BillingCenter authentication” on page 51

BillingCenter contact and tag permission check expressions

BillingCenter uses permission check expressions to control access to contact-related screens and widgets. Each contact permission check expression is associated with a permission. For example, the `NewAccountContactPopup` widget has its `CanVisit` property set to the permission check expression `perm.AccountContact.create`. This setting allows only the users who have the `acctcntcreate` permission to see this popup.

The base configuration of BillingCenter uses the following contact-related permission check expressions:

BillingCenter Expression	Description
`perm.AccountContact.create`	Related permission is `acctcntcreate`.

BillingCenter Expression	Description
`perm.AccountContact.delete`	Related permission is `acctcntdelete`.
`perm.AccountContact.edit`	Related permission is `acctcntedit`.
`perm.PolicyPeriodContact.create`	Related permission is `plcycntcreate`.
`perm.PolicyPeriodContact.delete`	Related permission is `plcycntdelete`.
`perm.PolicyPeriodContact.edit`	Related permission is `plcycntedit`.
`perm.ProducerContact.create`	Related permission is `prodcntcreate`.
`perm.ProducerContact.delete`	Related permission is `prodcntdelete`.
`perm.ProducerContact.edit`	Related permission is `prodcntedit`.

The Guidewire contact security infrastructure that provides the *ab* and *anytag* contact permissions also provides permission check expressions like `perm.Contact.createab` and `perm.Contact.editab`. You can configure and extend these permissions and expressions in BillingCenter.

Build and view the BillingCenter Security Dictionary

Procedure

1. At a command prompt, run the following command from the BillingCenter installation folder:

   ```
   gwb genDataDictionary
   ```

   This command builds the *Security Dictionary* in the following location

   `BillingCenter/build/dictionary/security/index.html`

2. Navigate in a web browser to the location of the *Security Dictionary*. For example:

   `file:///C:/BillingCenter/build/dictionary/security/index.html`

Viewing permissions in the BillingCenter Security Dictionary

You can use the *Security Dictionary* to get information on the application permission keys.

For example, open the *Security Dictionary* and click the **System Permissions** filter at the top of the left pane. You see all the system permissions listed on the left by code name. If you click the permission code **acctcntcreate**, you see that it has the related application permission key **AccountContact create**, which has the Gosu check expression `perm.AccountContact.create`. You can filter the list by application permission keys, pages, system permissions, and roles.

Using the *Security Dictionary*, you can determine the following:

- The system permission related to an application permission key
- The PCF files and widgets that use an application permission key
- The roles, application permission keys, PCF pages, and widgets that use a system permission
- A list of the Gosu application permission expressions called from each PCF page
- A list of the permissions assigned to each role

See also

- “Build and view the BillingCenter Security Dictionary” on page 98
- “BillingCenter contact and tag permission check expressions” on page 97

ClaimCenter contact security

To fully understand ClaimCenter contact security, you need also to understand ClaimCenter permissions and configuration values. See the following references:

- “Security: Roles, Permissions, and Access Controls” in the *ClaimCenter Application Guide*
- “Securing Access to ClaimCenter Objects” in the *ClaimCenter Administration Guide*

ClaimCenter contact roles

As described in “Security: Roles, Permissions, and Access Controls” in the *ClaimCenter Application Guide*, a role is a collection of permissions.

By grouping permissions into roles, you can define a user’s authority with a few assigned roles, rather than with a much larger list of permissions. A user can have any number of roles, but has to have at least one.

You might need more granular control over who gets to view, edit, create, and delete contacts, rather than using the simple view and edit permissions. For example, you have ContactManager users that manage certain subtypes of contacts, and you want the system to enforce permissions at the contact subtype level. This role is especially important for the Service Provider Management feature, of which the list of contact subtypes—service providers—is an integral part. Only specific ContactManager users can manage the lists of these contact subtypes.

The permissions used in the base configuration of ClaimCenter for ContactManager are `abview`, `abviewsearch`, `abedit`, `abeditpref`, `abcreate`, `abcreatepref`, `abdelete`, `abdeletepref`, `anytagcreate`, `anytagdelete`, `anytagedit`, and `anytagview`.

With administrator privileges, you can assign permissions to particular users for certain contacts or contact subtypes. For example, you can grant one user the ability to manage the Auto Body Shops contact subtype and another the permission to manage other contact subtypes.

See also

- ClaimCenter contact permissions are described in more detail in “ClaimCenter contact subtype and tag permissions” on page 99

ClaimCenter contact subtype and tag permissions

ClaimCenter provides subtype and tag permissions that you can use to control access to contacts. These permissions make a distinction between local contacts and centralized, or address book, contacts. The `SystemPermissionType` typelist lists all the subtype and tag permissions in your system.

The following table lists the base subtype and tag permissions provided with ClaimCenter contacts:

Code	Description of Permission
`anytagview`	See a contact that has any contact tag.
`anytagcreate`	Create a new contact regardless of which contact tag it requires.
`anytagdelete`	Delete a local, unlinked contact that has any contact tag.
`anytagedit`	Edit a contact that has any contact tag.
`ctcview`	View and search local contacts.
`ctccreate`	Create a new, local contact.
`ctcedit`	Edit local contacts.
`abview`	View the details of contact entries retrieved from ContactManager.
`abviewsearch`	Search for contact entries in ContactManager.
`abcreate`	Create a new vendor contact in ContactManager. In the base configuration, this permission enables a ClaimCenter user to create a vendor contact and have it saved in ContactManager. Without this permission, a ClaimCenter user can create and save

Code	Description of Permission
	non-vendor contacts in ContactManager. Any vendor contacts created by a user without this permission are created in ContactManager with pending status and must be approved by a ContactManager user.
`abcreatepref`	Create a new preferred vendor in ContactManager.
`abedit`	Edit an existing vendor contact stored in ContactManager. In the base configuration, this permission enables a ClaimCenter user to edit a vendor contact and have it saved in ContactManager. Without this permission, a ClaimCenter user can edit and save non-vendor contacts in ContactManager. Any vendor contact changes by a user without this permission become pending changes in ContactManager and must be approved by a ContactManager user.
`abeditpref`	Edit an existing preferred vendor stored in ContactManager.

The system uses role-based security for these permissions. As described in the previous topic, to implement role-based security, a system administrator associates permissions with roles and assigns roles to users. For each role assigned, the user acquires the permissions associated with that role. For example, a role associated with the `abcreate` and `anytagcreate` permissions enables the user who has this role to create any type of contact.

The base contact and tag permissions apply across all contact subtypes. If you grant a permission to a contact type, you grant the same permissions to all that contact's subtypes.

ClaimCenter enables you to restrict permissions according to contact subtype or tag. For example, you can enable a user with `abcreate` permission to create only `PersonVendor` contacts, but not `CompanyVendor` contacts. You configure contact and tag permissions through the `SystemPermissionType` typelist and the `security-config.xml` resource.

Note: If you create a set of tag permissions for a specific tag, these permissions enable access only to contacts that have that one tag. For example, you create a set of Vendor tag permissions and a user has a role with only those tag permissions. That user will not be able to work with a contact that has both Claim Party and Vendor tags. You could also create a set of tag permissions for Claim Party tags. In that case, a user with both Vendor and Claim Party tag permissions would be able to work with contacts that have both Vendor and Claim Party tags.

See also

- For examples showing how to add typecodes to the `SystemPermissionType` typelist, set up `security-config.xml`, and implement role-based security, see "Configuring ClaimCenter contact security" on page 102.
- For examples of combinations of permissions that permit various kinds of work with contacts in ClaimCenter, see the *Application Guide*.

ClaimCenter contact and tag permission check expressions

In a page configuration file (PCF), you can control permissions on specific widgets with Gosu expressions that determine if a user has permission to perform an operation.

For example, the setting for the ClaimCenter **AddressBookContactDetail** page's `canVisit` attribute is `perm.Contact.viewab(externalContact.Contact)`. This setting limits users of this page to viewing only contact types for which they have subtype permissions and tag permissions to view the contact. Additionally, the `canEdit` attribute setting is `externalContact.Source.supportsUpdate()` and `perm.Contact.editab(externalContact.Contact)`. This setting limits a user of this page to editing only contacts that support updates and for which the user has subtype permissions and tag permissions to edit the contact.

Note: To see this file, in Guidewire Studio for ClaimCenter, press `Ctrl+Shift+N` and enter `AddressBookContactDetail`, and then click `AddressBookContactDetail.pcf (configuration\config\web\pcf\addressbook)` in the list of objects that the system finds.

ClaimCenter does not use tag permission checks to control access to screens or buttons. However, when a user tries to save a new contact to ContactManager, ClaimCenter applies the `createab` permission check expression. This permission check ensures that the user has permission to create the tag as well as the contact. If the user does not have this permission, ContactManager creates a pending contact that requires approval in ContactManager to become permanent.

Some Gosu permission check expressions require an input parameter, and some do not. The following table lists the ClaimCenter contact and tag permission check expressions:

ClaimCenter Expression	Description
`perm.Contact.createab`	Takes an input parameter that is either a `Contact` type or subtype or a `ContactTagType` typecode specified as an enumeration constant. Depending on the parameter, verifies that the user has the permission to create either a contact with the specified tag or a contact of the specified type.
`perm.Contact.createpreferredab`	Does not take an input parameter. Verifies that the user has permission to create the preferred contact.
`perm.Contact.deleteab`	Takes a `Contact` instance as an input parameter. Determines the subtype and contact tag or tags from the instance, and then verifies that the user has the contact and tag permissions to delete the contact.
`perm.Contact.deletepreferredab`	Does not take an input parameter. Verifies that the user has permission to delete the preferred contact.
`perm.Contact.editab`	Takes a `Contact` instance as an input parameter. Determines the subtype and contact tag or tags from the instance, and then verifies that the user has the contact and tag permissions to edit the contact.
`perm.Contact.editpreferredab`	Does not take an input parameter. Verifies that the user has permission to edit the preferred contact.
`perm.Contact.viewab`	Takes a `Contact` instance as an input parameter. Determines the subtype and contact tag or tags from the instance, and then verifies that the user has the contact and tag permissions to view the contact.
`perm.Contact.viewsearchab`	Does not take an input parameter. Verifies that the user has permission to edit the preferred contact in the address book.
`perm.Contact.createlocal`	Does not take an input parameter. Verifies that the user has permission to create the local contact.
`perm.Contact.editlocal`	Requires a `Contact` instance as an input parameter. Verifies that the user has permission to edit either an existing local contact or a user contact. To edit a user contact, the user needs edit user permission.
`perm.Contact.viewlocal`	Does not take an input parameter. Verifies that the user has permission to view and search local contact entries.

Viewing permissions in the ClaimCenter Security Dictionary

You can use the *Security Dictionary* to get information on the application permission keys. For example, if you click the **System Permissions** filter at the top of the left pane, you see all the permissions listed on the left by code name. If you click the permission code **abedit**, you see that it has the related application permission key **Contact editab**, which has the associated Gosu check expression `perm.Contact.editab`. You can filter the list by application permission keys, pages, system permissions, and roles.

By using the *Security Dictionary*, you can determine the following:

- The system permission related to an application permission key
- The PCF files and widgets that use an application permission key
- The roles, application permission keys, PCF pages, and widgets that use a system permission
- A list of the Gosu application permission expressions called from each PCF page
- A list of the permissions assigned to each role

See also

- “Build and view the ClaimCenter Security Dictionary” on page 102

Build and view the ClaimCenter Security Dictionary

Procedure

1. At a command prompt, run the following command from the ClaimCenter installation folder:

   ```
   gwb genDataDictionary
   ```

   This command builds the *Security Dictionary* in the following location

   `ClaimCenter/build/dictionary/security/index.html`

2. Navigate in a web browser to the location of the *Security Dictionary*. For example:

   `file:///C:/ClaimCenter/build/dictionary/security/index.html`

Contact permission search configuration parameters in ClaimCenter

There are two parameters you can set in the ClaimCenter `config.xml` file to control how searching for externally stored contacts interacts with contact permission. Those parameters are `RestrictSearchesToPermittedItems` and `RestrictContactPotentialMatchToPermittedItems`. You edit `config.xml` in Guidewire Studio for ContactManager.

You can use the `RestrictSearchesToPermittedItems` configuration parameter to control the interaction between the permissions `abviewsearch` and `abview`. The `abviewsearch` permission determines which users can use the Address Book search function. However, not all users with `abviewsearch` permission also have `abview` permission. The `abview` permission enables users to view the contact’s detailed information.

If `RestrictSearchesToPermittedItems` is `false`, in response to a search, the system returns all contacts that match the search criteria. If this parameter is `true`, the system returns only contacts for which the user has view permissions. This setting also interacts with contact and tag permissions. For example, if a user can view the `Person` subtype with any tag and `RestrictSearchesToPermittedItems` is `true`, the system returns only contacts of the `Person` subtype.

Additionally, you can set the `RestrictContactPotentialMatchToPermittedItems` parameter. This parameter controls the security of potential search results. If the parameter is `true`, only the potential matches for which the user has view permissions are returned.

Configuring ClaimCenter contact security

Use the `SystemPermissionType` typelist and the `security-config.xml` configuration file to define new ClaimCenter contact security permissions. These two files enable you to create more finely grained system permissions for contact subtypes and tags.

Create contact permissions for a subtype in ClaimCenter

Procedure

1. Start Guidewire Studio for ClaimCenter.

   At a command prompt, navigate to the ClaimCenter installation folder and enter the following command:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Typelist**.
3. Double-click `SystemPermissionType.ttx` to open this typelist in the editor.
4. For each of the following contact permission typecodes, right-click an existing typecode and choose **Add new** > **typecode**. Then enter the information for the new typecode.

New Code	Name	Description
companyvendorcreate	Create company vendor contacts	Permission to create company vendor contacts
companyvendoredit	Edit company vendor contacts	Permission to edit company vendor contacts
companyvendordelete	Delete company vendor contacts	Permission to delete company vendor contacts
companyvendorview	View company vendor contacts	Permission to view company vendor contacts
personvendorcreate	Create person vendor contacts	Permission to create person vendor contacts
personvendoredit	Edit person vendor contacts	Permission to edit person vendor contacts
personvendordelete	Delete person vendor contacts	Permission to delete person vendor contacts
personvendorview	View person vendor contacts	Permission to view person vendor contacts

5. In the **Project** window, navigate to **configuration** > **config** > **security** and double-click `security-config.xml`.
6. Associate the new permissions with a `Contact` subtype in the `security-config.xml` file.
   For example, add the new typecodes for the `Vendor` and `VendorPerson` contact subtype permissions to the `ContactPermissions` element as follows:

```
<ContactPermissions>
...
  <ContactSubtypeAccessProfile entity="CompanyVendor">
    <ContactCreatePermission permission="companyvendorcreate"/>
    <ContactEditPermission permission="companyvendoredit"/>
    <ContactDeletePermission permission="companyvendordelete"/>
    <ContactViewPermission permission="companyvendorview"/>
  </ContactSubtypeAccessProfile>
  <ContactSubtypeAccessProfile entity="PersonVendor">
    <ContactCreatePermission permission="personvendorcreate"/>
    <ContactEditPermission permission="personvendoredit"/>
    <ContactDeletePermission permission="personvendordelete"/>
    <ContactViewPermission permission="personvendorview"/>
  </ContactSubtypeAccessProfile>
...
</ContactPermissions>
```

7. Stop the ClaimCenter server, regenerate the data and security dictionaries, and then restart ClaimCenter, as follows:
   a) If ClaimCenter is running, open a command prompt in the ClaimCenter installation folder and then enter the following command:

   ```
   gwb stopServer
   ```

   b) To verify that the new permissions are correct, at a command prompt, navigate to the ClaimCenter installation folder and regenerate the data and security dictionaries:

   ```
   gwb genDataDictionary
   ```

   c) After you get a successful build of the dictionaries, restart ClaimCenter:

   ```
   gwb runServer
   ```

8. Add the new permissions to a user role. For example:
   a) Log in to ClaimCenter as a user that has the User Admin role, such as user name `su` with password `gw`.
   b) Click the **Administration** tab.
   c) Click **Users & Security** > **Roles** in the sidebar.
   d) Click **Add Role**.
   The New Role screen opens.

e) For **Name**, enter `Vendor Admin`.

f) For **Description**, enter `Manages vendor contacts. Requires Clerical role as well.`

g) Click **Add** to add a new permission.

h) Click the new field and choose the **Create company vendor contacts** permission from the drop-down list.

i) Repeat "step g" and "step h" for each of the following permissions, substituting the actual permission for **Create company vendor contacts** in "step h":

- **Create person vendor contacts**
- **Delete company vendor contacts**
- **Delete person vendor contacts**
- **Edit company vendor contacts**
- **Edit person vendor contacts**
- **View company vendor contacts**
- **View person vendor contacts**

j) Click **Update** to add the new role.

9. Remove the `abview` permission from the Clerical role.

   Note: Removing this permission is a quick way to test the new Vendor Admin role. With `abview` removed, using the Clerical role in conjunction with Vendor Admin makes it possible to limit users with the two roles from viewing non-vendor contacts. However, after this change, users who have the Clerical role and no other will be unable to view any contacts. For this change to be permanent, another role with `abview` permission would need to be created and then used in conjunction with the Clerical role for all previous Clerical users.

a) Click the **Administration** tab.

b) Click **Users & Security** > **Roles** in the sidebar.

c) Click **Clerical** and then click **Edit**.

d) Click the **Code** column heading to list `abview` at the top.

e) Click the check box for `abview`, and then click **Remove**.

f) Click **Update** to save your changes.

10. Create a new user for the role.

a) Click **Actions** > **New User**.
The **New User** screen opens.

b) For **First name**, enter `Devra`.

c) For **Last name**, enter `Rajan`.

d) For **User name**, enter `drajan`.

e) For **Password** and **Confirm Password**, enter `gw`.

f) Under **Roles**, click **Add**, and then click the **Name** field. Choose **Vendor Admin** from the drop-down list.

g) Under **Roles**, click **Add** again, and then click the **Name** field. Choose **Clerical** from the drop-down list.

h) Under **Groups**, click **Add** and then click the **Group** field. Choose a group from the list, such as the sample data group **Western Regional Claims Center**.

i) Click **Update** to create the new user.

11. Log out.

   For example, if you logged in as `su`, on the Options menu ⚙, click **Log Out Super User**.

12. Log in as `drajan` with password `gw`.
13. Verify that this user can create a new `CompanyVendor` or `PersonVendor` subtype and have it saved in ContactManager without being pending.
14. Verify that this user can edit and delete contacts of these types and have the changes saved in ContactManager without the contacts being made pending.
15. Verify that in an Address Book search, this user can click in the search results and see details only for `CompanyVendor` and `PersonVendor` contact subtypes and not for non-vendor subtypes. For example, clicking a contact of type `Company` or `Person` displays a message saying that you do not have permission to view the contact detail popup.

Create claim party and vendor tag permissions in ClaimCenter

About this task

The base application comes with a set of tag permissions that permit a user to create, delete, edit, and view all tags. You can add permissions that control access to tags at a more granular level. This topic shows how to create a set of tag permissions in ClaimCenter.

Note: In this example, you create a set of Vendor tag permissions. These permissions enable a user to see and work with contacts that have only the Vendor tag. If the contact has any other tags, these permission do not enable working with that contact. For example, a user who has a single role with only Vendor tag permissions is not able to work with a contact that has both Claim Party and Vendor tags. You could create a set of tag permissions for Claim Party tags and add them to the role that has the Vendor tag permissions. A user with that role would be able to work with contacts that have both Vendor and Claim Party tags. However, that user would not be able to work with contacts that have both Vendor and Client tags. For more information on the application tags in the base configuration, see "ClaimCenter contact subtype and tag permissions" on page 99.

Procedure

1. Start Guidewire Studio for ClaimCenter.

   At a command prompt, navigate to the ClaimCenter installation folder and enter the following command:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Typelist**.
3. Double-click `SystemPermissionType.etx` to open this typelist in an editor.
4. For each of the following contact permission typecodes, right-click an existing typecode and choose **Add new** > **typecode**. Then enter the information for the new typecode.

New Code	Name	Description
`claimpartytagcreate`	Create contact with Claim Party tag	Permission to create a contact with a Claim Party tag
`claimpartytagdelete`	Delete contact with Claim Party tag	Permission to delete a contact with a Claim Party tag
`claimpartytagedit`	Edit contact with Claim Party tag	Permission to edit a contact with a Claim Party tag
`claimpartytagview`	View contact with Claim Party tag	Permission to view a contact with a Claim Party tag
`vendortagcreate`	Create contact with Vendor tag	Permission to create a contact with a Vendor tag
`vendortagdelete`	Delete contact with Vendor tag	Permission to delete a contact with a Vendor tag
`vendortagedit`	Edit contact with Vendor tag	Permission to edit a contact with a Vendor tag
`vendortagview`	View contact with Vendor tag	Permission to view a contact with a Vendor tag

5. In the **Project** window, navigate to **configuration** > **config** > **security** and double-click `security-config.xml`.
6. Associate the new permissions with a contact tag in the `security-config.xml` file.

For example, add the new typecodes for the `Vendor` tag permissions to the `ContactPermissions` element as follows:

```
<ContactPermissions>
...
  <ContactTagAccessProfile tag="Vendor">
    <ContactCreatePermission permission="vendortagcreate"/>
    <ContactDeletePermission permission="vendortagdelete"/>
    <ContactEditPermission permission="vendortagedit"/>
    <ContactViewPermission permission="vendortagview"/>
  </ContactTagAccessProfile>
...
</ContactPermissions>
```

7. Stop the ClaimCenter server, regenerate the data and security dictionaries, and then restart ClaimCenter, as follows:

   a) If ClaimCenter is running, open a command prompt in the ClaimCenter installation folder and then enter the following command:

   ```
   gwb stopServer
   ```

   b) To verify that the new permission are correctly formatted, at a command prompt open in the ClaimCenter installation folder, regenerate the data and security dictionaries:

   ```
   gwb genDataDictionary
   ```

   c) Restart ClaimCenter:

   ```
   gwb runServer
   ```

8. Add the new permissions to a user role. For example:

   a) Log in to ClaimCenter as a user that has the User Admin role, such as user name `su` with password `gw`.

   b) Click the **Administration** tab.

   c) Click **Users & Security** > **Roles** in the Sidebar.

   d) Click **Add Role**.
   The New Role screen opens.

   e) For **Name**, enter `Vendor & Claim Party Tag Admin`.

   f) For **Description**, enter `Manages contacts with Vendor and Claim Party tags. Requires Clerical role as well.`

   g) Click **Add** to add a new permission.

   h) Click the new field and choose the **Create contact with Vendor tag** permission from the drop-down list.

   i) Repeat "step g" and "step h" for each of the following permissions, substituting the actual permission for **Create contact with Vendor tag** in "step h":

      - **Create address book contact**
      - **Create contact with Claim Party tag**
      - **Delete address book contact**
      - **Delete contact with Claim Party tag**
      - **Delete contact with Vendor tag**
      - **Edit address book contact**
      - **Edit contact with Claim Party tag**
      - **Edit contact with Vendor tag**
      - **View address book contact**

        - **View contact with Claim Party tag**
        - **View contact with Vendor tag**

    j) Click **Update** to add the new role.

9. Remove the `anytagview` permission from the Clerical role.

   Note: Removing this permission is a quick way to test the new Vendor & Claim Party Tag Admin role. However, after this change, users who have only the Clerical role and no other will be unable to view contacts that have tags. For contacts stored in ContactManager, all contacts must have tags, so they would not be visible. For this change to be permanent, another role with `anytagview` permission would need to be created and then used in conjunction with the Clerical role for all previous Clerical users.

   a) Click the **Administration** tab.

   b) Click **Users & Security** > **Roles** in the Sidebar.

   c) Click **Clerical** and then click **Edit**.

   d) Click the **Code** column heading to list `anytagview` on the first page.

   e) Click the check box for `anytagview`, and then click **Remove**.

   f) Click **Update** to save your changes.

10. Create a new user for the role.

    a) Click **Actions** > **New User**.
    The **New User** screen opens.

    b) For **First name**, enter `Mira`.

    c) For **Last name**, enter `Nolo`.

    d) For **User name**, enter `mnolo`.

    e) For **Password** and **Confirm Password**, enter `gw`.

    f) Under **Roles**, click **Add**, and then click the **Name** field. Choose **Vendor & Claim Party Tag Admin** from the drop-down list.

    g) Under **Roles**, click **Add** again, and then click the **Name** field. Choose **Clerical** from the drop-down list.

    h) Under **Groups**, click **Add** and then click the **Group** field. Choose a group from the list, such as the sample data group **Eastern Regional Claims Center**.

    i) Click **Update** to create the new user.

11. Log out.

    For example, if you logged in as `su`, on the Options menu ⚙, click **Log Out Super User**.

12. Log in as `mnolo` with password `gw`.

13. Verify that this user can find contacts with Vendor and Claim Party tags set, or with only one tag or the other set. If necessary, either create those contacts in ContactManager or import ContactManager sample data.

14. Verify that this user can create, edit, and delete contacts with these tags and have the changes saved in ContactManager without the contacts being made pending.

chapter 6

Extending the contact data model

In the base configuration, ContactManager provides a set of `ABContact` entities. The core applications provide a corresponding set of `Contact` entities. You can extend these entities by adding subtypes, and you can make changes to the user interface to support the new subtypes.

Overview of contact entities

Before extending the contact data model, you need to know what is provided in the ContactManager and core application base configurations. Additionally, there are guidelines you must follow when planning to extend these data models.

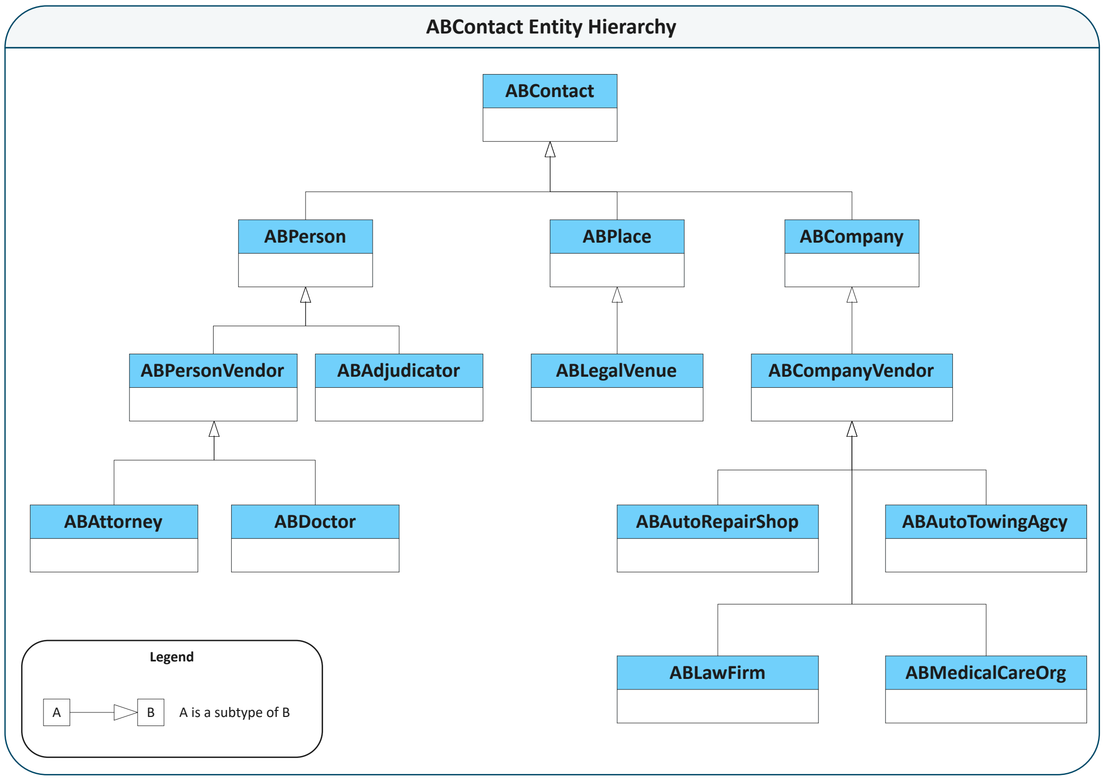
ABContact data model

The `ABContact` entity is the ContactManager data model entity used in both client and vendor contact management. `ABContact` has an entity hierarchy and a set of relationships to other entities.

ABContact entity hierarchy

The `ABContact` entity has three primary subtypes for various types of contacts, `ABPerson`, `ABCompany`, and `ABPlace`. These subtypes have additional subtypes, like `ABAdjudicator`, `ABCompanyVendor`, `ABLegalVenue`, and so on.

The following figure shows the `ABContact` entity hierarchy. This entity hierarchy has a parallel in the `Contact` entity hierarchy in the core applications. See “ContactManager and core application contact entity hierarchies” on page 113.

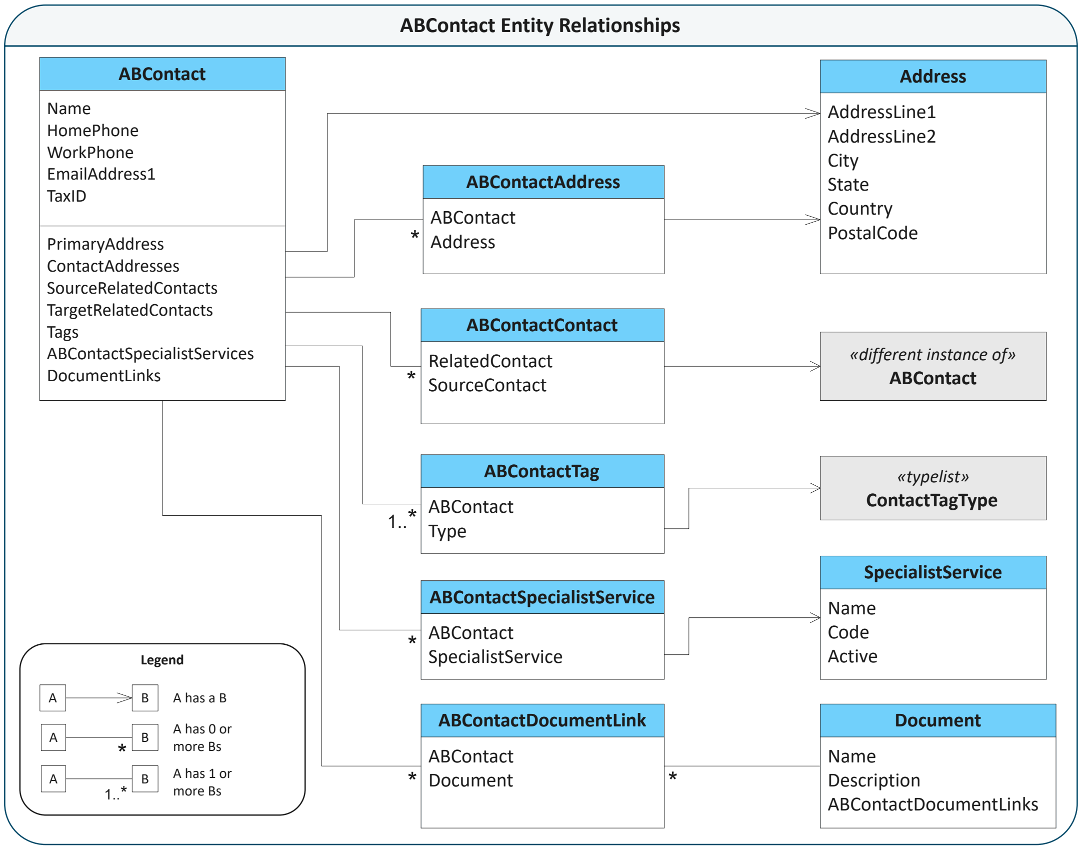


Note: This figure omits the subtypes `ABUserContact`, `ABPolicyCompany`, and `ABPolicyPerson`. Although they exist in the base configuration, these entities are not intended to be used in ContactManager. If you need to store address and phone information for a ContactManager user, use the `UserContact` entity as described in "Contact data model" on page 111.

ABContact entity relationships

The `ABContact` entity has fields for name, phone number, email address, and so forth. For many vendor contacts it is necessary to maintain a tax ID for tax reporting. The `ABContact` entity tracks this data as well. Some of this data is stored in other entities.

In the base configuration, a contact must have at least one tag. A contact can have multiple addresses, related contacts, tags, and services. References to those entities are handled with arrays of join entities, such as `ContactAddress` for addresses and `ContactContact` for related contacts, and derived properties, as shown in the following figure:



The physical location, the address, of a contact is not maintained in the ABContact entity. Instead, the ABContact entity references another entity, the Address entity, which maintains the contact's street or mailing address. An ABContact entity can reference a primary address and, through the ABContactAddress entity, other secondary addresses.

Contacts can have relationships with other contacts. For example, an ABPerson contact might be employed by a particular ABCompany contact. The ABContactContact entity maintains data about the relationships a contact can have with other contacts.

Contacts can have tags, like Client and Vendor, and specialist services that the contact provides, like carpentry or independent appraisal. An ABContact entity references its tags by using an array of ABContactTag entities. An ABContact entity references its services by using an array of ABContactSpecialistService entities.

Contact data model

A Contact is the Guidewire core application data model entity used in both client and vendor contact management. In the base configuration, this entity is the core application equivalent of the ContactManager entity ABContact, described previously in "ABContact data model" on page 109.

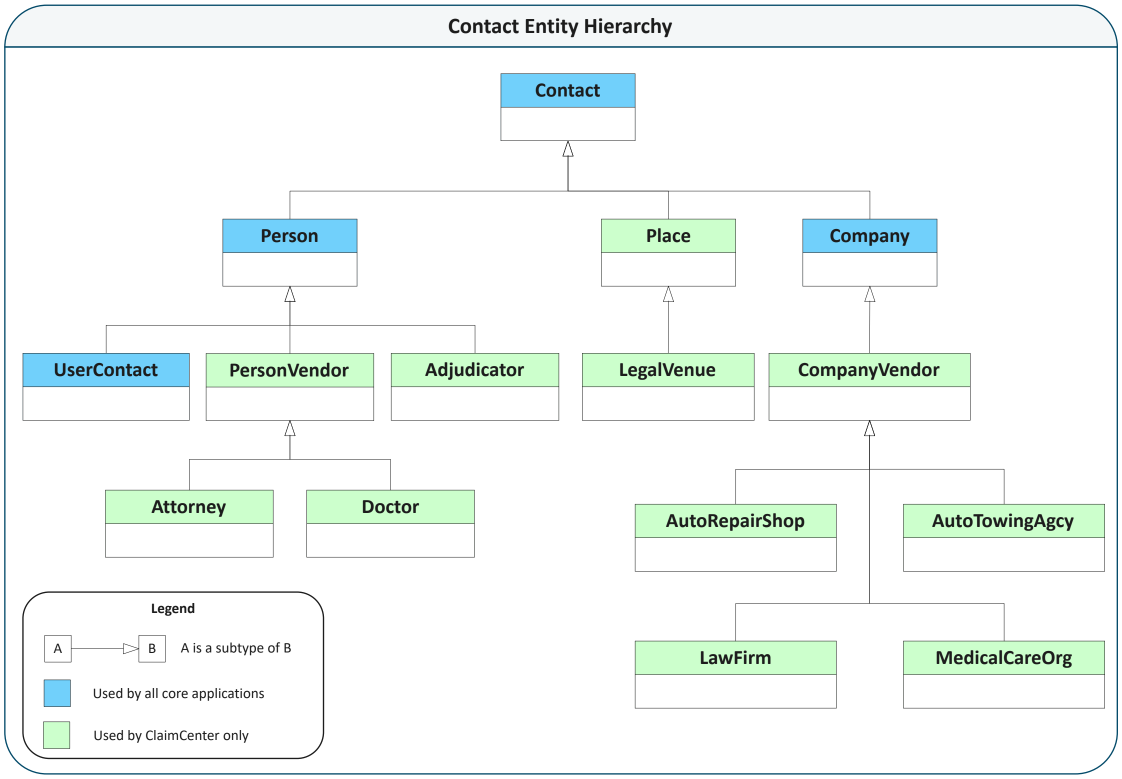
The Contact entity has an entity hierarchy and a set of relationships to other entities.

Contact entity hierarchy

The Contact entity has subtypes for various types of contacts, like Person, Company, and Place. In their base configurations, PolicyCenter and BillingCenter use just the Person and Company subtypes to link with ContactManager.

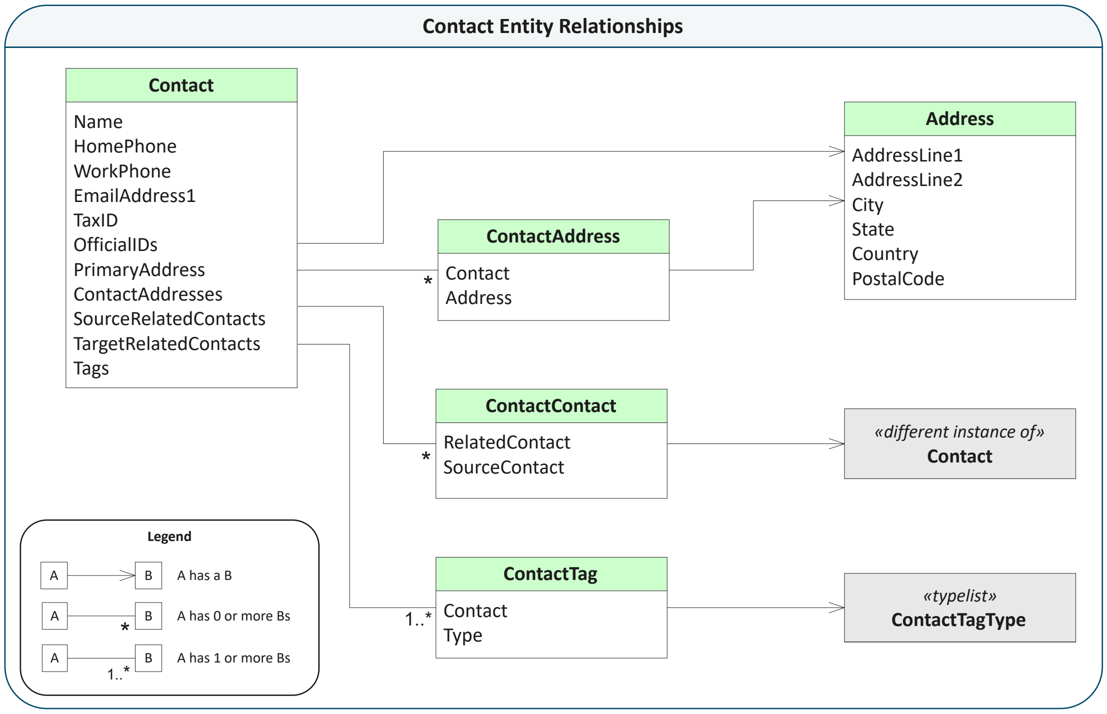
In ClaimCenter, these subtypes have additional subtypes, like Adjudicator, CompanyVendor, LegalVenue, and so on. The following figure shows the Contact entity hierarchy. This entity hierarchy has a parallel in the ABContact entity hierarchy in ContactManager. See "ContactManager and core application contact entity hierarchies" on page 113.

Note: The following diagram includes a special `Contact` subtype, `UserContact`. This entity is used by the `User` entity, which represents a user of the application. The `User` entity has a foreign key to `UserContact` to store data like the user's address and phone number, which the application can display in its user interface. Additionally, the `UserContact` entity makes it possible to do things like proximity search to assign users to claims. `UserContact` entities are not intended to be used as either vendor contacts or client contacts, and they do not link to ContactManager.



Contact entity relationships

The `Contact` entity is associated with other entities. A contact can have multiple addresses, related contacts, and tags. A `Contact` entity must have at least one tag. Except for the primary address, references to those entities are handled with arrays of join entities. For example, there is `ContactAddress` for addresses and `ContactContact` for related contacts, as shown in the following figure:



The physical location, the address, of a `Contact` is stored in the `Address` entity, which maintains the contact's street or mailing address. A `Contact` can reference a primary address and, through the `ContactAddress` entity, other secondary addresses.

Contacts can have relationships with other contacts. For example, a `Person` might be employed by a particular `Company`. The `ContactContact` entity maintains data about the relationships a contact can have with other contacts.

> **Note:** For simplicity, the diagram shows `ContactContact` connecting to a different instance of `Contact`. However, `ContactContact` can also point back to the original contact. For example, you can be your own primary contact.

Contacts can have tags, like Client and Vendor, and specialist services that the contact provides, like carpentry or independent appraisal. A `Contact` entity references its tags by using the `ContactTag` entity. The relationship between a contact and its services are maintained by ContactManager, which is why there is no property accessing services in the `Contact` entity relationship model.

See also

- "Contact tags" on page 159
- "Vendor services" on page 163

ContactManager and core application contact entity hierarchies

The Guidewire core application data models and the ContactManager data model both group and classify contact data as hierarchies. By default, ContactManager provides a contact entity hierarchy that supports both the Client Data Management add-on module and ClaimCenter vendor data management. Additionally, ContactManager entities have the prefix `AB` (for *Address Book*).

PolicyCenter and BillingCenter use only part of the hierarchy for client data management: `Contact`, `Person`, `Company`, and `UserContact`.

ClaimCenter uses the entire hierarchy. In the default application, all the ClaimCenter contact types but `UserContact` appear in ContactManager with `AB` prefixes. The following table shows the ClaimCenter hierarchy and the supporting hierarchy in ContactManager supplied in the base products:

ClaimCenter	ContactManager
`Contact`	`ABContact`
`Person`	`ABPerson`
`Adjudicator`	`ABAdjudicator`
`PersonVendor`	`ABPersonVendor`
`Attorney`	`ABAttorney`
`Doctor`	`ABDoctor`
`UserContact`	
`Company`	`ABCompany`
`CompanyVendor`	`ABCompanyVendor`
`AutoRepairShop`	`ABAutoRepairShop`
`AutoTowingAgcy`	`ABAutoTowingAgcy`
`LawFirm`	`ABLawFirm`
`MedicalCareOrg`	`ABMedicalCareOrg`
`Place`	`ABPlace`
`LegalVenue`	`ABLegalVenue`

Each core application `Contact` entity and subentity has a corresponding `ABContact` entity or subentity in ContactManager, with the exception of `UserContact`.

> **Note:** While there is an `ABUserContact` entity in the `ABContact` hierarchy, this entity is not used, even for ContactManager users. `UserContact` is used both in core applications and ContactManager solely to provide address and other contact information for a `User`. A core application `UserContact` does not link to ContactManager. `UserContact` is never used for vendor or client information.

When ContactManager creates a contact, an instance of an `ABContact` entity or subentity, ContactManager also creates a unique identifier for that instance. This unique identifier is used both by core applications and by ContactManager to ensure that each application is referencing the same contact. In the core applications, `AddressBookUID` is the field used to uniquely identify a contact that is stored in ContactManager. In ContactManager, the corresponding field is `LinkID`.

For a contact stored both in ContactManager and in a core application, ContactManager sends the unique identifier in the `LinkID` field to each core application that uses the contact. The core application stores this unique identifier in the `AddressBookUID` field of the contact. If a contact in a core application has a non-null value in its `AddressBookUID` field, that value means that the contact is stored in ContactManager. Otherwise, the `AddressBookUID` field is `null`, meaning that the contact is stored only locally in the core application, and not in ContactManager.

To enable all applications to reference the same contact, you must not change an `AddressBookUID` or a `LinkID` field.

Guidewire designed the data models to enable the Guidewire core applications and ContactManager to be compatible. If you make a contact-related extension to a Guidewire core application's data model, you typically extend the ContactManager data model as well to reflect the change. You must extend both applications if you want contact information that is captured, stored, and used in one application to be available to the other. If you have more than one core application installed, you might have to make the same extension in your other core applications as well.

Mapping of entities between applications ensures that contact records are stored with the correct entity in both the Guidewire core application and ContactManager. Mapping is not the same as linking contacts or *synchronizing* them, which determines if the records are the same between the two applications. Linking and synchronizing are separate operations from mapping.

See also

- “Linking and synchronizing contacts” on page 193
- “ContactManager link IDs and comparison to other IDs” on page 299

General guidelines for extending contacts

With some restrictions, you can customize the contact entity hierarchy by adding subtypes or custom fields. You cannot make any of the following hierarchy modifications:

- You cannot delete or move the root entity `Contact`.
- You cannot delete or move the three subtypes `Person`, `Company`, and `Place`.
- You cannot create a contact entity that is a peer to `Contact`.
- You cannot create a contact entity that is a parent to `Person`, `Company`, or `Place`.

You can add a field to any of these entities. If you add a field to an entity that is higher in the hierarchy, all entities below it inherit the field. Before adding a field, ensure that the field makes sense for all the entity’s subtypes.

You can create a peer to `Person`, `Company`, or `Place`, and you can modify these three entities. A peer entity to one of these three entities is a subtype of `Contact`. You can also create a new subtype and modify the other subtypes.

If you have integrated your Guidewire core application with ContactManager, you typically extend both the core application and the ContactManager hierarchies to mirror each other. Most contacts require central management. If you have a contact subtype that does not require central management, you can create that type just in the Guidewire core application and not in ContactManager.

The Guidewire core applications provide the Gosu class `ContactMapper` that you use when extending the `Contact` data model for coordination with ContactManager. The corresponding class you use in ContactManager is `gw.webservice.ab.ab1000.abcontactapi.ContactMapper`. Each Guidewire core application also has a pair of XML configuration files for mapping contact names and typelists to and from ContactManager.

The matching and searching functions for contacts require each subtype to have a collection of fields that makes it unique.

See also

- “Working with contact mapping files” on page 116
- “Linking and synchronizing contacts” on page 193

Deciding whether to create a subtype

Plan carefully before manipulating your contact hierarchy. A new subtype inherits the attributes and matching behavior of its supertype. Create a new subtype only if you need to treat one set of contacts differently from another. As much as possible, limit the number of subtypes you need to represent your contacts.

If you have ClaimCenter installed, consider using vendor services rather than creating new vendor contact subtypes.

Another alternative is to create your own tags and apply them to contacts.

The main reason to limit creation of new subtypes is that the more you specialize the subtype hierarchy, the more restrictive and the less flexible your model becomes. For example, there are subtypes for `AutoRepairShop` and `AutoTowingAgcy`. If you work with an auto repair shop that also does towing, you cannot create a single contact that does both. However, you can add a service for Towing to an auto repair shop contact.

If a contact has different roles or skill sets, you can use one subtype and add contact tags or a specialty array to represent those skills and specialties.

Additionally, each time you create a new subtype, you must modify PCF files to support both creating the subtype and searching for it. You might also need to make supporting modifications to the screens that reference contacts.

After making any data model modification, you must refresh the application and possibly the web services as well.

See also

- "Vendor services" on page 163
- "Contact tags" on page 159
- "Refresh applications after contact data model changes" on page 116
- For examples of modifications to contact entities and the class hierarchy, see the following topics:
  - "Adding a field to a contact subtype" on page 118
  - "Adding a vendor contact subtype" on page 125
  - "Extending contacts with an array" on page 138

Refresh applications after contact data model changes

About this task

After making a data model modification, you must refresh the application in which you made them. If you change the data model in ContactManager, you must also refresh core application web services.

Procedure

1. Stop the applications in which you have made the data model modifications.
2. Optionally regenerate the *Data Dictionary* for the application by running `gwb genDataDictionary` from a command prompt.

   This step enables you to verify that your changes work before rebuilding the application.
3. If you have made a data model change to ContactManager, start ContactManager to refresh its web services.
4. Start Guidewire Studio for the core application and refresh the web service collection for that web service.
5. Start the applications and confirm your changes.

Limitations on configuring contact entity extensions

There are limitations on the kinds of things you can do with contact type extensions. Your subtype can inherit from only a single parent entity—multiple inheritance is not supported. For example, you could represent an adjudication practice with a single owner-proprietor either as a `Company` or as an `Adjudicator`. You cannot create a subtype that inherits the fields of both entities. To create the subtype you want, you can select one entity or the other and subtype it as something like `Practice`. Then, you add the fields that are missing from the other entity because of single inheritance.

Unless you restrict the visibility of some subtypes through configuration, they can still appear in search results because searches return all subtypes. This behavior has special implications if your installation is integrated with ContactManager. For example, suppose that you configure ClaimCenter to make `Adjudicator` not a choice in the user interface. However, in the ContactManager database there are a number of `ABAdjudicator` contact entities. Searching the **Address Book** for a `Person` returns `ABAdjudicator` matches if they exist in ContactManager.

See also

- For information about restricting access to contact subtypes, see "Securing access to contact information" on page 85.

Working with contact mapping files

If you extend your Guidewire core application's contact data model, for ContactManager to be able to work with the extension, you must also make a matching extension in ContactManager. You then map the entities to each other in both the Guidewire core application and in ContactManager.

A Guidewire core application uses the `ContactMapper` class to map the fields of contact entities sent to and received from ContactManager. There is a mapping class in each Guidewire core application and in ContactManager. The mapping classes, set up by default for the base application `Contact` types and the base ContactManager `ABContact` types, match Guidewire core application entity types to entity types in ContactManager.

IMPORTANT: When PolicyCenter receives an `Address` update from ContactManager, PolicyCenter uses the class `gw.webservice.pc.pc1000.contact.AddressDataCopier` to populate the address for the contact. PolicyCenter also uses `AddressDataCopier` to update linked addresses. If you make extensions to the Address object, ensure that you make the same updates in both `ContactMapper` and `AddressDataCopier`.

If you extend the contact data model, you must edit these classes and map every field for the new entity that you want to send and receive. In each `ContactMapper` class in the Guidewire core application and in ContactManager, you map both incoming and outgoing entities. If you leave a field out, it is not processed.

Additionally, you might need to map `Contact` subtypes and typecodes whose names in a Guidewire core application are different from the names in ContactManager. For example, Guidewire core applications use the `Person` subtype, which in ContactManager is `ABPerson`. All contact data passed through the ContactManager web services use the ContactManager domain namespace. Therefore, it is up to the core applications to map names, like `Contact`, between the core application domain namespace and the ContactManager domain namespace, which uses `ABContact`. There is a Gosu class in each Guidewire core application for this purpose as described in “Core application mapping”.

Note: There might be situations in which you add a contact subtype to a Guidewire core application and not to ContactManager. If you have a contact subtype that does not require central management, you can create that type in the Guidewire core application only and not use the mapping files.

ContactManager mapping

In ContactManager, you use the class `gw.contactmapper.ab1000.ContactMapper` to map contact entity fields between ContactManager and ClaimCenter or one of the other Guidewire core applications.

You can access this class in Guidewire Studio for ContactManager from the **Project** window. Navigate to **configuration** > **gsrc**, and then open `gw.contactmapper.ab1000.ContactMapper`.

For reference information on this class, see “ContactManager ContactMapper class” on page 314.

Core application mapping

In core applications, you use the class `gw.contactmapper.ab1000.ContactMapper` to map the fields of contact entities between ClaimCenter and ContactManager.

You can access this class in Guidewire Studio from the **Project** window. Navigate to **configuration** > **gsrc** and open `gw.contactmapper.ab1000.ContactMapper`.

For reference information on this class, see “ContactMapper class” on page 313.

While `ContactMapper` maps the fields of the entities, it does not map differing entity names. ContactManager contact entity names, such as `ABContact` or `ABPerson`, typically start with `AB`. Core application contact entity names, such as `Contact` or `Person`, typically do not start with `AB`. Additionally, there can be differences in typecodes between the entities. Core applications do their own name mapping—no changes are needed in ContactManager, and there is no equivalent class in ContactManager.

To support mapping of differing entity names and typecodes, the core applications use the following name mapping classes:

ClaimCenter

`gw.contactmapper.ab1000.CCNameMapper`

PolicyCenter

`gw.contactmapper.ab1000.PCNameMapper`

BillingCenter

`gw.contactmapper.ab1000.BCNameMapper`

See also

- “Map the new subtype names in ClaimCenter” on page 127

Extending the client data model

The techniques required to extend the client data model are essentially the same as those required to extend the vendor data model.

The primary differences are:

- The client data model includes only the `Contact`, `Person`, and `Company` entities.
- All three core Guidewire applications use client data. If you have more than one core application installed, you must apply your extension changes to all the applications. For example, you have installed the entire InsuranceSuite set of applications. If you extend the client data model in PolicyCenter and ContactManager, you must also extend it in ClaimCenter and BillingCenter.
- PolicyCenter and BillingCenter PCF files are different from those of ClaimCenter.

See also

- “Extending the vendor contact data model” on page 118
- “Overview of contact entities” on page 109

Extending the vendor contact data model

You can extend the vendor contact data model used in ClaimCenter to add your own subtypes or fields. The extension examples show some common configurations you can use to extend your own installation.

Adding a field to a contact subtype

You can extend a contact with a new field.

This example is a multi-step process divided into a series of topics. In this example, you add a new `BoardCertified_Ext` field to the `Doctor` subtype in ClaimCenter and to the `ABDoctor` subtype in ContactManager. You then update the necessary files and screens to make the new field usable across the applications.

Add a field to the Doctor subtype in ClaimCenter

As part of an example of adding a field to a contact subtype, extend the `Doctor` entity in ClaimCenter.

Before you begin

If you have not done so already, follow the instructions for installing ContactManager in “Installing ContactManager” on page 17. Additionally:

- You must have integrated ContactManager with ClaimCenter as described in “Integrating ContactManager with Guidewire core applications” on page 21.
- If you do not have any users or claims defined, you can load sample data for both applications. See “Load sample data for ContactManager” on page 18.

About this task

This step is part of the example “Adding a field to a contact subtype” on page 118.

Procedure

1. Start Guidewire Studio™ for ClaimCenter.

   At a command prompt, navigate to the ClaimCenter installation folder and enter:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Entity**.
3. Double-click `Doctor.eti` to open it in the Entity editor.
4. Click **subtype** at the top of the **Element** hierarchy.
5. Click the drop-down list for ➕ and choose **column**.
6. Enter the following values for the new column:

Name	Value
name	`BoardCertified_Ext`
type	`bit`
nullok	`false`
desc	`Is the doctor Board certified in the specialty?`
default	`false`

7. Regenerate the *Data Dictionary* to ensure that your changes were correct.

   At a command prompt, navigate to the ClaimCenter installation folder and enter:

```
gwb genDataDictionary
```

What to do next

"Add the new field to the ContactMapper class in ClaimCenter" on page 119

Add the new field to the ContactMapper class in ClaimCenter

For ClaimCenter to be able to send a new `Contact` field to ContactManager and receive updates for the field, you must add it to the `ContactMappper` class.

Before you begin

Complete "Add a field to the Doctor subtype in ClaimCenter " on page 118.

About this task

For more information on the `ContactMappper` class, see "ContactMapper class" on page 313.

Procedure

1. In Guidewire Studio™ for ClaimCenter, press `Ctrl+N` and enter `ContactMapper`.
2. Double-click the `ab1000` version of the class in the search results to open it in the Gosu editor.
3. Press `Ctrl+F` and search for the following line of code:

```
fieldMapping(Doctor#MedicalLicense),
```

4. On a new line after the `MedicalLicense` line, enter `fieldMapping(Doctor#Board` and press `Ctrl+Space` to complete the property name, and then add a comma. The new line will be:

```
fieldMapping(Doctor#BoardCertified_Ext),
```

5. If necessary, stop ClaimCenter as follows:
   a) Open a command prompt in the ClaimCenter installation folder and then enter the following command:

   ```
   gwb stopServer
   ```

   b) Wait to restart the ClaimCenter application and pick up the data model changes until you have made the ClaimCenter user interface changes in a later topic.

What to do next

"Add the BoardCertified field to the Address Book user interface" on page 120

Add the BoardCertified field to the Address Book user interface

In this step, you modify the ClaimCenter Address Book screen so you can see the medical specialty field with the `Doctor` subtype.

Before you begin

Complete "Add the new field to the ContactMapper class in ClaimCenter" on page 119.

About this task

Update the input sets used by the detail view.

Procedure

1. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
2. Press `Ctrl+F` and search for the following entry in the file:

   ```
   Web.ContactDetail.Doctor.MedicalLicense
   ```

3. Add a line under that entry, and then enter the following key and value:

   `Web.ContactDetail.Doctor.BoardCertified = Board Certified`

4. In the **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **addressbook**. Then double-click **AddressBookDoctorAdditionalnfoInputSet.Doctor** to open it in the editor.
5. Select the `Input` widget `Medical Specialty` and copy it, and then paste the copy below this widget.
6. Right-click the new field and choose **Change element type**.
7. Click **Boolean Radio Button Input** in the drop-down list.
8. Click the new `BooleanRadioInput` widget and set the following properties:

editable	`true`
id	`DoctorBoardCertified`
label	`displaykey.Web.ContactDetail.Doctor.BoardCertified`
required	`false`
value	`(personVendor as Doctor).BoardCertified_Ext`

What to do next

"Add the BoardCertified field to the claim contacts user interface" on page 120

Add the BoardCertified field to the claim contacts user interface

Enable the ClaimCenter user to edit the `BoardCertified` field of a `Doctor` entity when working with contacts for a claim.

Before you begin

Complete "Add the BoardCertified field to the Address Book user interface" on page 120.

Procedure

1. In the Guidewire Studio™ for ClaimCenter **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **shared** > **contacts.** Then double-click `DoctorAdditionalInfoInputSet.Doctor` to open it in the editor.
2. Select the `Input` widget `Medical Specialty` and copy it, and then paste the copy below this widget.
3. Right-click the new field and choose **Change element type**.
4. Click **Boolean Radio Button Input** in the drop-down list.
5. Click the new `BooleanRadioInput` widget and set the following properties:

editable	`true`
id	`DoctorBoardCertified`
label	`displaykey.Web.ContactDetail.Doctor.BoardCertified`
required	`false`
value	`Doctor.BoardCertified_Ext`

What to do next

"Add a field to the ABDoctor subtype in ContactManager " on page 121

Add a field to the ABDoctor subtype in ContactManager

In this example of adding a field to a contact subtype, you extend the `ABDoctor` entity in ContactManager.

Before you begin

Complete "Add the BoardCertified field to the claim contacts user interface" on page 120.

Procedure

1. Start Guidewire Studio™ for ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter:

```
gwb studio
```

2. In the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Entity**.
3. Double-click `ABDoctor.eti` to open it in the Entity editor.
4. Click **subtype** at the top of the **Element** hierarchy.
5. Click the drop-down list next to ✚ and choose **column**.
6. Enter the following values for the new column:

Name	Value
name	`BoardCertified_Ext`
type	`bit`
nullok	`true`
desc	`Is the doctor Board certified in the specialty?`
default	`false`

7. Regenerate the *Data Dictionary* to ensure that your changes are correct.

   At a command prompt, navigate to the ContactManager installation folder and enter:

```
gwb genDataDictionary
```

What to do next

"Add the new field to the ContactMapper class in ContactManager" on page 122

Add the new field to the ContactMapper class in ContactManager

To be able to send a new `ABContact` field to a Guidewire core application and receive updates for the field, you must add it to the `ContactMappper` class.

Before you begin

Complete the step "Add a field to the ABDoctor subtype in ContactManager " on page 121.

About this task

For more information on the `ContactMapper` class, see "ContactMapper class" on page 313.

Procedure

1. In Guidewire Studio™ for ContactManager, click `Ctrl+N` and enter `ContactMapper`, and then double-click the `ab1000` version of the class in the search results to open it in the Gosu editor.
2. Press `Ctrl+F` and find the following line of code:

   ```
   fieldMapping(ABDoctor#MedicalLicense),
   ```

3. Add the following code for the new field after the line for the `MedicalLicense` field:

   `fieldMapping(ABDoctor#BoardCertified_Ext),`
4. If necessary, stop ContactManager as follows:
   a) Open a command prompt in the ContactManager installation folder and then enter the following command:

      ```
      gwb stopServer
      ```

   b) Wait to restart the ContactManager application and pick up the data model changes until after making the changes in the next step.

What to do next

"Add the BoardCertified field to the ContactManager user interface" on page 122

Add the BoardCertified field to the ContactManager user interface

Enable ContactManager users to use the new `BoardCertified` field when they create and edit `ABDoctor` contacts.

Before you begin

Complete "Add the new field to the ContactMapper class in ContactManager" on page 122.

About this task

You add the new field to the input set for the `ABDoctor` subtype for use in the `ContactBasicsDV.ABPerson` detail view.

Procedure

1. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
2. Click `Ctrl+F` and find the following entry in the file:

   ```
   Web.ContactDetail.Doctor.MedicalLicense
   ```

3. Insert a line after that entry and then enter the following key and value:
   `Web.ContactDetail.Doctor.BoardCertified = Board Certified`

4. In the **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **contacts** > **basics**.
5. Double-click `ABPersonVendorSpecialtyInputSet.ABDoctor` to open it in the editor.
6. Select the `Input` widget `Medical Specialty` and copy it, and then paste the copy below the `Medical Specialty` widget.
7. Right-click the new field and choose **Change element type**.
8. Click **Boolean Radio Button Input**.
9. Click the new `BooleanRadioInput` widget and set the following properties:

editable	`true`
id	`DoctorBoardCertified`
label	`displaykey.Web.ContactDetail.Doctor.BoardCertified`
required	`false`
value	`(person as ABDoctor).BoardCertified_Ext`

What to do next

"Restart both applications and test the new field" on page 123

Restart both applications and test the new field

In this step of adding a field to a contact subtype, you restart ClaimCenter and ContactManager to pick up all the changes made in the previous steps. Then you test the new field to ensure that both applications can use it and can communicate with each other.

Before you begin

Complete "Add the BoardCertified field to the ContactManager user interface" on page 122 before starting this step

Procedure

1. If necessary, stop both ClaimCenter and ContactManager, and then restart both applications.
2. Log in to ClaimCenter as a user who can create new contacts.
   For example, log in as user `ssmith` with password `gw`.
3. Open an existing claim.
4. Click **Parties Involved** in the left Sidebar, and then on the **Contacts** screen choose **New Contact** > **Vendor** > **Doctor** to open the **New Doctor** screen.
5. Verify that **Board Certified** is listed in the **Additional Info** section.
6. Enter enough information to create a `Doctor` contact.

   The minimum is **First name**, **Last name**, and **Tax ID**. Include a **Medical Specialty** and click **Yes** for **Board Certified**.
7. At the top of the edit screen in the **Roles** list view table, click **Add**.
8. Click the **Role** cell for the new entry and choose a role from the drop-down list that the new `Doctor` vendor will take on the claim. For example, choose **Doctor**.
9. Click **Update** to save the new contact to the claim.
   ClaimCenter adds the new `Doctor` contact to the list of contacts on the **Contacts** screen. ClaimCenter also ensures that the Claim Party and Vendor tags are set for the contact and sends it to ContactManager.
10. Select the new doctor vendor in the **Contacts** list view table.
    If you logged in as a user with permission to create contacts, below, on the **Basics** card, the message above the contact says:

    ```
    This contact is linked to the Address Book and is in sync
    ```

This message means that the new contact has been saved to ContactManager, and the contact data for this contact on this claim is the same for ClaimCenter and ContactManager.

It is possible that the contact is still being saved to ContactManager and ClaimCenter has not yet been notified. In that case, the message you see is `Waiting for link message from ContactManager. Refresh screen to get updated status`. You can refresh the screen by clicking another contact or screen and then clicking this contact again.

11. Click **View in Address Book** to see a screen showing the data saved for this contact in ContactManager.
12. On this screen, click **Edit in ContactManager** to open ContactManager and edit the contact.

    You might have to log in to ContactManager if your ClaimCenter user name is not in ContactManager. Log in as a ContactManager user who has `ABContact` view, edit, update, and delete permissions, such as the sample user aapplegate.
13. Click **Edit** and make some changes to the contact, such as a new address. Change the setting for the **Board Certified** field. Click **Update** to save your changes.
14. In ClaimCenter, click the **Address Book** tab.
15. On the **Search Address Book** screen, pick **Doctor** from the **Type** drop-down list and search for the doctor you added.
16. Click the contact found by **Search** and verify that the entry found by ClaimCenter in ContactManager is correct and that the data matches the changes you made in ContactManager.
17. Click the drop-down list for the **Claim** tab and choose the claim you previously edited.
18. Click **Parties Involved** in the left Sidebar, and then, on the **Contacts** screen, select the doctor you added to this claim.
    Below, on the **Basics** card, the message above the contact says:

    ```
    This contact is linked to the Address Book but is out of sync
    ```

    This message means that the contact data you changed in ContactManager is now different from the contact data for this contact on this claim.
19. Click **Copy from Address Book** to update the contact data for this claim.
    You see the changes on the **Basics** card and the following message:

    ```
    This contact is linked to the Address Book and is in sync
    ```
20. Click **Edit** and change the setting for the **Board Certified** field, and then click **Update**.

    Because you are logged in as a user who can edit contacts, this change is sent to ContactManager.

    The message above the contact changes to:

    ```
    This contact is linked to the Address Book but is out of sync
    ```

    This message means that the change you made to the contact has not yet registered in ContactManager. It can take a few seconds for the message sent to ContactManager to take effect.
21. Click another contact in the **Contacts** list view table, and then click your original `Doctor` contact to refresh the **Basics** card.

    If the message has not changed to say that the contact is in sync, wait a few seconds, and then click another contact and then this one again.

    Eventually, you see the following message:

    ```
    This contact is linked to the Address Book and is in sync
    ```

Adding a vendor contact subtype

You can extend contacts with a new subtype. In this multi-topic example, you add a new `Interpreter` subtype to ClaimCenter and its counterpart, `ABInterpreter`, to ContactManager. You then update the necessary files and screens to make the new field usable across the applications.

Note: Creating a `Contact` subtype is usually not necessary if you define vendor services. See “Vendor services” on page 163. Additionally, see “Deciding whether to create a subtype” on page 115.

Add a new Contact subtype to ClaimCenter

Part of adding a vendor contact subtype is extending the `Contact` entity in ClaimCenter.

Before you begin

If you have not done so already:

- Follow the instructions for installing ContactManager in “Installing ContactManager” on page 17.
- You must have integrated ContactManager with ClaimCenter as described in “Integrating ContactManager with Guidewire core applications” on page 21.
- After installing ContactManager, for testing purposes, it would be helpful to import sample data as described in “Load sample data for ContactManager” on page 18.

About this task

This step is the first in the example “Adding a vendor contact subtype” on page 125. In this step, you add a new `Interpreter` entity in ClaimCenter.

Procedure

1. Start Guidewire Studio™ for ClaimCenter.

   At a command prompt, navigate to the ClaimCenter installation folder and enter:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Entity**.
3. Right-click **Entity** and choose **New** > **Entity**.
4. In the **Entity** field, enter the following name:

   `Interpreter`

5. Click the **Entity Type** drop-down list and choose **subtype**.
6. In the **Desc** field, enter the following description:

   `Interpreter`

7. Click the **Supertype** search button ... and then choose **PersonVendor** from the drop-down list.
8. Click **OK**.
9. In the Entity editor for the new entity, click **subtype** at the top of the **Element** hierarchy.
10. For **displayName**, enter `Interpreter`.
11. Click the drop-down list next to ✚ and choose **column**.
12. Enter the following values for the new column:

Name	Value
name	`InterpreterSpecialty`

Name	Value
type	`varchar`
nullok	`false`
desc	`Interpreter's language specialties`

13. Click the drop-down list next to ✚ and choose **columnParam**.
14. Enter the following values to specify the size of this `varchar` column:

Name	Value
name	`size`
value	`30`

15. Regenerate the *Data Dictionary*.

    At a command prompt, navigate to the ClaimCenter installation folder and enter:

    ```
    gwb genDataDictionary
    ```

16. Verify in the *Data Dictionary* that the data model extension is correct.
17. Close Guidewire Studio for ClaimCenter and then restart it to enable Studio to pick up the data model change.

What to do next

"Add the new subtype to the ContactMapper class in ClaimCenter" on page 126

Add the new subtype to the ContactMapper class in ClaimCenter

To be able to send a new `Contact` subtype to ContactManager and receive updates for it, you must add it to the `ContactMappper` class.

Before you begin

Complete "Add a new Contact subtype to ClaimCenter " on page 125.

About this task

For more information on the `ContactMappper` class, see "ContactMapper class" on page 313.

Procedure

1. In Guidewire Studio™ for ClaimCenter, press `Ctrl+N` and enter `ContactMapper`, and then double-click the `ab1000` version of the class in the search results to open it in the Gosu editor.
2. Press `Ctrl+F` and search for the following line of code:

    ```
    fieldMapping(MedicalCareOrg#MedicalOrgSpecialty),
    ```

3. After the line for `MedicalCareOrg`, add the following code for the new entity:

    ```
    fieldMapping(Interpreter#InterpreterSpecialty),
    ```

    Note: You do not have to put this `fieldMapping` method call in this exact location. It must be in the method `override property get Mappings() : Set<CCPropertyMapping>`. Additionally, it is useful to group it with the other method calls after the comment `//Other Contact subtypes`.

What to do next

"Map the new subtype names in ClaimCenter" on page 127

Map the new subtype names in ClaimCenter

To be able to send a new `Contact` subtype to ContactManager and receive updates for it, you must map the ClaimCenter subtype name to the ContactManager `ABContact` subtype name. You do this mapping only in ClaimCenter in the class `gw.contactmapper.ab1000.CCNameMapper`.

Before you begin

Complete “Add the new subtype to the ContactMapper class in ClaimCenter” on page 126.

About this task

The ClaimCenter subtype is `Interpreter` and the ContactManager subtype, which you will create later, is `ABInterpreter`.

Procedure

1. In Guidewire Studio™ for ClaimCenter, press `Ctrl+N` and enter `CCNameMapper`.
2. In the search results, double-click the `ab1000` version of the class to open it in the Gosu editor.
3. In the editor, press `Ctrl+F` and search for `MedicalCareOrg`. The search finds the following line of code:

```
.entity(MedicalCareOrg, "ABMedicalCareOrg")
```

4. Insert the following entry below the line for `MedicalCareOrg`.

```
.entity(Interpreter, "ABInterpreter")
```

5. If ClaimCenter is running, stop ClaimCenter as follows:
   a) Open a command prompt in the ClaimCenter installation folder and then enter the following command:

   ```
   gwb stopServer
   ```

   b) Do not restart the ClaimCenter application until after modifying the ClaimCenter Address Book in a later step.

What to do next

“Add display keys for the new subtype to ClaimCenter” on page 127

Add display keys for the new subtype to ClaimCenter

Adding a new `Interpreter` subtype requires changes to the user interface. Add display keys to ClaimCenter for menu items and fields.

Before you begin

Complete “Map the new subtype names in ClaimCenter” on page 127.

About this task

The new menu items and inputs for `Interpreter` have labels that require display keys, which support localization.

Procedure

1. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
2. In the editor, press `Ctrl+F` and find the entry `Web.NewContactMenu.Doctor` in the file.
3. Insert a line under that entry, and then enter the following key and value:

```
Web.NewContactMenu.Interpreter = Interpreter
```

4. Use the search field at the top of the editor to find the entry `Web.ContactDetail.Email.Secondary` in the file.
5. Insert a new line under that entry, and then enter the following keys and values:

```
Web.ContactDetail.Interpreter = Interpreter
Web.ContactDetail.Interpreter.InterpreterSpecialty = Interpreter Specialty
```

What to do next

"Add the subtype to the ClaimCenter New Contact menu" on page 128

Add the subtype to the ClaimCenter New Contact menu

Modify the ClaimCenter `New Contact` menu to enable the user to create contacts with the new `Interpreter` subtype.

Before you begin

Complete "Add display keys for the new subtype to ClaimCenter" on page 127.

About this task

Note: In general, when you create new subtypes, the PCF files you need to edit depend on the parent of the subtype. For example, `Person` subtypes use a different set of PCF files from `Company` subtypes. You can discover the specific PCF files you need by examining those used by other subtypes. Additionally, the PCF files can use different modes to show appropriate fields for different subtypes.

Procedure

1. In Guidewire Studio™ for ClaimCenter, navigate in the **Project** window to **configuration** > **config** > **Page Configuration** > **pcf** > **claim** > **partiesinvolved**.
2. Double-click `ClaimContacts` to open that PCF file in the editor.
3. On the toolbar above the list view panel with the ID `PeopleInvolvedDetailedLV`, there is a `New Contact` button. Click the drop-down control to open the embedded menu.
4. Select the `Vendor` submenu, which has the **id** value `ClaimContacts_NewVendor`.

   This submenu has menu elements for vendors like Autobody Repair Shop, Doctor, and Medical Care Organization.
5. Right-click the menu item widget for `Medical Care Organization` and copy it.
6. Paste the copy below the menu item for `Medical Care Organization`.

   The new menu item is in the `Vendor` submenu. The new item turns red and stays that way until you enter the properties in the next step.
7. Click the new menu item widget and set the following properties:

action	`NewPartyInvolvedPopup.push(claim, typekey.Contact.TC_INTERPRETER)`
id	`ClaimContacts_Interpreter`
label	`displaykey.Web.NewContactMenu.Interpreter`
showConfirmMessage	`true`

8. Navigate in the **Project** window to **configuration** > **config** > **Page Configuration** > **pcf** > **shared** > **contacts**.
9. Double-click `ClaimNewServiceRequestSpecialistPickerMenuItemSet` to open it in the editor.
10. Right-click the menu item widget for `Medical Care Organization` and copy it.
11. Paste the copy below the menu item for `Medical Care Organization`.

    The new menu item is in the `New Vendor` submenu. The new item turns red and stays that way until you enter the properties in the next step.

12. Click the new menu item widget and set the following properties:

action	`NewContactPopup.push(typekey.Contact.TC_INTERPRETER, parentContact, claim)`
id	`NewInterpreter`
label	`displaykey.Web.NewContactMenu.Interpreter`
showConfirmMessage	`true`

What to do next

"Add the new subtype to the Address Book input sets" on page 129

Add the new subtype to the Address Book input sets

Enable a ClaimCenter user to select an `Interpreter` contact and see the **Interpreter Specialty** field.

Before you begin

Complete "Add the subtype to the ClaimCenter New Contact menu" on page 128.

About this task

In this step you update the input sets used by the ClaimCenter `Address Book` detail view.

Procedure

1. In Guidewire Studio™ for ClaimCenter, navigate in the **Project** window to **configuration** > **config** > **Page Configuration** > **pcf** > **addressbook**.
2. Right-click `AddressBookDoctorAdditionalInfoInputSet.Doctor` and click **Copy**.
3. Right-click again and choose **Paste**.
4. In the **Copy** dialog box, enter the new name `AddressBookInterpreterAdditionalInfoInputSet.Interpreter.pcf`, and then click **OK**.
   The new widget opens in the editor.
5. Click the name of the input set window, `AddressBookInterpreterAdditionalInfoInputSet`, to open the **Properties** tab.
6. Set the value of the `mode` property to `Interpreter`.
7. In the editor, select the `Medical License` input widget and set the following properties:

editable	`true`
id	`InterpreterSpecialty`
label	`displaykey.Web.ContactDetail.Interpreter.InterpreterSpecialty`
required	`false`
value	`(personVendor as Interpreter).InterpreterSpecialty`

8. Select the `Medical Specialty` widget and delete it.
9. If it exists, select the `Board Certified` widget and delete it.
10. In the **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **addressbook**.
11. Open `AddressBookContactBasicsDV.Person` in the editor and click its name at the top to open the **Properties** tab.
12. For the `mode` property, add the `Interpreter` mode to the list:

```
Person|PersonVendor|Adjudicator|UserContact|Doctor|Attorney|Interpreter
```

13. Click **OK**.

14. Open `AddressBookAdditionalInfoInputSet.PersonVendor` in the editor and click its name at the top to open the **Properties** tab.
15. For the `mode` property, add the `Interpreter` mode to the list:

```
PersonVendor|Attorney|Doctor|Interpreter
```

16. Drag a new `InputSetRef` widget from the **Toolbox** and drop it below the second `InputSetRef` widget, the one containing the input set `AddressBookAttorneyAdditionalInfoInputSet`.
17. Click the new `InputSetRef` widget and set the following properties:

def	`AddressBookInterpreterAdditionalInfoInputSet(contact as PersonVendor)`
id	`AddressBookInterpreterAdditionalInfoInputSet`
mode	`contact typeis Interpreter ? "Interpreter" : null`

The mode statement enables the `Interpreter Specialty` field to display in the `Additional Info` section when the contact type is `Interpreter`.

What to do next

"Add the new contact to the shared contacts detail view" on page 130

Add the new contact to the shared contacts detail view

Enable the ClaimCenter user to create an `Interpreter` contact on the **New Contact** screen and see the **Interpreter Specialty** field.

Before you begin

Complete "Add the new subtype to the Address Book input sets" on page 129.

About this task

In this step, you update the input sets used by the shared contacts detail view.

Procedure

1. In Guidewire Studio™ for ClaimCenter, navigate in the **Project** window to **configuration** > **config** > **Page Configuration** > **pcf** > **shared** > **contacts**.
2. Open `ContactBasicsDV.Person` and click the top line in the screen, `Detail View: ContactBasicsDV`, to open the **Properties** tab.
3. For the `mode` property, add `Interpreter` to the list of modes, as follows:

```
Person|PersonVendor|Adjudicator|UserContact|Doctor|Attorney|Interpreter
```

4. In the same **Project** window folder, navigate to **pcf** > **shared** > **contacts**.
5. Right-click `DoctorAdditionalInfoInputSet.Doctor` and click **Copy**.
6. Right-click again and choose **Paste**.
7. In the **Copy** dialog box, enter the new name `InterpreterAdditionalInfoInputSet.Interpreter.pcf`, and then click **OK**.
   The new widget opens in the editor.
8. In the **Project** window, open `InterpreterAdditionalInfoInputSet.Interpreter`.
9. In the editor, select the new widget by clicking its identifier at the top of the widget: `InputSet: InterpreterAdditionalInfoInputSet` to open the **Properties** tab.
10. Verify that `Interpreter` is set in the `mode` property.

11. Click the **Code** tab and replace the existing code with the following Gosu statement:

    `property get Interpreter() : Interpreter {return contactHandle.Contact as Interpreter;}`

12. Select the `Medical License` input widget and change it to an `Interpreter Specialty` widget by setting the following properties:

editable	`true`
id	`InterpreterSpecialty`
label	`displaykey.Web.ContactDetail.Interpreter.InterpreterSpecialty`
required	`false`
value	`Interpreter.InterpreterSpecialty`

13. Select the `Medical Specialty` widget and delete it.
14. If it exists, select the `Board Certified` widget and delete it.
15. In the same **Project** window location, **pcf** > **shared** > **contacts**, open `AdditionalInfoInputSet.PersonVendor`.
16. In the editor, select the new widget by clicking its identifier at the top of the widget: `InputSet: AdditionalInfoInputSet` to open the **Properties** tab.
17. Click the `mode` property and add the `Interpreter` mode to the list:

```
PersonVendor|Attorney|Doctor|Interpreter
```

18. Drag a new `InputSetRef` widget from the **Toolbox** and drop it below the widget containing the input set `AttorneyAdditionalInfoInputSet`.
19. Click the new `InputSetRef` widget and set the following properties:

def	`InterpreterAdditionalInfoInputSet(contactHandle)`
mode	`PersonVendor typeis Interpreter ? "Interpreter" : null`

    The mode statement enables the `Interpreter Specialty` field to show in the **Additional Info** section when the contact type is `Interpreter`.

What to do next

"Add a new ABContact subtype to ContactManager" on page 131

Add a new ABContact subtype to ContactManager

Part of adding a vendor contact subtype is extending the `ABContact` entity in ContactManager.

Before you begin

Complete "Add the new contact to the shared contacts detail view" on page 130.

About this task

In this step, you add an `ABInterpreter` subtype to ContactManager. This subtype is the counterpart of the `Interpreter` subtype you added to ClaimCenter in "Add a field to the Doctor subtype in ClaimCenter " on page 118.

Procedure

1. Start Guidewire Studio™ for ContactManager. At a command prompt, navigate to the ContactManager installation folder and enter the following command:

```
gwb studio
```

2. In the **Project** window open in **Project** view, navigate to **configuration** > **config** > **Extensions** > **Entity**.
3. Right-click **Entity** and choose **New** > **Entity**.
4. In the **Entity** field, type the following name: `ABInterpreter`
5. Click the **Entity Type** drop-down list and choose **subtype**.
6. In the **Desc** field, type the following description: `Interpreter`
7. Click the **Supertype** search button ... and choose `ABPersonVendor` from the drop-down list.
8. Click **OK**.
9. In the Entity editor for the new entity, click **subtype** at the top of the **Element** hierarchy.
10. For **Display name** , enter `Interpreter`.
11. Click the drop-down list next to ✚ and choose **column**.
12. Enter the following values for the new column:

Name	Value
name	`InterpreterSpecialty`
type	`varchar`
nullok	`false`
desc	`Interpreter's language specialties`

13. Click the drop-down list next to ✚ and choose **columnParam**.
14. Enter the following values to specify the size of this `varchar` column:

Name	Value
name	`size`
value	`30`

15. Click **subtype** at the top of the **Element** hierarchy.
16. Click the drop-down list next to ✚ and choose **index**.
17. Enter the following values for the new index:

Name	Value
name	`intrprtrindx1`
desc	`Speed up searches using InterpreterSpecialty field`
unique	`true`

18. Right-click **index** in the **Element** column, and choose **Add new** > **indexcol**.

    This index column is the first of three to add to the index.
19. Enter the following values for the new `indexcol`:

Name	Value
name	`InterpreterSpecialty`
keyposition	`1`

20. Right-click **index** in the **Element** column, and choose **Add new** > **indexcol**.
21. Enter the following values for the new `indexcol`:

Name	Value
name	`Retired`
keyposition	2

22. Right-click **index** in the **Element** column, and choose **Add new** > **indexcol**.
23. Enter the following values for the new `indexcol`:

Name	Value
name	`Id`
keyposition	3

24. Regenerate the data dictionary to ensure that your changes are correct.

    At a command prompt, navigate to the ContactManager installation folder and enter:

```
gwb genDataDictionary
```

What to do next

"Add the new subtype to the ContactMapper class in ContactManager" on page 133

Add the new subtype to the ContactMapper class in ContactManager

To be able to send the new subtype to a Guidewire core application and receive updates for the subtype, you must add it to the `ContactMappper` class.

Before you begin

Complete "Add a new ABContact subtype to ContactManager" on page 131.

About this task

For more information on the `ContactMappper` class, see "ContactMapper class" on page 313.

Procedure

1. In Guidewire Studio™ for ContactManager, press `Ctrl+N` and enter `ContactMapper`, and then double-click the `ab1000` version of the class in the search results to open it in the Gosu editor.
2. Press `Ctrl+F` and search for the following line of code:

```
fieldMapping(ABMedicalCareOrg#MedicalOrgSpecialty),
```

3. After the line for `ABMedicalCareOrg`, add the following code for the new entity:

   `fieldMapping(ABInterpreter#InterpreterSpecialty),`

   Note: You do not have to put this `fieldMapping` method call in this exact location. It must be in the method with declaration `override property get Mappings() : Set<PropertyMapping>`. Additionally, it is useful to group it with the other method calls after the comment `// Other ABContact subtypes`.

4. If ContactManager is running, you must stop and restart it to pick up the data model change. Wait to restart the ContactManager application until you complete the step "Restart both applications and test the new subtypes" on page 136.

What to do next

"Add display keys for the new subtype to ClaimCenter" on page 127

Add display keys for the new subtype to ContactManager

Adding a new `ABInterpreter` subtype requires changes to the user interface. Add display keys to ContactManager for menu items and fields.

Before you begin

Complete “Add the new contact to the shared contacts detail view” on page 130.

About this task

In this step, you add display keys to ContactManager to support changes to the user interface screens for the new `ABInterpreter` subtype.

Procedure

1. Open Guidewire Studio™ for ContactManager.
2. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
3. Press `Ctrl+F` and search for `Web.NewContactMenu.Doctor`.
4. Add a new line under that entry, and then enter the following key and value:

   ```
   Web.NewContactMenu.Interpreter = Interpreter
   ```
5. Use the Search field at the top of the editor to find the entry `Web.ContactDetail.Email.Secondary`.
6. Insert a new line under that entry, and then enter the following keys and values:

   ```
   Web.ContactDetail.Interpreter = Interpreter
   Web.ContactDetail.Interpreter.InterpreterSpecialty = Interpreter Specialty
   ```

What to do next

“Add the new subtype to the ContactManager actions menu” on page 134

Add the new subtype to the ContactManager actions menu

As part of adding a new vendor contact subtype, you can add an **Actions** menu entry for creating a new `ABInterpreter` contact.

Before you begin

Complete “Add display keys for the new subtype to ContactManager” on page 134.

Procedure

1. In Guidewire Studio™ for ContactManager, navigate in the **Project** window to **configuration** > **config** > **Page Configuration** > **pcf** > **contacts** and double-click `ContactsMenuActions` to open this file in the PCF editor.
2. Drag a `MenuItem` widget from the **Toolbox** on the right and drop it under the `Doctor` menu item.

   The new menu item is in the `Vendor` submenu, which has ID `ContactsMenuActions_PersonVendorMenuItem`.
3. Click the new `MenuItem` and enter the following values for the **Basic properties** of this `Interpreter` menu item:

action	`NewContact.go(entity.ABInterpreter)`
id	`ContactsMenuActions_InterpreterMenuItem`
label	`displaykey.Web.NewContactMenu.Interpreter`
showConfirmMessage	`true`

What to do next

"Add ABInterpreter fields to ContactManager modal input sets" on page 135

Add ABInterpreter fields to ContactManager modal input sets

Part of adding a new `ABContact` subtype is configuring a screen for entering data for the new entity.

Before you begin

Complete "Add the new subtype to the ContactManager actions menu" on page 134.

About this task

In this step, you add a modal input set supporting the `ABInterpreter` contact.

Procedure

1. In Guidewire Studio™ for ContactManager, navigate in the **Project** window to **configuration** > **config** > **Page Configuration** > **pcf** > **contacts** > **basics**.
2. Right-click `ABPersonVendorSpecialtyInputSet.ABDoctor` and choose **Copy**.
3. Right-click again and choose **Paste**.
4. Name the new file `ABPersonVendorSpecialtyInputSet.ABInterpreter.pcf` and click **OK**.
5. In the new PCF file, click its name, `ABPersonVendorSpecialtyInputSet`, at the top to open the **Properties** tab.
6. Verify that the `mode` property is set to `ABInterpreter`.
7. In the editor, click the `Medical License` input widget and change it to an `Interpreter Specialty` widget by setting the following properties:

editable	`true`
id	`InterpreterSpecialty`
label	`displaykey.Web.ContactDetail.Interpreter.InterpreterSpecialty`
required	`false`
value	`(person as ABInterpreter).InterpreterSpecialty`

8. Select the `Medical Specialty` widget and delete it.
9. If it exists, select the `Board Certified` widget and delete it.
10. In the same location in the **Project** window, **pcf** > **contacts** > **basics**, open `ABPersonVendorInputSet.ABPersonVendor`.
11. Click the name of the PCF file at the top, `ABPersonVendorInputSet`, to open the **Properties** tab.
12. Add `ABInterpreter` to the mode property as follows:

```
ABPersonVendor|ABAttorney|ABDoctor|ABInterpreter
```

13. In the same location in the **Project** window, **pcf** > **contacts** > **basics**, open `ContactBasicsDV.ABPerson` and click its name at the top to open the **Properties** tab.
14. Add `ABInterpreter` to the `mode` property, as follows:

```
ABPerson|ABPersonVendor|ABAdjudicator|ABUserContact|ABDoctor|ABAttorney|ABPolicyPerson|ABInterpreter
```

15. Click **OK** to save the new mode.

What to do next

"Add the new subtype to the ContactManager contact picker menus" on page 136

Add the new subtype to the ContactManager contact picker menus

Part of adding an `ABContact` extension is configuring the ContactManager contact picker menu so the user can choose the new entity.

Before you begin

Complete “Add ABInterpreter fields to ContactManager modal input sets” on page 135.

About this task

Add the new `ABInterpreter` entity to the ContactManager picker menu.

Procedure

1. In Guidewire Studio™ for ContactManager, navigate in the **Project** window to **configuration** > **config** > **Page Configuration** > **pcf** > **contacts**.
2. Double-click `NewContactPickerMenuItemSet` to open this file in the PCF editor.
3. Select the submenu labeled `Vendor`, which has the following **id** property:

```
NewContactPickerMenuItemSet_PersonVendorMenuItem
```

4. Copy the `Doctor` menu item and paste the copy under the existing `Doctor` menu item.
5. Select the new menu item.
   The menu item turns red, indicating an error. It stays red until you set the properties in the next step.
6. Set the following properties for this new menu item:

action	`NewContactPopup.push(entity.ABInterpreter, parentContact)`
id	`NewContactPickerMenuItemSet_InterpreterMenuItem`
label	`displaykey.Web.NewContactMenu.Interpreter`
showConfirmMessage	`true`
visible	`requiredContactType.isAssignableFrom(entity.ABInterpreter)`

What to do next

“Restart both applications and test the new subtypes” on page 136

Restart both applications and test the new subtypes

After adding new contact subtypes and configuring the user interfaces in ClaimCenter and ContactManager, restart both applications and test that they support and communicate about the new subtypes.

Before you begin

Complete “Add the new subtype to the ContactManager contact picker menus” on page 136.

Procedure

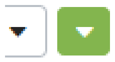
1. If they are running, stop the ClaimCenter and ContactManager servers, and then restart them to refresh each application with your PCF and data model changes.
2. Log in to ClaimCenter as a user who can create new contacts.
   For example, log in as user `ssmith` with password `gw`.
3. Open a claim and navigate to the **Parties Involved** > **Contacts** screen.
4. Choose **New Contact** > **Vendor** > **Interpreter** to see the **New Interpreter** dialog. Verify that **Interpreter Specialty** is listed in the **Additional Info** section.

5. Enter enough information to create an interpreter contact, such as name, address, interpreter specialty, and tax ID.
6. Add a role for the contact on the claim, and then click **Update** to save the new contact information.
7. Because you are logged in as a user with permission to create contacts, ClaimCenter sends the new contact to ContactManager.
   If you click the contact on the Contacts screen right away, you are likely to see the following message:

   `Waiting for link message from ContactManager. Refresh screen to get updated status.`
8. You can refresh the screen by clicking another contact on the list and then clicking this one again.
   Eventually, you see the following message:

   `This contact is linked to the Address Book and is in sync`
9. Click the **Address Book** tab.
10. In the **Search Address Book** screen, pick **Interpreter** from the **Type** drop-down list and search for the new contact.
11. Select the new contact found by **Search** and verify that the entry is correct and that you can see the **Interpreter Specialty** field.
12. Return to the claim and click **Actions** > **New** > **Service**.
13. For **Request Type**, choose **Perform Service**.
14. For **Vendor Name**, click the contact picker and choose **New Vendor** > **Interpreter** from the drop-down list.

    The *contact picker* is a second drop-down button to the right of the field:

    
15. Enter a name, an address, an interpreter specialty, and a tax ID, and then click **OK**.
16. On the **Create Service Requests** screen under **Services to Perform**, click **Add** to specify a service to perform.

    There is no Interpreter service in the base configuration. Just pick any service on the list.
17. At the bottom of the screen, choose a **Service Address**, and then click **Submit**.
18. In the Sidebar, click **Parties Involved**.
    The **Contacts** screen opens.
19. Verify that your new contacts have been added. Click one of the contacts to open the **Basics** card and verify that **Interpreter Specialty** is listed under **Additional Info**.
20. Log in to ContactManager as a user who can create new contacts.
    For example, log in as user `aapplegate` with password `gw`.
21. In the Sidebar, choose **Search**, pick **Interpreter** as the **Contact Type**, and search for one of the interpreter contacts you created in ClaimCenter.
22. Click the contact's name to see the details screen for the contact.
23. On the **Basics** tab, click **Edit** and change the value of **Interpreter Specialty**, and then click **Update** to save the change.
24. In the Sidebar, choose **Actions** > **New Person** > **Vendor** > **Interpreter** to see the **New Interpreter** screen.
25. Verify that **Interpreter Specialty** is listed in the **Additional Info** section.
26. Enter enough information to create an `Interpreter` contact. For **Tags**, click **Vendor**. Click **Update** to save the new contact.
27. Return to ClaimCenter.
28. Click the **Address Book** tab.
29. On the **Search Address Book** screen, pick **Interpreter** from the drop-down list and search for the new contact you created in ContactManager.

30. Click the contact found by **Search** and verify that the entry is correct.
31. Search for the interpreter that you originally created in ClaimCenter and then edited in ContactManager.
32. Click that contact and verify that the changes you made in ContactManager are in this contact's data.

Extending contacts with an array

You can extend the `ABContact` entity in ContactManager and the `Contact` entity in ClaimCenter with an array. In this example, a multi-step process divided into a series of topics, the array is composed of states that comprise the service area for the contact. After creating and extending entities, you update the necessary files and screens to make the new array usable across the applications.

Create a service state entity in ContactManager

Part of extending contacts with an array is to create the entity that will be in the array.

About this task

This step is the first in the example described at "Extending contacts with an array" on page 138. In this step, you add an entity to ContactManager representing a state in which a contact provides service.

Procedure

1. Start Guidewire Studio™ for ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Entity**, and then right-click **Entity** and choose **New** > **Entity**.
3. Enter the following file name: `ContactServiceState`
4. Enter the following for **Desc**:

   `Represents a state where the contact provides services`
5. In addition to **Extendable** and **Final**, which are already selected, select the **Exportable** check box.
6. Click **OK**.
7. In the editor, click **entity** at the top of the **Element** hierarchy
8. Click the drop-down list for ✚ and choose **implementsEntity**, and then choose **ABLinkable** from the drop-down list for the **name** value.
9. Click the drop-down list for ✚ and choose **implementsEntity**, and then choose **Extractable** from the drop-down list for the **name** value.

   Choosing **Extractable** makes the entity part of the `ABContact` entity graph. If you implement personal data destruction, being part of the entity graph makes it possible to destroy the entity if necessary. See "Extensions of ABContact and other entities and the domain graph" on page 228.
10. Click the drop-down list for ✚ and choose **implementsEntity**, and then choose **Obfuscatable** from the drop-down list for the **name** value.

    Choosing **Obfuscatable** makes it possible to obfuscate the entity as part of personal data destruction. See "Implementing the Obfuscatable delegate in an entity" on page 233.
11. Click the drop-down list for ✚ and choose **implementsInterface**.

    Enter the following values:

    - For **iface**, enter `gw.api.obfuscation.Obfuscator`.

- For **impl**, enter `gw.personaldata.obfuscation.DefaultPersonalDataObfuscator`.

This step specifies how fields of the entity will be obfuscated. See:

- “Implementing the Obfuscator interface in an entity” on page 233
- “Personal data obfuscation class hierarchy” on page 234

12. Click **entity** at the top of the **Element** hierarchy.
13. Click the drop-down list for ✚ and choose **foreignKey**, and then enter the following values:

Name	Value
name	`Contact`
fkentity	`ABContact`
nullok	`false`
columnName	`ContactID`
desc	`Contact that this Service State row relates to`

14. Click **entity** at the top of the **Element** hierarchy.
15. Click the drop-down list for ✚ and choose **typekey**, and then enter the following values:

Name	Value
name	`ServiceState`
typelist	`State`
nullok	`false`
desc	`State serviced by the contact`
exportable	`true` (default value)
loadable	`true` (default value)

16. Click **entity** at the top of the **Element** hierarchy.
17. Click the drop-down list next to ✚ and choose **index**.
18. Enter the following values for the new index:

Name	Value
name	`absrvstatelinkid`
desc	`Preserve uniqueness of LinkID`
unique	`true`

19. Right-click **index** in the **Element** column, and choose **Add new** > **indexcol**.

    This index column is the first of two to add to the index.

20. Enter the following values for the new `indexcol`:

Name	Value
name	`LinkID`
keyposition	`1`

21. Right-click **index** in the **Element** column, and choose **Add new** > **indexcol**.
22. Enter the following values for the new `indexcol`:

Name	Value
name	`Retired`

Name	Value
keyposition	`2`

23. Click **entity** at the top of the **Element** hierarchy.
24. Click the drop-down list next to ✚ and choose **index**.
25. Enter the following values for the new index:

Name	Value
name	`ind1`
unique	`true`

26. Right-click the ind1 **index** in the **Element** column, and choose **Add new** > **indexcol**.

    This index column is the first of three to add to the index.
27. Enter the following values for the new `indexcol`:

Name	Value
name	`ServiceState`
keyposition	`1`

28. Right-click the ind1 **index** in the **Element** column, and choose **Add new** > **indexcol**.
29. Enter the following values for the new `indexcol`:

Name	Value
name	`Retired`
keyposition	`2`

30. Right-click the ind1 **index** in the **Element** column, and choose **Add new** > **indexcol**.
31. Enter the following values for the new `indexcol`:

Name	Value
name	`ContactID`
keyposition	`3`

32. Click **entity** at the top of the **Element** hierarchy.
33. Click the drop-down list next to ✚ and choose **index**.
34. Enter the following values for the new index:

Name	Value
name	`ind2`
unique	`true`

35. Right-click the ind2 **index** in the **Element** column, and choose **Add new** > **indexcol**.

    This index column is the first of three to add to the index.
36. Enter the following values for the new `indexcol`:

Name	Value
name	`ContactID`
keyposition	`1`

37. Right-click the ind2 **index** in the **Element** column, and choose **Add new** > **indexcol**.
38. Enter the following values for the new `indexcol`:

Name	Value
name	`Retired`
keyposition	`2`

39. Right-click the `ind2` **index** in the **Element** column, and choose **Add new** > **indexcol**.
40. Enter the following values for the new `indexcol`:

Name	Value
name	`ServiceState`
keyposition	`3`

What to do next

"Add the new array to the ABContact entity" on page 141

Add the new array to the ABContact entity

Part of extending contacts with an array is to extend the `ABContact` entity in ContactManager to enable it to use an array of service state entities.

Before you begin

Complete "Create a service state entity in ContactManager" on page 138.

About this task

The array has the `triggersValidation` attribute set to `true` to enable triggering of validation rules when an element of the array changes.

Procedure

1. In Guidewire Studio™ for ContactManager, navigate in the **Project** window to **configuration** > **config** > **Extensions** > **Entity**, and then double-click `ABContact.etx` to open this file in the Entity editor.
2. In the editor, click **entity (extension)** at the top of the **Element** hierarchy
3. Click the drop-down list for ✚ and choose **array**, and then enter the following values:

Name	Value
name	`ContactServiceArea`
arrayentity	`ContactServiceState`
arrayfield	`Contact`
desc	`Geographical area where the contact provides service`
triggersValidation	`true`

4. If necessary, stop ContactManager.

   Open a command prompt in the ContactManager installation folder and then enter the following command:

```
gwb stopServer
```

5. Regenerate the *Data Dictionary* to ensure that the data model updates are correct.

   At a command prompt, navigate to the ContactManager installation folder and enter:

```
gwb genDataDictionary
```

What to do next

"Map the entity and array extensions in ContactManager" on page 142

Map the entity and array extensions in ContactManager

After creating new `ABContact` entities and array extensions, you need to map them from ContactManager to any core applications that support them.

Before you begin

Complete "Add the new array to the ABContact entity" on page 141.

About this task

In this topic, you add the new array reference and the entity it references to the `ContactMapper` class in ContactManager. For more information on this class, see "ContactMapper class" on page 313.

Procedure

1. In Guidewire Studio™ for ContactManager, press `Ctrl+N` and search for the `ContactMapper` class.
2. Double-click the `ab1000` version of the class in the search results to open it in the editor.
3. Find the `Mappings` method, which has the following declaration:

```
override property get Mappings() : Set<PropertyMapping>
```

4. At the end of the method, add a comma to the last line of code so you can add more code. For example:

```
fieldMapping(ABContactCategoryScore#Score),
```

5. After this last line of code, add an `arrayMapping` method for `ABContact#ContactServiceArea`, preceded with a comment for this part of the `Mappings` method:

```
//ContactServiceState mapping
arrayMapping(ABContact#ContactServiceArea),
```

6. Add `fieldMapping` methods for the elements of the `ContactServiceState` entity, which the `ABContact` array reference refers to:

```
fieldMapping(ContactServiceState#LinkID).withMappingDirection(TO_XML),
fieldMapping(ContactServiceState#External_PublicID),
fieldMapping(ContactServiceState#External_UniqueID),
fieldMapping(ContactServiceState#ServiceState)
```

What to do next

"Add support for service states to the ContactManager user interface" on page 142

Add support for service states to the ContactManager user interface

Enable ContactManager users to work with the `ServiceState` property when editing and viewing contacts.

Before you begin

Complete "Map the entity and array extensions in ContactManager" on page 142.

Procedure

1. Open Guidewire Studio™ for ContactManager.
2. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.

3. Press `Ctrl+F` and search for the entry `Web.ContactDetail.RetiredMessage` in the file.
4. Add a new line below that entry, and then enter the following display keys:

```
Web.ContactDetail.ServiceStateName = State Name
Web.ContactDetail.ServiceStates = Service States
```

5. In the **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **contacts** > **basics**, and then double-click `ContactBasicsDV.ABCompany` to edit this PCF file.
6. On the right, at the bottom of the input column containing the `TextAreaInput` called Notes, drag a `ListViewInput` from the **Toolbox** and drop it above Notes.
7. Click the new `ListViewInput` and set the following properties:

label	`displaykey.Web.ContactDetail.ServiceStates`
labelAbove	`true`
boldLabel	`true`

8. From the **Toolbox**, drag a `RowIterator` and drop it on the new list view panel, and then click it and set the following properties:

editable	`true`
elementName	`currentServiceState`
toAdd	`contact.addToContactServiceArea(currentServiceState)`
toRemove	`contact.removeFromContactServiceArea(currentServiceState)`
value	`contact.ContactServiceArea`
canPick	`true`

The new list view input remains red until you add the `Row` widget in the next step.

9. From the **Toolbox**, drag a `Row` and drop it on the `RowIterator`, and then drag a `Cell`, and drop it on the `Row`.
10. Click the cell and set the following cell properties:

editable	`true`
id	`ServiceState`
label	`displaykey.Web.ContactDetail.ServiceStateName`
value	`currentServiceState.ServiceState`
unique	`true`

11. When you dropped the `RowIterator`, the PCF editor created a `ListViewPanel` as a container for it. Click the `ListViewPanel` and set its **id** property to `CurrentServiceStateLV`.
12. From the **Toolbox** on the right, drag a `Toolbar` and drop it above the `ListViewPanel` named CurrentServiceStateLV.
13. From the **Toolbox**, drag an `IteratorButtons` widget and drop in on the toolbar so you can add and remove states.
14. Click the new `IteratorButtons` widget and set the property **iterator** to `CurrentServiceStateLV`.

What to do next

"Create a service state entity in ClaimCenter" on page 144

Create a service state entity in ClaimCenter

Part of extending contacts with an array is to create the ClaimCenter entity that will be in the array.

Before you begin

Complete “Add support for service states to the ContactManager user interface” on page 142.

About this task

Create an entity that is the ClaimCenter counterpart of the ContactManager `ContactServiceState` entity.

Procedure

1. If necessary, start Guidewire Studio™ for ClaimCenter.

   At a command prompt, navigate to the ClaimCenter installation folder and enter the following command:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Entity**.
3. Right-click **Entity** and choose **New** > **Entity**.
4. Enter the following file name: `ContactServiceState`
5. Enter the following for **Desc**:

   `Represents a state where the contact provides services`

6. Select **Extendable** and **Final**.
7. Click **OK**.
8. In the editor, click **entity** at the top of the **Element** hierarchy
9. Click the drop-down list for ✚ and choose **implementsEntity**, and then choose **Extractable** from the drop-down list for the **name** value.
10. Repeat the two previous steps, but use them to create **implementsEntity** elements named **OverlapTable** and **AddressBookLinkable**.
11. Click **entity** at the top of the **Element** hierarchy.
12. Click the drop-down list next to ✚ and choose **foreignKey**, and then enter the following values:

Name	Value
name	`Contact`
fkentity	`Contact`
nullok	`false`
columnName	`ContactID`
desc	`Contact that this Service State row relates to`

13. Click **entity** at the top of the **Element** hierarchy.
14. Click the drop-down list next to ✚ and choose **typekey**, and then enter the following values:

Name	Value
name	`ServiceState`
nullok	`false`
typelist	`State`
desc	`State serviced by the contact`

Name	Value
loadable	`true` (default value)

15. Click **entity** at the top of the **Element** hierarchy, and then click the drop-down list next to ➕ and choose **index**.
16. Enter the following values for the new index:

Name	Value
name	`ind1`
unique	`true`

17. Right-click the ind1 **index** in the **Element** column, and choose **Add new** > **indexcol**.

    This index column is the first of three to add to the index.
18. Enter the following values for the new `indexcol`:

Name	Value
name	`ServiceState`
keyposition	`1`

19. Right-click the ind1 **index** in the **Element** column, and choose **Add new** > **indexcol**.
20. Enter the following values for the new `indexcol`:

Name	Value
name	`Retired`
keyposition	`2`

21. Right-click the ind1 **index** in the **Element** column, and choose **Add new** > **indexcol**.
22. Enter the following values for the new `indexcol`:

Name	Value
name	`ContactID`
keyposition	`3`

23. Click **entity** at the top of the **Element** hierarchy.
24. Click the drop-down list next to ➕ and choose **index**.
25. Enter the following values for the new index:

Name	Value
name	`ind2`
unique	`true`

26. Right-click the ind2 **index** in the **Element** column, and choose **Add new** > **indexcol**.

    This index column is the first of three to add to the index.
27. Enter the following values for the new `indexcol`:

Name	Value
name	`ContactID`
keyposition	`1`

28. Right-click the ind2 **index** in the **Element** column, and choose **Add new** > **indexcol**.
29. Enter the following values for the new `indexcol`:

Name	Value
name	`Retired`
keyposition	2

30. Right-click the ind2 **index** in the **Element** column, and choose **Add new** > **indexcol**.
31. Enter the following values for the new `indexcol`:

Name	Value
name	`ServiceState`
keyposition	3

What to do next

"Add the new array to the Contact entity in ClaimCenter " on page 146

Add the new array to the Contact entity in ClaimCenter

Extend the `Contact` entity in ClaimCenter to enable it to use an array of service state entities.

Before you begin

Complete "Create a service state entity in ClaimCenter" on page 144.

About this task

The array of service state entities has the `triggersValidation` attribute set to `true` to enable validation rules to be triggered when an element of the array changes. Additionally, to simplify communications between ContactManager and ClaimCenter, the array has the same name as the array added to ContactManager.

Procedure

1. In Guidewire Studio™ for ClaimCenter, navigate in the **Project** window to **configuration** > **config** > **Extensions** > **Entity**, and then double-click `Contact.etx` to open it in the editor.
2. In the editor, click **extension** at the top of the **Element** hierarchy
3. Click the drop-down list next to ✚ and choose **array**, and then enter the following values:

Name	Value
name	`ContactServiceArea`
arrayentity	`ContactServiceState`
arrayfield	`Contact`
desc	`Geographical area where the contact provides service`
triggersValidation	`true`

4. If necessary, stop ClaimCenter.

   Open a command prompt in the ClaimCenter installation folder and then enter the following command:

```
gwb stopServer
```

5. Regenerate the *Data Dictionary* to ensure that the data model updates are correct.

   At a command prompt, navigate to the ClaimCenter installation folder and enter:

```
gwb genDataDictionary
```

What to do next

"Map the new entity and array extensions in ClaimCenter" on page 147

Map the new entity and array extensions in ClaimCenter

After creating new `Contact` entities and array extensions, you need to map them from ClaimCenter to ContactManager.

Before you begin

Complete "Add the new array to the Contact entity in ClaimCenter " on page 146.

About this task

In this step, you add the new array reference and the entity it references to the `ContactMapper` class in ClaimCenter.

For more information on this class, see "ContactMapper class" on page 313. Additionally, for more information on the last step in this topic, see "ContactManager link IDs and comparison to other IDs" on page 299.

Procedure

1. In Guidewire Studio™ for ClaimCenter, press `Ctrl+N` and search for the class `ContactMapper`, and then double-click the `ab1000` version of this class in the search results to open it in the editor.
2. Find the `Mappings` method, which has the following declaration:

   ```
   override property get Mappings() : Set<CCPropertyMapping>
   ```

   In this method, you will add a method for `Contact` array reference `ContactServiceArea` and methods that map the elements of the array, `ContactServiceState`.
3. At the end of the method, add a comma to the last line of code so you can add more code. For example:

   ```
   .withEntityBeanBlock( \ lm, bp -> populateBeanFromXml(bp)),
   ```

4. After this last line of code, add an `arrayMapping` method for `Contact#ContactServiceArea`, preceded with a comment for this part of the `Mappings` method:

   ```
   //ContactServiceState mapping
   arrayMapping(Contact#ContactServiceArea),
   ```

5. Add `fieldMapping` methods for the elements of the `ContactServiceState` entity, which the `ABContact` array reference refers to:

   ```
   fieldMapping(ContactServiceState#AddressBookUID)
     .withABName(MappingConstants.LINK_ID),
   fieldMapping(ContactServiceState#PublicID)
     .withMappingDirection(TO_XML)
     .withABName(MappingConstants.EXTERNAL_PUBLIC_ID),
   fieldMapping(ContactServiceState#ServiceState)
   ```

   This code maps between the following `ContactServiceState` fields in ClaimCenter and ContactManager.

ClaimCenter	ContactManager	Direction
`AddressBookUID`	`LINK_ID`	Both ways
`PublicID`	`EXTERNAL_PUBLIC_ID`	From ClaimCenter to ContactManager only
`ServiceState`	`ServiceState`	Both ways

What to do next

"Add support for service state searches to the Address Book" on page 148

Add support for service state searches to the Address Book

Enable ClaimCenter users to use the Address Book to search for contacts in ContactManager by `ServiceState`.

Before you begin

Complete “Map the new entity and array extensions in ClaimCenter” on page 147.

Procedure

1. Open Guidewire Studio™ for ClaimCenter.
2. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
3. Press `Ctrl+N` and search for `Web.ContactDetail.RetiredMessage`.
4. Insert a new line below that one and add the following entries:

```
Web.ContactDetail.ServiceStateName = State Name
Web.ContactDetail.ServiceStates = Service States
```

5. In the **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **addressbook**, and then right-click **addressbook** and click **New** > **PCF File**.
6. For the **File name**, enter `AddressBookServiceStates`.
7. For **File type,** click **List View**, and then click **OK** to edit this PCF file.
8. Click the new `ListViewPanel` to open its **Properties** window, and then click the **Required Variables** tab.
9. Click the green plus icon, ➕, and add the following new required variable:

name	`contact`
type	`Contact`

10. Click the **Exposes** tab, and then click the green plus icon, ➕, and choose **ExposeIterator**.
11. Enter the following values:

valueType	`entity.ContactServiceState`
widget	`AddressBookServiceStatesLV`

12. From the **Toolbox** on the right, drag a `RowIterator` and drop in on the list view.
13. Select the `RowIterator` widget and set the following properties:

editable	`true`
elementName	`currentServiceState`
toAdd	`contact.addToContactServiceArea(currentServiceState)`
toRemove	`contact.removeFromContactServiceArea(currentServiceState)`
value	`contact.ContactServiceArea`
canPick	`true`

14. From the **Toolbox** on the right, drag a a new `Row` and drop it on the `RowIterator`.
15. From the **Toolbox** on the right, drag a new `Cell` and drop it on the `Row`, and then set the following cell properties:

editable	`true`
id	`ServiceState`
label	`displaykey.Web.ContactDetail.ServiceStateName`
value	`currentServiceState.ServiceState`

unique	`true`

16. If the outline of the `ListViewPanel` remains red, it is likely that the editor has lost the value type of the **ExposeIterator** named `AddressBookServiceStatesLV` in the **Exposes** tab. To see if that is the case, click the `ListViewPanel` to open its **Properties** window. Then, as described in “step 9”, click the **Exposes** tab, and then click `AddressBookServiceStatesLV` and, if necessary, reenter the following **valueType**:

    ```
    entity.ContactServiceState
    ```

17. In the **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **addressbook**, and then double-click `AddressBookContactBasicsDV.Company` to edit this PCF file.
18. On the right, in the `InputColumn` with the `TextAreaInput` called Notes at its bottom, drag a `ListViewInput` widget and drop it above Notes. Then enter the following attributes:

def	`AddressBookServiceStatesLV(contact)`
label	`displaykey.Web.ContactDetail.ServiceStates`
labelAbove	`true`
boldLabel	`true`

    The new list view input remains red until you add the **Toolbar** widget in the next step.

19. From the **Toolbox** on the right, drag a `Toolbar` and drop it in the new `ListViewInput` above the `ListViewPanel`.
20. From the **Toolbox** on the right, drag an `IteratorButtons` widget and drop it on the toolbar so you can add and remove states.
21. Click the new `IteratorButtons` widget and set the **iterator** property value to `AddressBookServiceStatesLV.AddressBookServiceStatesLV`.

What to do next

“Enable addition of service states to a company on a claim” on page 149

Enable addition of service states to a company on a claim

As part of extending contacts with an array, extend the ClaimCenter user interface to enable users to add service states to a company that is associated with a claim.

Before you begin

Complete “Add support for service state searches to the Address Book” on page 148.

Procedure

1. In Guidewire Studio™ for ClaimCenter, navigate in the **Project** window to **configuration** > **config** > **Page Configuration** > **pcf** > **shared** > **contacts**.
2. Right-click **contacts** and click **New** > **PCF File**.
3. For the **File name**, enter `VendorServiceStates`, and for **File type** choose **List View**, and then and click **OK** to edit this PCF file.
4. Click the new list view panel to open its **Properties** window, then click the **Required Variables** tab.
5. Click ✚ and add the following new required variable:

name	`contactHandle`
type	`contact.ContactHandle`

6. Click the **Code** tab and enter the following code:

    ```
    property get contact() : Contact { return contactHandle.Contact }
    ```

7. Click the **Exposes** panel, and then click ✚ and click **ExposeIterator**.
8. Enter the following values:

valueType	`entity.ContactServiceState`
widget	`VendorServiceStatesLV`

9. From the **Toolbox** on the right, drag a `RowIterator` and drop it on the new list view panel, and then set the following properties:

editable	`true`
elementName	`currentServiceState`
toAdd	`contact.addToContactServiceArea(currentServiceState)`
toRemove	`contact.removeFromContactServiceArea(currentServiceState)`
value	`contact.ContactServiceArea`
canPick	`true`

10. From the **Toolbox**, drag a new `Row` and drop it on the `RowIterator`.
11. From the **Toolbox**, drag a new `Cell`, drop it on the `Row`, and then set the following cell properties:

editable	`true`
id	`ServiceState`
label	`displaykey.Web.ContactDetail.ServiceStateName`
value	`currentServiceState.ServiceState`
unique	`true`

12. If the outline of the `ListViewPanel` remains red, it is likely that the editor has lost the `valueType` of the `ExposeIterator` in the **Exposes** tab, `VendorServiceStatesLV`. To see if that is the case, click the `ListViewPanel` to open its **Properties** window. Then, click the **Exposes** tab, click `VendorServiceStatesLV`, and, if necessary, reenter the following **valueType**:

    ```
    entity.ContactServiceState
    ```
13. In the **Project** window, navigate to **configuration** > **config** > **Page Configuration** > **pcf** > **shared** > **contacts**, and then double-click `ContactBasicsDV.Company` to edit this PCF file.
14. On the right, in the input column containing the `TextAreaInput` called `Notes`, drag a `ListViewInput` from the **Toolbox** and drop it above `Notes`.
15. Click the new `ListViewInput` widget and set the following properties:

def	`VendorServiceStatesLV(contactHandle)`
label	`displaykey.Web.ContactDetail.ServiceStates`
labelAbove	`true`
boldLabel	`true`

    The new list view input remains red until you add the `Toolbar` widget in the next step.
16. From the **Toolbox** on the right, drag a `Toolbar` and drop it in the `ListViewInput`.
17. From the **Toolbox** on the right, drag an `IteratorButtons` widget and drop it on the toolbar so you can add and remove states.
18. Click the new `IteratorButtons` widget and set the following property:

iterator	`VendorServiceStatesLV.VendorServiceStatesLV`

What to do next

"Test service state entity and array extensions" on page 151

Test service state entity and array extensions

After extending contacts with an array and configuring the ContactManager and ClaimCenter user interfaces, test that the extensions work in both applications.

Before you begin

Complete "Enable addition of service states to a company on a claim" on page 149.

Procedure

1. Shut down ClaimCenter and ContactManager if they are running, and then restart both applications.
2. Log in to ClaimCenter as a user who can create new contacts.
   For example, if you have loaded sample data, log in as user `ssmith` with password `gw`.
3. Click the **Claim** tab and open an existing claim.
4. Click **Parties Involved** in the Sidebar, and then on the **Contacts** screen click **New Contact** > **Company**.
5. On the **New Company** screen, **Service States** is listed above the **Notes** section on the right.
6. Click **Add** to add a row for a state, and then click the new **State Name** field and select a state from the list.
7. Select the new row, and then click **Remove** to delete it.
8. Add several states, and then enter enough information to create the new company.
9. Under **Roles** near the top of the screen, click **Add**, and then add a role for this company for the current claim, such as **Vendor**.
10. Click **Update** to add the company to the list on the **Contacts** screen.
11. Click the company in the list on the **Contacts** screen.

    On the **Basics** card, you see the following message:

    ```
    This contact is linked to the Address Book and is in sync
    ```

    Because you logged in as a user with permission to create and edit contacts, ClaimCenter saved the new contact in ContactManager.
12. Click the **Address Book** tab, and then search for the company you just created.
13. Click the company name in the search results to open its **Basics** card.
14. Look for the **Service States** list view above the **Notes**, and see that they are the states you added.
15. Click the **Edit in ContactManager** button, and, if necessary, log in as a user who can create new contacts.
    For example, log in as user `aapplegate` with password `gw`.
    ContactManager opens showing the company you created in ClaimCenter.
16. In ContactManager, verify that there is a **Service States** list above **Notes**, and that the states are the same as the ones you added in ClaimCenter.
17. Click **Edit** and delete one of the states.
18. Add a different state, and then click **Update**.
19. Click **Actions** > **New Company** > **Vendor** and choose each kind of company in turn to verify that there is a **Service States** list for each type of company.
20. Choose a type of company to create, then add and delete a service state. Add several service states, and then enter enough information to create the new `Company`.
21. Click **Update** to save the company.

22. Go back to ClaimCenter and click the **Address Book** tab, and then search for the company you created in ContactManager to verify that ClaimCenter can find the company.
23. In the claim in which you created the company with states, click **Parties Involved**.
24. On the **Contacts** screen, click the company you created and verify that it is now out of sync.
25. Click **Copy from Address Book** and verify that the company now has the set of states that you set up in ContactManager.

Changing the subtype of a Contact instance

You can change the subtype of a contact instance by using a command-prompt utility.

Important considerations before changing a contact subtype

This feature must be used with caution by experienced database and configuration professionals who are familiar with the `Contact` data model. It applies only to ContactManager and ClaimCenter. To use this feature, you must log out of the application and set the application server run level to `maintenance` during the contact subtype change.

The change of subtype directly affects the database. It causes ContactManager to be unable to synchronize the contact with ClaimCenter until you make the same subtype change in both ClaimCenter and ContactManager. Additionally, fields can be deleted and array fields can cause user interface exceptions.

For example:

- A subtype change can delete fields that do not exist in the new subtype. If feasible, determine which fields will change and where you want to move data in the new subtype. Then make any data updates you can in ContactManager before changing the subtype. You might also want to save the original contact data until you can review the contact after the subtype change.
- Related contacts and other arrays are not deleted when you change a subtype. However, they can cause problems in the user interface.
- In ContactManager, you can delete problematic data for the contact, such as related contacts that are not compatible with the new subtype, before making the subtype change.

See also

- “Contact subtype field differences and changing subtypes” on page 154

Overview

In ClaimCenter and ContactManager, you can change the subtype of a contact that was created with the wrong subtype without having to delete and re-create the contact. For example, you created a contact with subtype `Doctor` when you intended the contact to be subtype `Attorney`. If it is possible to delete the contact and re-create it with the correct subtype, do so. However, deleting and re-creating the contact might not be practical if the contact has history, such as being the payee on a check.

Changing subtype with a command-prompt utility

Changing the contact subtype is available through a command-prompt utility available in ContactManager and in ClaimCenter:

```
maintenance_tools
                  -user user-name
                  -password user-password
                  -changesubtypetargettype subtype
                  -changesubtypepublicid publicID
```

Note: Do not copy this command and paste it directly into the command line. You must first get rid of the linefeed characters that were inserted in this example for legibility.

You run this command from a command prompt open in the admin/bin folder of the installed product.

This command takes the following parameters:

`user-name`

User name of a user who has the role Contact Subtype Changer. This role includes the permissions required to run this command, `changecontactsubtype` and `soapadmin`. If you do not specify the `-user` parameter, the command defaults to Super User and requires that user's password.

`user-password`

Password for the user specified in the `-user` parameter, or for the Super User if there is no user specified.

`subtype`

New `Contact` or `ABContact` subtype for the contact, depending on whether you are running the command in ClaimCenter or ContactManager.

`publicID`

Value of the `PublicID` property of the contact. The value of `PublicID` is not necessarily the same for a contact stored in ContactManager and the equivalent contact instance or instances stored in ClaimCenter. Additionally, the value of `PublicID` is not necessarily the same as that of the `LinkID` in ContactManager or the `AddressBookUID` in ClaimCenter.

See also

- "Restrictions on changing the subtype of a Contact instance" on page 153
- "ContactManager link IDs and comparison to other IDs" on page 299
- "Troubleshooting the change subtype command-prompt utility" on page 157

Restrictions on changing the subtype of a Contact instance

There is information you must consider before changing the subtype of a `Contact` instance, such as:

- Which applications store the contact
- Restrictions on client contact subtype
- Differences in fields supported by various contact subtypes

Contact instance location and changing subtype

The instance of a contact for which you want to change the subtype can be stored either in ClaimCenter or in ContactManager, or in both. In ClaimCenter, the contact would be on at least one claim's **Parties Involved** > **Contacts** page. If the ClaimCenter contact is linked to ContactManager, you must make the subtype change separately in each application.

There can be multiple instances of the contact in ClaimCenter, each stored as a claim contact with a claim. If contacts are not linked to ContactManager, you can change any number of them, based on the reason for the subtype change.

If the contacts are linked with ContactManager, you need to change all the linked contact instances in ClaimCenter and the single instance in ContactManager. One way to find the ClaimCenter instances is to first get the value of the `LinkID` of the contact in ContactManager. Then use that value in a query that uses the equivalent ClaimCenter contact property, `AddressBookUID`, to find all the contact instances in ClaimCenter. The value that you need from each contact to make the subtype change is the `PublicID`. You then set the ClaimCenter run level to `maintenance` and run the command-prompt tool for each instance of the contact, specifying the value of its `PublicID`.

To change a contact subtype in ContactManager, set the run level to `maintenance` and then run the command-prompt tool once for the contact instance, using its `PublicID` to change its subtype. If the contact is linked to ClaimCenter, you set the run levels for both applications to `maintenance` and perform the subtype changes at the same time.

Subtype changes for client contacts

If the contact has the Client tag, there are restrictions on which subtypes you can change it to. Contacts with the Client tag cannot have their subtypes changed across the three main contact subtypes, `Person`, `Company`, and `Place`. This restriction is in place because neither PolicyCenter nor BillingCenter can handle such changes.

For example, if you try to make a subtype change in ContactManager for a client contact of type `ABDoctor` to the subtype `ABMedicalCareOrg`, the command fails with an error message. Because `ABDoctor` is an `ABPerson` subtype, the client contact's subtype cannot be changed to `ABMedicalCareOrg`, which is an `ABCompany` subtype.

If your subtype change stays within a single subtype hierarchy, such as `ABPerson` in ContactManager and `Person` in ClaimCenter, you are allowed to make the change. The same is true for the `ABCompany` and `Company` subtype hierarchies and the `ABPlace` and `Place` subtype hierarchies in ContactManager and ClaimCenter.

PolicyCenter and BillingCenter work only with `Company` and `Person` subtypes. Subtypes of `Person` all look the same to these applications, as do subtypes of `Company`. For example, a subtype change from a client contact of type `ABDoctor` to the subtype `ABAttorney` is permitted, because PolicyCenter and BillingCenter work with the contact as a `Person` subtype.

See also

- For information on contact subtypes in the base configuration, see "ContactManager and core application contact entity hierarchies" on page 113.

Contact subtype field differences and changing subtypes

Changing the subtype of a contact instance can cause some predictable changes in contact data. This topic describes some common changes. Before making the subtype change, you can save the current data for the contact, and then add data to the new subtype if that subtype supports the data. After the subtype change, you can see which fields were changed in the Notes field for the contact.

Name field changes between a company or place and a person

A `Person` subtype in the `en_US` locale can have first name, last name, prefix, and suffix fields, while a `Company` or `Place` subtype has only a name field. If you change the subtype from a person subtype to a company or place subtype, the person's first name, last name, prefix, and suffix are combined into a single name. They become the name of the company or place subtype. If you make the opposite subtype change, the name of the company or place subtype becomes the last name of the person subtype, with no first name.

> **Note:** The set of name fields for a `Person` subtype can vary by locale. However, for those locales, the conversion works the same way between the multiple name fields of a `Person` subtype and the single name field of a `Company` subtype.

For example, you change the subtype of an instance of `Doctor` with prefix Dr, first name Samantha, and last name Andrews to `MedicalCareOrg`. The `Name` field of the `MedicalCareOrg` contact instance becomes Dr Samantha Andrews.

Subtype changes and incompatible data

If there are fields on one subtype that are not on the new one, the fields are dropped or converted to similar fields as part of the subtype change. However, array fields are not dropped, and if they are not compatible with the new subtype, that can cause problems in the user interface.

For example, a contact can have related contacts, such as a company that has employees, or a person who has an employer. However, in the base configuration, a company cannot have an employer, and a person cannot have employees. If you change a person who has an employer to a company subtype, the employer field is preserved. However, the next time you edit the contact, you see an exception saying that a company cannot have an employer. If you go to the **Related Contacts** card and delete that relationship, you can then save the contact.

Additionally, if the person had the primary phone defined as a cell phone, that field is deleted during the transfer. In the base configuration, the primary contact number for a company contact cannot be a cell phone.

You can compare the fields for the subtypes in the *Data Dictionary*. If there are fields that will be dropped, you can record the data. After you change the subtype, you can add data that is appropriate for the new subtype. You can also consider deleting fields that will cause problems before making the contact subtype change.

See also

- For information on relationships between contacts and for an example of how to add bidirectional contact relationship, see the *Configuration Guide*

Changing the subtype of a contact instance

Be sure to read the important note at the beginning of the main topic and the restrictions on changing subtypes before starting this multi-step, multi-topic process. In particular, if the contact has the Client tag, you cannot change the subtype between `Person`, `Company`, or `Place` subtypes. For example, you cannot change the subtype of a Client contact from `Doctor`, a `Person` subtype, to `MedicalCareOrg`, a `Company` subtype. The tool will prevent you from making this change.

- For an Important note, see “Changing the subtype of a Contact instance” on page 152.
- For information you need to know before you start, see “Restrictions on changing the subtype of a Contact instance” on page 153

If there is data in the contact that will be lost or will not be compatible with the new subtype, if possible, fix the data before you change the subtype. If the contact is stored in ClaimCenter and is linked to ContactManager, first make the data changes to the contact in ContactManager. The changes are then propagated to ClaimCenter, which updates all linked instances of the contact. After you complete your data changes, you can proceed with the contact subtype change.

> **Note:** It is not always possible to make all the data changes before the subtype change, so you might have to make some data changes afterwards.

You can change the subtype of a contact stored only in ClaimCenter, stored only in ContactManager, or stored in both and linked together. The server runlevel must be set to `maintenance` to make the change. If the contact is stored in both ClaimCenter and ContactManager, set both servers’ runlevels to `maintenance`. Leave them at maintenance runlevel until you complete the subtype changes and any subsequent edit verifications in both applications.

Change the subtype of an ABContact instance in ContactManager

Before you begin

There are limitations on changing the subtype of a contact instance that you must be aware of before starting this process. See “Changing the subtype of a contact instance” on page 155.

Procedure

1. Run a query on the ContactManager database to get the `PublicID` and `LinkID` of the contact instance.
2. Ensure that all users are logged out.

   Note: This step is an important one. ContactManager must not have its user interface open when you apply the subtype change.
3. Set the ContactManager server’s runlevel to `maintenance`.
   For example, at a command prompt open in `ContactManager/admin/bin`, enter:

   `system_tools -password` *`adminuser-password`* `-maintenance`
4. Run the command-prompt tool to change the subtype of the contact instance.
   For example, the user with login alinu has the role Contact Subtype Changer and password x2wTz@71P. For user alinu to change the contact with `PublicID` ab:123 to the subtype `ABDoctor`, at a command line open in `ContactManager/admin/bin`, the user would enter the following command, all on one line:

```
maintenance_tools -user alinu
                  -password x2wTz@71P
                  -changesubtypetargettype ABDoctor
                  -changesubtypepublicid ab:123
```

5. Note the messages in the console.

   If your command is successful, you see messages similar to the following:

```
Changing contact with publicID: ab:123 to subtype: ABDoctor
done
```

6. Log in to ContactManager and ensure that you can open the contact and save changes. If necessary, update data as needed.
7. Take ContactManager out of `maintenance` runlevel.

   For example, at a command prompt open in `ContactManager/admin/bin`, enter:

   ```
   system_tools -password adminuser-password -maintenance
   ```

What to do next

"Change the subtype of a Contact instance in ClaimCenter" on page 156

Change the subtype of a Contact instance in ClaimCenter

Before you begin

Complete "Change the subtype of an ABContact instance in ContactManager" on page 155.

Procedure

1. Make a query for contact data. You might proceed differently depending on whether the contact is linked to ContactManager or is just local to ClaimCenter:
   - If the contact is linked to ContactManager, run a query to find all instances of the contact with the `AddressBookUID` value equal to the `LinkID` value you got from ContactManager. Save the `PublicID` of each contact instance that the query finds. Additionally, note each claim for which the contact is a claim contact.
   - If the contact is local only to ClaimCenter, run a query in ClaimCenter to find the contact and save its `PublicID`. Note the claim for which the contact is a claim contact. You can run multiple queries if there are duplicate contact instances on other claims that you want to change.
2. Set the ClaimCenter server's runlevel to `maintenance`.

   For example, at a command prompt open in `ClaimCenter/admin/bin`, enter:

   ```
   system_tools -password adminuser-password -maintenance
   ```
3. Run the command-prompt tool for each contact instance your query found.

   For example, the user with login alinu has the role Contact Subtype Changer and password x2wTz@71P. For user alinu to change the contact with `PublicID` ab:123 to the subtype `Doctor`, at a command line open in `ContactManager/admin/bin`, the user would enter the following command, all on one line:

   ```
   maintenance_tools -user alinu -password x2wTz@71P
                     -changesubtypetargettype Doctor
                     -changesubtypepublicid ab:123
   ```
4. Note the messages in the console.

   For each command that completes successfully, you see messages similar to the following:

   ```
   Changing contact with publicID: ab:123 to subtype: Doctor
   done
   ```
5. Log in to ClaimCenter and ensure that you can open the contact in each claim and save changes. If necessary, update data as needed.
6. Take ClaimCenter out of `maintenance` runlevel.

   For example, at a command line open in `ClaimCenter/admin/bin`, enter:

   ```
   system_tools -password adminuser-password -multiuser
   ```

Troubleshooting the change subtype command-prompt utility

The command-prompt tool that changes the subtype of a contact instance is described at "Changing subtype with a command-prompt utility" on page 152. When you run this command, the following errors can be reported in the console.

Error message	Cause of the error
`Failed to change contact to subtype: Bad username or password`	Any of the following entries could have caused this problem: • You entered an invalid user name. • You entered the wrong password. • You entered the password without a user specified and you are not using the password of the Super User. Unless you are user Super User, the default user for this command, you must use the `-user` parameter and specify a user that has the permissions `changecontactsubtype` and `soapadmin`. For example, the role Contact Subtype Changer has these two permissions.
`Failed to change contact to subtype: Incorrect runlevel for subtype change. Required: MAINTENANCE, Actual: MULTIUSER`	You have not set the application runlevel to `maintenance` prior to running the command. Use the following command: `system_tools -password` *`adminuser-password`* `-maintenance`
`Failed to change contact to subtype: Contact with PublicID` *`public-id-value`* `not found`	You entered an incorrect value for the `PublicID` of the contact.
`Failed to change contact to subtype:` *`entity-name`* `is not a valid entity type`	The entity name you used is not valid. For example, you used `ABDoctor` in ClaimCenter, or `Doctor` in ContactManager.
`Failed to change contact to subtype:` *`entity-name`* `is not a valid subtype of Contact`	The entity name you used for the new subtype is not a subtype of `ABContact` or `Contact`. For example, you tried to change the contact subtype from `ABPerson` to `ABContact`.
`Failed to change contact to subtype: Cannot change Person to Company/Place or Company to Person/Place for contacts with Client tags`	The contact has a Client tag, which makes the contact usable by PolicyCenter and BillingCenter. Those applications do not support changing between a `Person` contact subtype and a `Company` or `Place` subtype. See "Subtype changes for client contacts" on page 153.
You see the list of command-prompt options for `maintenance_tools`, but no error message	Either you misspelled one of the command-prompt options or you did not enter one. Changing a contact subtype requires that you enter both options, `-changesubtypetargettype` and `-changesubtypepublicid`. See "Changing the subtype of a Contact instance" on page 152.

chapter 7

Contact tags

Contact tags enable you to classify contacts without having to add new subtypes. Every contact stored by ContactManager must have at least one tag. The three contact tag types provided in the base configuration are:

Client
A policy or account contact in PolicyCenter. For example, the holder of an account, the primary named insured on a policy, or a driver of a vehicle insured by a policy.

Vendor
ClaimCenter vendor providing services that help resolve claim losses. For example, a body shop, assessor, attorney, or physical therapy clinic.

Claim Party
A party to a claim in ClaimCenter, such as the insured or a claimant.

These tags are used to ensure that a contact is either a vendor, and therefore of interest only to ClaimCenter, or a client, of interest to PolicyCenter. A client can also be of interest to ClaimCenter and BillingCenter. Additionally, the Claim Party tag indicates that a contact has been added as a party to a claim. You can add additional tags and change how the applications use them.

IMPORTANT: Do not remove or edit the three contact tag types provided in the base configuration in the `ContactTagType` typelist. Any changes to these three tag types can cause your contact system integration to stop working. You can add new contact tag types to the base typelist in Guidewire Studio™. Have a good reason for doing so. If the new additions are not related to identifying contacts for searches, consider adding a separate array of identifiers rather than extending this typelist.

Applications and contact tags

The base configurations of PolicyCenter and BillingCenter save and retrieve only ContactManager contacts that have the Client tag. For example, a PolicyCenter user adds a client contact to a policy. PolicyCenter automatically sets the tag to Client and saves the contact in ContactManager. Later a claims adjuster adds this client contact to a claim as a party to the claim and saves the contact. ClaimCenter applies the Claim Party tag to the contact before sending it to ContactManager. The contact now has both the Client tag and the Claim Party tag.

When you add a new contact or edit a contact in ContactManager, you see a drop-down list that enables you to choose one or more contact tags. Additionally, you can see the contact tag on the detail screen for every contact.

When you work with contacts in the base configurations of ClaimCenter, PolicyCenter, and BillingCenter, you do not see any tag information in the user interface. The applications work with contact tags in the background and add the appropriate tags before sending a contact to ContactManager. After the tags are added to the contact in ContactManager, those tags are synchronized as part of the contact data copied over from ContactManager to the local contact record.

Note: A contact saved only locally in a core application does not have to have a tag. However, in their base configurations, the core applications always set tags for contacts when they are created and when they are saved to the database. Core applications set tags for contacts that they store locally even if the contacts are not stored in ContactManager.

Add a new contact tag

About this task

You can add a new contact tag type.

Procedure

1. Navigate in the Guidewire Studio™ **Project** window to **configuration** > **config** > **Metadata** > **Typelist**.
2. Right-click `ContactTagType.tti`.
3. Choose **New** > **Typelist Extension**.

   If you get a message saying that you cannot create a typelist from `ContactTagType.tti`, then `ContactTagType.ttx` already exists. You can open it in **configuration** > **config** > **Extensions** > **Typelist**.
4. If you do not get an error message, click **OK**.
   Guidewire Studio™ creates the file `ContactTagType.ttx` in **configuration** > **config** > **Extensions** > **Typelist**.

ClaimCenter contact tag generation

If a new contact created in ClaimCenter is a vendor subtype, ClaimCenter adds a Vendor tag when it saves the contact to ContactManager. If you add a contact to a claim, ClaimCenter ensures that it has the Claim Party tag before saving the contact in ContactManager.

The base configuration of ClaimCenter sets tags in the class `gw.api.contact.CCContactMinimalTagsImpl`. If you add new tags, you must also set them in this class.

PolicyCenter contact tag generation

PolicyCenter ensures that any contacts it sends to ContactManager have Client tags. PolicyCenter cannot find a contact in ContactManager unless the contact has a Client tag.

The base configuration of PolicyCenter sets tags in the class `gw.api.contact.PCContactLifecycle`. If you add new tags, you must also set them in this class.

BillingCenter contact tag generation

BillingCenter ensures that any contacts it sends to ContactManager have Client tags. BillingCenter cannot find a contact in ContactManager unless the contact has a Client tag.

The base configuration of BillingCenter sets tags in the Contact Preupdate rule **Add default tags**. If you add new tags and want to set them for contacts that BillingCenter sends to ContactManager, you must set them in this rule.

Contact tag-based security

There are permissions associated with tags and there are permission check expressions used to control access to screens.

Contact tag permissions

The base application permissions, listed in the table that follows, are special permissions that enable a user to create, edit, delete, and view contacts that have any tag. You can add permissions for creating, editing, deleting, and viewing specific tags, just as you can for contact subtypes.

The tag permissions provided in the base configurations are:

Code	Permission Description
`anytagcreate`	Create a new contact regardless of which contact tag it requires.
`anytagedit`	Edit a contact that has any contact tag.
`anytagdelete`	Delete a contact that has any contact tag.
`anytagview`	See a contact that has any contact tag.

Contact tag permission check expressions

Guidewire applications use a set of permission check expressions to control access to screens and widgets related to contacts. In ContactManager, to perform an action, users need permission for both the contact's subtype and the contact's tags.

For example, to edit a `CompanyVendor` contact, the user needs permission to edit contacts of that subtype. If the contact has the Vendor tag, the user also needs permission to edit contacts with the Vendor tag. If you have not extended the tag permissions, the base configuration `anytagedit` permission gives that user permission to edit vendor contacts.

See also

- For more information on contact tag permissions, see:
  - “ContactManager contact and tag permission check expressions” on page 90
  - “ClaimCenter contact and tag permission check expressions” on page 100
  - “BillingCenter contact and tag permission check expressions” on page 97
  - “ClaimCenter contact subtype and tag permissions” on page 99
  - “BillingCenter contact security” on page 97
- You can associate permissions with tags in Guidewire Studio™. For information on adding new tag permissions, see:
  - “Creating client tag permissions in ContactManager” on page 94
  - “Create claim party and vendor tag permissions in ClaimCenter” on page 105

chapter 8

Vendor services

Overview of vendor services

Vendor services in ContactManager support the Services feature in ClaimCenter. You can create the services that your vendors provide, like carpentry or auto towing, and connect them to the vendor contacts who provide them. ContactManager maintains the connection between each vendor and the vendor's services and enables ClaimCenter to search for services and get a list of vendors who provide them.

Vendor services provide a way to classify vendor contacts, contacts that have the Vendor tag, without having to add new subtypes. For example, a construction company that is a `CompanyVendor` entity provides several construction services for property claims: carpentry, painting, and plumbing, as well as independent appraisal. A claim comes in to ClaimCenter that requires major plumbing work on a property. The claims adjuster creating the claim in the New Claim wizard in ClaimCenter can search in ContactManager for vendors who provide the plumbing service. The adjuster can then choose a vendor from the list of contacts returned by ContactManager.

Only vendor contacts, contacts that have the Vendor tag, can have services. ContactManager maintains the connection between a vendor contact and the services that the contact provides. When ContactManager receives a search request for a service from ClaimCenter, ContactManager returns vendor contacts that match the service.

Both ContactManager and ClaimCenter support a services directory, a tree structure of `SpecialistService` entities, each of which can have child and parent `SpecialistService` entities. The purpose of this tree structure in ContactManager is to display the tree in contact editing screens so you can add and remove services for contacts. When you add services to a contact, ContactManager links them to that contact.

In the base configuration, the services directory is not populated. You must implement services in XML and import them through the administration interface to load your services into ContactManager. In the base configuration, there is a sample services directory in the file `vendorservicetree.xml`. You can edit this file and create your own services.

If you load sample data for testing purposes, the sample services directory also loads, and you can see the services directory in various places in the user interface.

IMPORTANT: If you load sample data, the services that are loaded with it cannot be deleted from the database without dropping the entire database. Do not load sample services into an installation for which you cannot drop the database, such as a production system. While you cannot delete a service from the database, you can set a service's `Active` flag to `false`, which makes it inactive.

In ContactManager, if you have loaded sample data, you can see the services tree when you edit a contact that has the Vendor tag. An example is AB Construction, which has the Vendor tag and is a `VendorCompany` entity. If you search for that contact and then click that contact in the search results, on the **Services** card you can see all the services that contact provides. If you edit that contact and click the **Services** card, you see the full Services hierarchy, and you can add and remove services by clicking their check boxes.

See also

- For information on vendor services in ClaimCenter, see the *Application Guide*

Working with vendor services in ContactManager

To be able to work with vendor services in ContactManager, you must first create a set of services in an XML file, such as those in the `vendorservicetree.xml` sample file. You then import the file into ContactManager. For information on this service tree data model and the XML syntax required, see "Vendor services directory data model" on page 171.

Importing and exporting vendor service data

You can import and export vendor services data in ContactManager on the **Administration** tab. You must be logged in as an administrator with the permissions `soapadmin` and `viewadmin`. For example, you might import vendor service data to establish your service tree for the first time, to add more services, or to make an obsolete service inactive.

IMPORTANT: Guidewire recommends that you keep your vendor service tree definitions identical in ContactManager and ClaimCenter. As you make changes, import the same vendor service tree XML definitions into both ContactManager and ClaimCenter. ClaimCenter has an additional set of vendor service details definitions that do not apply to ContactManager. Do not import the ClaimCenter `vendorservicedetails.xml` file into ContactManager. See information on how to import and configure services in the *Configuration Guide*.

Import vendor service data

About this task

When you import vendor services, ContactManager identifies them by their `code` value. Importing vendor services instantiates any new `SpecialistService` entities, defined with new `code` values, in the file and adds them to the tree. Any services you import become permanent and cannot be deleted from the database, although you can deactivate them.

Procedure

1. In ContactManager, open the **Administration** tab and navigate to **Utilities** > **Import Data**.
2. Click **Browse** to find the file in which you have defined the vendor services tree.
   For example, navigate to the sample `vendorservicetree.xml` file in `ContactManager/modules/configuration/config/sampledata`.
3. Double-click the file to open it in ContactManager.
   ContactManager parses the file and verifies that it is valid and can be loaded.
4. If you have existing service tree definitions, you might see a prompt asking what to do to resolve matches with existing records. In response to this prompt, choose:

   Overwrite all existing records
5. Click **Finish** to load the file.

Export vendor service data

Procedure

1. In ContactManager, open the **Administration** tab and navigate to **Utilities** > **Export Data**.
2. Click the drop-down list for **Data to Export** and choose **Vendor Service Tree**.

3. Click **Export**.

Results

ContactManager exports the data into the file `vendorservicetree.xml`, and your browser downloads the file. For example, if your browser is Google Chrome on Windows 7, exporting this file saves it by default in `C:/Users/`*`yourLogin`*`/Downloads`.

Viewing, adding, and removing vendor service data for contacts

In ContactManager when you view a contact, if the contact is a vendor—has a Vendor tag—you see a **Services** card. If you click the **Services** card, you can see all the vendor services assigned to this contact.

When you edit a contact, if the contact is a vendor, there is a **Services** card in which you can add and remove contacts. If you click the **Services** card, you see the entire **Vendor Services Tree**. In this tree, you select check boxes to add vendor services, and you deselect check boxes to remove services.

View vendor service data for a contact

Procedure

1. Log in to ContactManager as a user with the `anytagview`, `abview`, and `abviewsearch` permissions.
   If you have imported sample data, you can log in as a user that has the Contact Manager role, such as `aapplegate` with password `gw`.
2. Click the **Contact** tab and click **Search**.
3. Search for a contact who is a vendor.
   For example, if you have imported sample data, you can click the **Contact Type** drop-down list and choose **Vendor (Company)**. Then enter `a` as the name and click **Search**.
4. Select a contact in the search results.
   For example, click the sample contact AB Construction.
5. When the contact details screen opens, click the **Services** card. On that card, all services defined for this contact are listed. Each service is shown with the category and subcategory to which it belongs, if any.
   When you click the **Services** card for the sample contact AB Construction, the services listed include Carpentry, Drying, Flooring, and Painting.

Add services to or remove services from a contact

Procedure

1. Log in to ContactManager as a user who has the `ABContact` permissions `abcreate`, `abedit`, `abdelete`, `abview`, and `abviewsearch`, and the tag permissions `anytagiew`, `anytagedit`, and `anytagcreate`.
   If you have imported sample data, you can log in as a user that has the Contact Manager role, such as `aapplegate` with password `gw`.
2. Click the **Contact** tab and click **Search**.
3. Search for a contact who is a vendor.
   For example, if you have imported sample data, you can click the **Contact Type** drop-down list and choose **Vendor (Company)**. Then enter `a` as the name and click **Search**.
4. Click a contact in the search results.
5. When the contact details screen opens, click **Edit**.
6. Click the **Services** card, which displays the entire vendor services tree. Each service you can select has a check box next to it.
   - Select a check box to add a service to the contact.

- Clear a check box to remove a service from the contact.

7. When you have finished selecting and clearing all the services for this contact, click **Update**.
For example, click the sample contact AB Construction and then click **Edit**. On the **Services** card, you see that services like Carpentry, Drying, Flooring, and Painting are all selected. If you clear a check box next to one of these services, that service is removed when you click **Update**.

Adding services to or removing services from multiple contacts

If you have many contact instances that need to have services added to them, adding or removing services one at a time in the user interface might not be practical. You can use instead a feature called Vendor Services Onboarding. This feature requires exporting, modifying, and importing a CSV file to add services to and remove services from multiple contacts.

See also

- “Vendor services onboarding” on page 281

Making changes to vendor services

You can export existing vendor services to an XML file, make changes to the exported file, and then import the file to update your services. This process is described in detail at “Importing and exporting vendor service data” on page 164.

For most changes, this process works without requiring additional actions. For example:

- You can disable a service. ContactManager removes it from any contacts that were defined to have that service. You might want to assign different services to those contacts, but you do not have to.
- You can add new services to existing categories or add new categories and services. ContactManager adds them to the lists in the user interface. You can then assign the services to contacts.

There are two types of service changes that can cause errors or require additional work:

- Adding more levels to the service hierarchy.
- Changing an existing service type—a leaf in the services tree—into a category, which is a parent node in the services tree.

Adding additional levels to the vendor service hierarchy

In the base configuration, the vendor service hierarchy has up to three levels, called Category, Sub Category, and Service Type in application screens. You can define additional levels of services in `vendorservicetree.xml`. If you add additional service tree levels, you must configure the ClaimCenter and ContactManager application screens that display the service tree to support these additional levels.

See also

- For information on how to configure the depth of the service hierarchy, see the *Configuration Guide*

Change an existing vendor service into a category

You can change a vendor service type into a category that includes other vendor services.

About this task

If you change a vendor service type into a category, if that service type was previously assigned to one or more contacts, the relationship becomes invalid. Therefore, in addition to changing the service type to a category, you also need to query for contacts that were assigned the original service type.

> **Note:** You change services by importing an XML file with new service definitions in it. As with any service changes, be sure to import the new service definitions into both ContactManager and ClaimCenter.

Initially, in this example, the following category and service type are defined in ContactManager and ClaimCenter:

Category	Service Type
Auto	Adjudicate Claim

- This service is defined in `vendorservicetree.xml` as shown in the following partial definition:

```
...
<SpecialistService public-id="svc:aut">
    <Active>true</Active>
    <Code>auto</Code>
    <Description displayKey="true">Services.Auto.Description</Description>
    <Name displayKey="true">Services.Auto.Name</Name>
    <Parent/>
</SpecialistService>
<SpecialistService public-id="svc:aut_adj">
    <Active>true</Active>
    <Code>autoadjudicate</Code>
    <Description displayKey="true">Services.Auto.AdjudicateClaim.Description</Description>
    <Name displayKey="true">Services.Auto.AdjudicateClaim.Name</Name>
    <Parent public-id="svc:aut"/>
</SpecialistService>
...
```

  For information on this XML syntax, see “SpecialistService XML syntax” on page 171. For a fuller example, see “Specialist services directory example” on page 173.

- Additionally, the Adjudicate Claim service has been assigned to the following contacts:
  - Allendale, Myers & Associates, a law firm
  - Express Auto, an auto repair shop
  - James Anderson, an attorney
  - Leland Associates, a law firm
  - Lily Watson, an attorney

Procedure

1. Make Adjudicate Claim a subcategory for new services named Litigation and Arbitration:

Category	Sub Category	Service Type
Auto	Adjudicate Claim	Litigation
		Arbitration

   A contact, such as a law firm or attorney, that provides litigation services can pursue a lawsuit, an adversary proceeding. A contact that provides arbitration services can provide an arbitrator to negotiate a settlement without an adversary proceeding that could lead to court. Some contacts can do both.

2. The new service definitions in `vendorservicetree.xml` are:

```
<SpecialistService public-id="svc:aut">
    <Active>true</Active>
    <Code>auto</Code>
    <Description displayKey="true">Services.Auto.Description</Description>
    <Name displayKey="true">Services.Auto.Name</Name>
    <Parent/>
</SpecialistService>
<SpecialistService public-id="svc:aut_adj">
    <Active>true</Active>
    <Code>autoadjudicate</Code>
    <Description displayKey="true">Services.Auto.AdjudicateClaim.Description</Description>
    <Name displayKey="true">Services.Auto.AdjudicateClaim.Name</Name>
    <Parent public-id="svc:aut"/>
</SpecialistService>
<SpecialistService public-id="svc:aut_adj_litig">
    <Active>true</Active>
    <Code>autolitigate</Code>
    <Description displayKey="true">Services.Auto.AdjudicateClaim.Litigation.Description</Description>
    <Name displayKey="true">Services.Auto.AdjudicateClaim.Litigation.Name</Name>
    <Parent public-id="svc:aut_adj"/>
</SpecialistService>
<SpecialistService public-id="svc:aut_adj_arbit">
    <Active>true</Active>
    <Code>autoarbitrate</Code>
    <Description displayKey="true">Services.Auto.AdjudicateClaim.Arbitration.Description</Description>
    <Name displayKey="true">Services.Auto.AdjudicateClaim.Arbitration.Name</Name>
    <Parent public-id="svc:aut_adj"/>
</SpecialistService>
```

3. Open Guidewire Studio™.
4. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.
5. Add the four new display keys for U.S English to the file:
   For example, add the following display keys:

```
Services.Auto.AdjudicateClaim.Litigation.Description = Litigation
Services.Auto.AdjudicateClaim.Litigation.Name = Litigation
Services.Auto.AdjudicateClaim.Arbitration.Description = Arbitration
Services.Auto.AdjudicateClaim.Arbitration.Name = Arbitration
```

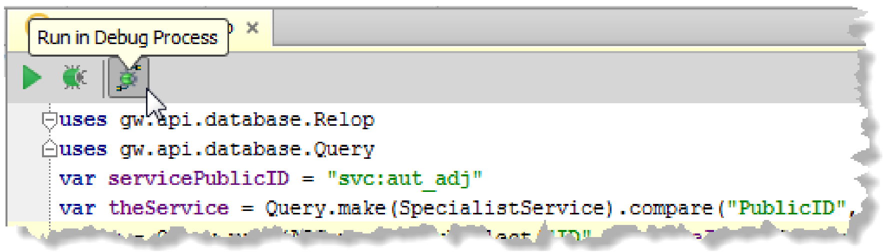
6. Optionally open other `display_LanguageCode.properties` files to add translations for the new display keys.
   For example, open `display_de.properties` and add the following German translations for the new display keys:

```
Services.Auto.AdjudicateClaim.Litigation.Description = Rechtsstreit
Services.Auto.AdjudicateClaim.Litigation.Name = Rechtsstreit
Services.Auto.AdjudicateClaim.Arbitration.Description = Schiedsverfahren
Services.Auto.AdjudicateClaim.Arbitration.Name = Schiedsverfahren
```

7. Import `vendorservicetree.xml` as described in “Import vendor service data” on page 164.
   You do not see any error messages when you perform this import.
8. When you change a service type into a category, you must then check to see if the changed service type has been assigned to any contacts. The steps that follow show how to run a query to do this check.
9. If ContactManager is running, stop the server.
10. Open Guidewire Studio™ for ContactManager and choose **Run** > **Debug** > **Server** to run the ContactManager server in debug mode.
    A **Console** pane opens below the current window and shows the progress of the server startup.
11. Wait for ContactManager to start in debug mode.
12. Open **Tools** > **Gosu Scratchpad**.
13. Enter a query in the **Gosu Scratchpad** that searches for contacts that have been assigned the original service type, in this case, `svc:aut_adj`:

```
uses gw.api.database.Relop
uses gw.api.database.Query
var servicePublicID = "svc:aut_adj"
var theService = Query.make(SpecialistService).compare(
    "PublicID", Relop.Equals,
    servicePublicID).select().AtMostOneRow
var q = Query.make(ABContact).subselect("ID", CompareIn,
    ABContactSpecialistService, "ABContact").compare(
    "SpecialistService", Relop.Equals, theService.ID)
var result = q.select()
for(r in result) {
  print("ABContact - ${r.DisplayName}, ${r.LinkID}")
}
```

14. Click the **Run in Debug Process** button in the **Gosu Scratchpad** tool bar, as shown in the following:



```
uses gw.api.database.Relop
uses gw.api.database.Query
var servicePublicID = "svc:aut_adj"
var theService = Query.make(SpecialistService).compare("PublicID",
```

15. The results display in the **Console** pane below the current window. In this example, there are five results:
    - `ABContact` - Express Auto, absample:2

- `ABContact` - Lily Watson, ab:67
- `ABContact` - James Andersen, ab:68
- `ABContact` - Leland Associates, ab:93
- `ABContact` - Allendale, Myers & Associates, ab:92

16. At this point, you can decide how to reassign the services to these contacts:
- For a small number of contacts, use ContactManager to search for each contact and edit its services. See "Reassign vendor services in the ContactManager contact screens" on page 169.
- For a large number of contacts, use the vendor onboarding process. See "Use vendor onboarding to reassign services" on page 170.

Reassign vendor services in the ContactManager contact screens

Procedure

1. Log in to ContactManager as a user who has the `ABContact` permissions `abcreate`, `abedit`, `abdelete`, `abview`, and `abviewsearch`, and the tag permissions `anytagiew`, `anytagedit`, and `anytagcreate`.
If you have imported sample data, you can log in as a user that has the Contact Manager role, such as `aapplegate` with password `gw`.

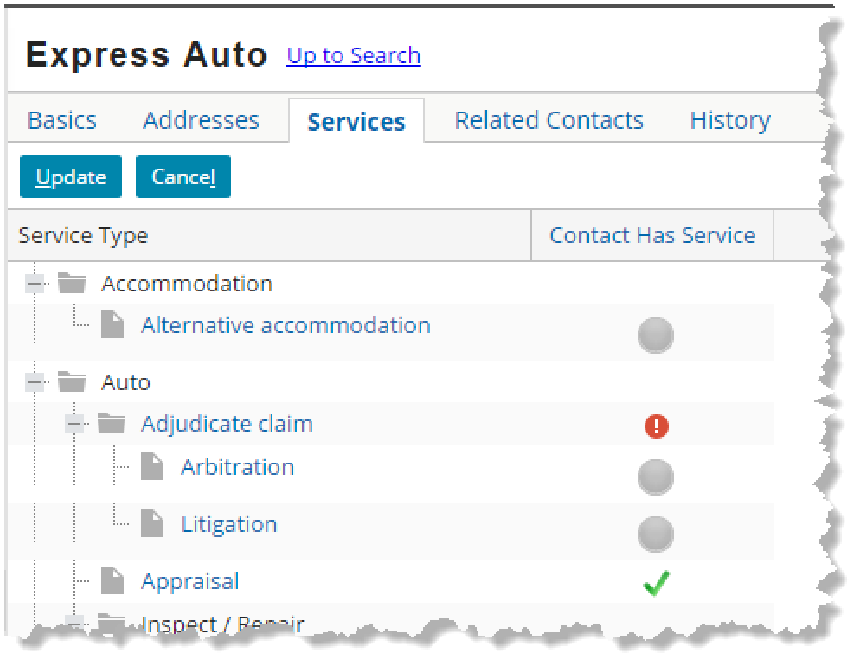
2. Click the **Contact** tab and click **Search**.

3. Search for a vendor contact who is on the list returned by the query in the previous steps.
For example, click the **Contact Type** drop-down list and choose **Contact**. In the **Tags** field, choose **Vendor**. Then enter `Express Auto` as the name and click **Search**.

4. Click the name that appears in the **Search Results**.
For example, Express Auto.

5. Click the **Services** card, and then click **Edit**.

6. Click the red warning icon next to Adjudicate Claim to deselect it, as shown in the following:



Service Type	Contact Has Service
Accommodation	
Alternative accommodation	●
Auto	
Adjudicate claim	!
Arbitration	●
Litigation	●
Appraisal	✓
Inspect / Repair	

7. Click either Arbitration or Litigation or both, as appropriate for this contact.

8. Repeat these steps for each contact in the list of query results.
For example:

- `ABContact` - Express Auto, absample:2
- `ABContact` - Lily Watson, ab:67
- `ABContact` - James Andersen, ab:68
- `ABContact` - Leland Associates, ab:93
- `ABContact` - Allendale, Myers & Associates, ab:92

Use vendor onboarding to reassign services

Before you begin

These steps use the example at “Export the initial set of vendor services onboarding CSV files” on page 286.

About this task

Vendor Services Onboarding enables you to map contacts by exporting a set of CSV files, mapping contacts to services in the exported files, and then importing the changed CSV files. If you have a lot of contacts to update with the new services, this technique can be useful.

Procedure

1. Export your vendor contact data as CSV files.
   In the example, you see the following message for the Adjudicate Claim service because it is now a category and not a service:
   ```
   Branch service Adjudicate Claim exported but will not import
   ```
2. Open each CSV file in turn and do the following:
   a) Look for mappings in the column that has become a category and move them to the new service type columns.

      In the example, the old service type column is named Adjudicate Claim. It is likely to be the last column on the right. If a contact is mapped to this column, you see `On` in the cell for that contact. Copy this value and paste it into either the Litigation or Arbitration column, as appropriate for each contact row in the file.

   b) When you have finished processing all the cells in the column for the former service type, delete that column. In the example, you delete the Adjudicate Claim column.

   c) Save the file as a CSV file.
3. When you have updated the contact information in all the CSV files and you have saved the files, re-import the files into ContactManager.
   If you have not deleted the former service type column from a file, you see a message similar to the following:
   ```
   Column with ID: autoadjudicate cannot be added. File C:\outputfiles
   \VendorServicesLoad-Tue Apr 21 15.51.51 PDT 2018\Attorney-0.csv needs to be fixed
   to complete import.
   ```

What to do next

See also

- For information on mapping contacts, see “Map services to vendor contacts” on page 287.
- “Import mapped contacts in vendor services onboarding CSV files” on page 289.

Vendor services data model

There are two aspects to the services data model in ContactManager:

- The tree-structured Services directory, which is composed of `SpecialistService` entities, each of which can have child and parent `SpecialistService` entities.

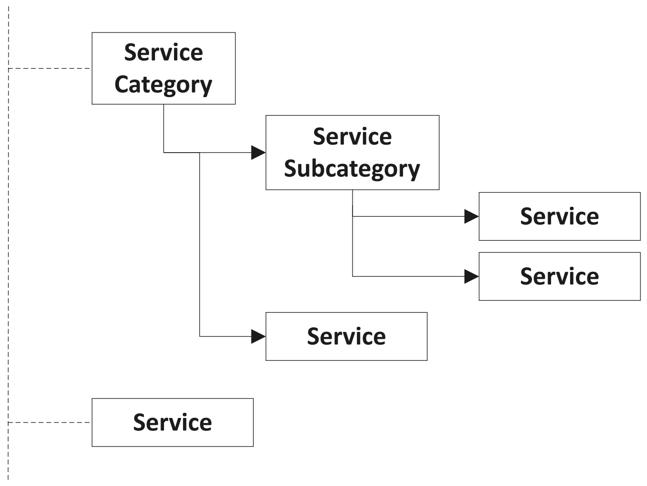
- The data model for connecting the `ABContact` entity to `SpecialistService` entities.

Vendor services directory data model

The data model for the vendor services directory consists of a hierarchy of parent services, or *categories*, and child services. Each node of the directory is an instance of a `SpecialistService` entity. This data model supports a tree-structured directory that is used in ContactManager screens in which you select services to add to or remove from a vendor contact. In those screens, the services you can add and remove are terminal nodes, also known as *leaf nodes*, that are not parents of other nodes.

Additionally, the vendor services directory enables ContactManager to show the category, subcategory, and service for each service that has been added to a contact.

In the base configuration, the Services Directory tree has three levels as shown in the following figure:



A leaf node, a *Service* in the figure, can appear at any level.

A node without a parent can be useful as a general service category, such as the Service Category node in the figure. It becomes a top level node in the tree.

A single service or category can have only one immediate parent. If you want a service or category to be listed in the tree under two different parents, you must create it twice and give each instance a separate parent.

You can create a tree with more than three levels, but you also have to configure PCF files to support the extra levels. There is an example of how to do this configuration in the "ABContact and SpecialistService entity relationships" on page 173.

See also

- "SpecialistService XML syntax" on page 171
- For information on configuring the depth of the service hierarchy, see the *Configuration Guide*.

SpecialistService XML syntax

You can specify all the persistent `SpecialistService` fields in XML by using the XML syntax for a `SpecialistService` entity. There are also virtual properties, like `Leaf`, that are determined after ContactManager constructs the entire tree.

Note: You define vendor services in an XML file and import the file into ContactManager and ClaimCenter. For a sample file, you can open the `vendorservicetree.xml` file. See "Specialist services directory example" on page 173.

The following sample code shows the beginning of the file and the definition of an Auto category, to help you identify the elements.

```
<?xml version="1.0"?>
<import>
<SpecialistService public-id="svc:aut">
    <Active>true</Active>
    <Code>auto</Code>
    <Description displayKey="true">Services.Auto.Description</Description>
    <Name displayKey="true">Services.Auto.Name</Name>
    <Parent/>
</SpecialistService>
...
</import>
```

<import>

Specifies that the enclosed XML uses the administrative data import XSD files `ab_import.xsd` and `ab_entities.xsd`.

When you export this file from ContactManager, the Administrative Export feature populates the following two attributes:

`xmlns`
: The namespace for this XML file. The default values is `"http://guidewire.com/ab/exim/import"`

`version`
: A version assigned to this XML file. For example, `"p5.62.a10.225.26"`.

Both attributes are optional. It is not necessary to maintain these attributes in your files.

If you want to see the XSD files, you can generate them as follows:

- At a command prompt open in the ContactManager installation folder, enter the following command:

  ```
  gwb genImportAdminDataXsd
  ```

- ContactManager generates the XSD files in the following directory:

  ```
  ContactManager/build/xsd
  ```

<SpecialistService>

Specifies the name of the entity, `SpecialistService`.

This element takes the attribute `public-id`. In the `SpecialistService` entity, the value of the `public-id` attribute becomes the value of the `PublicID` field.

Elements of <SpecialistService>

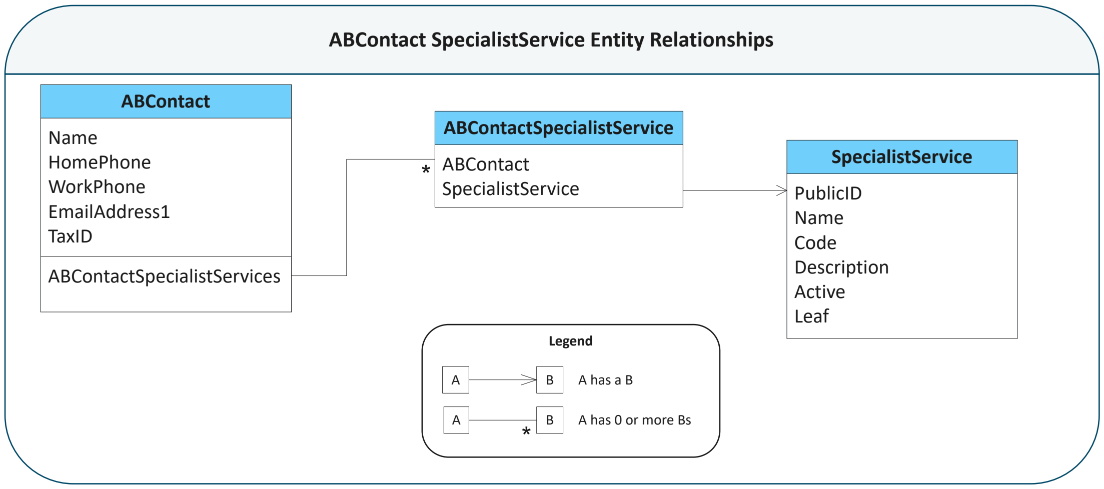
The following table describes the elements of the `<SpecialistService>` XML element.

XML Element	Field	Description
`<Active>`	`Active`	Indicates if the service is available to be used in new service requests. Setting this element to `false` is a way to make the service unavailable in the user interface if you no longer use it. You cannot delete a service from the database without dropping the database.
`< Code>`	`Code`	A unique string identifying the service. Used by ContactManager code to reference the service instance.
`<Description displayKey="true">`	`Description`	Specifies a display key that defines the service description. Any new display key must also be added in Guidewire Studio™ to the display properties files.
`<Name displayKey="true">`	`Name`	Specifies a display key that describes the service name. Any new display key must also be added in Guidewire Studio™ to the

XML Element	Field	Description
		display properties files. This name is used in the service tree in contact editing screens and as the name of a service for a contact in a contact's **Services** card.
`<Parent>`	`Parent`	The parent `SpecialistService` for this entity instance. If the XML tag is defined as empty, `<Parent/>`, the `Parent` field is `null`, and the service is a top-level category or service in the Services Directory.

ABContact and SpecialistService entity relationships

ContactManager can link an array of specialist services to an `ABContact`. A contact must have a Vendor tag for you to be able to add a service to it in the contact editing screen. After you add services, the contact references the services as an array, as shown in the following figure.



See also

- "Specialist services directory example" on page 173

Specialist services directory example

For this example, you categorize some services at the top level as Auto and Property.

- The Auto category in this example has one subcategory, Inspect/Repair, with two services, Audio Equipment and Auto Body.
- The Property category has two subcategories, Inspection and Construction Services.
- The Inspection subcategory has one child, the service Independent Appraisal.
- The Construction Services subcategory has two children, General Contractor and Carpentry.

The following table shows this set of categories and services. The Services column shows leaf nodes, services that can be assigned to a contact.

Category	Subcategory	Services
Auto	Inspection / Repair	Audio Equipment
		Auto Body
Property	Inspection	Independent appraisal

Category	Subcategory	Services
	Construction services	General contractor
		Carpentry

Specialist services example XML entries

When you import the service tree XML file, ContactManager creates `SpecialistService` entities based on the definitions in the XML file and sets properties like parent, name, description, and public ID.

IMPORTANT: You must import the same service tree XML file in both ContactManager and ClaimCenter so the service definitions match exactly. The base configuration provides a `vendorservicetree.xml` file with a set of services predefined. You can define your own services by using this file as an example, and then import it into ContactManager and ClaimCenter.

The following code, when imported, produces the specialist services directory example:

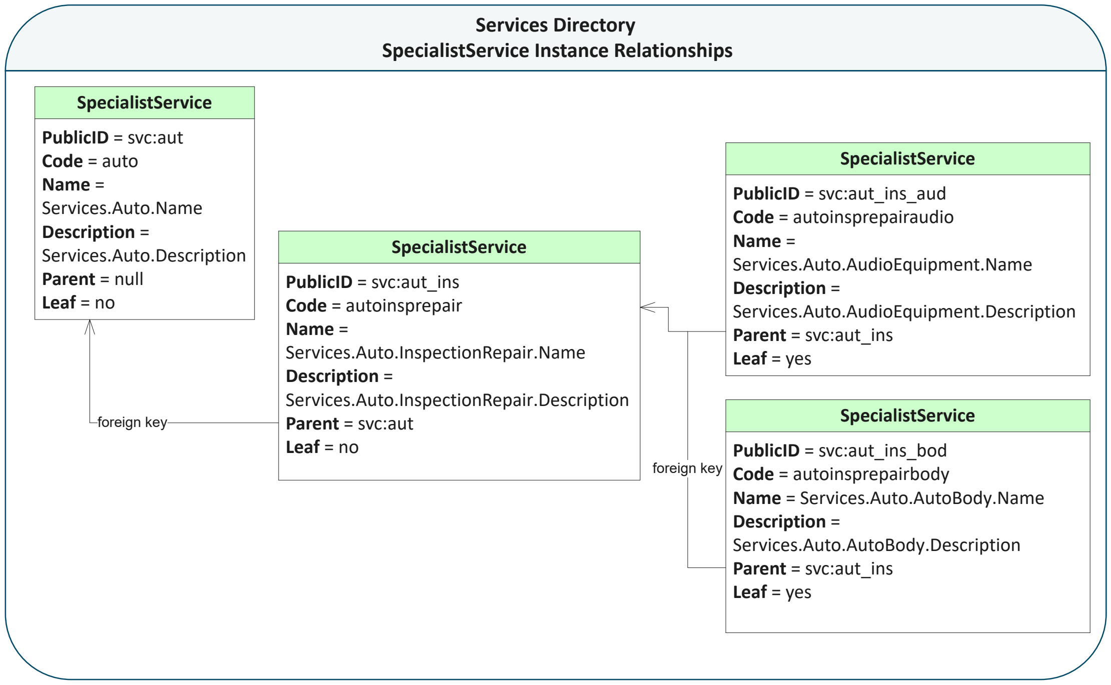
```
<?xml version="1.0"?>
<import>
<SpecialistService public-id="svc:aut">
    <Active>true</Active>
    <Code>auto</Code>
    <Description displayKey="true">Services.Auto.Description</Description>
    <Name displayKey="true">Services.Auto.Name</Name>
    <Parent/>
</SpecialistService>
<SpecialistService public-id="svc:aut_ins">
    <Active>true</Active>
    <Code>autoinsprepair</Code>
    <Description displayKey="true">Services.Auto.InspectionRepair.Description</Description>
    <Name displayKey="true">Services.Auto.InspectionRepair.Name</Name>
    <Parent public-id="svc:aut"/>
</SpecialistService>
<SpecialistService public-id="svc:aut_ins_aud">
    <Active>true</Active>
    <Code>autoinsprepairaudio</Code>
    <Description displayKey="true">Services.Auto.AudioEquipment.Description</Description>
    <Name displayKey="true">Services.Auto.AudioEquipment.Name</Name>
    <Parent public-id="svc:aut_ins"/>
</SpecialistService>
<SpecialistService public-id="svc:aut_ins_bod">
    <Active>true</Active>
    <Code>autoinsprepairbody</Code>
    <Description displayKey="true">Services.Auto.AutoBody.Description</Description>
    <Name displayKey="true">Services.Auto.AutoBody.Name</Name>
    <Parent public-id="svc:aut_ins"/>
</SpecialistService>
<SpecialistService public-id="svc:pro">
    <Active>true</Active>
    <Code>prop</Code>
    <Description displayKey="true">Services.Property.Description</Description>
    <Name displayKey="true">Services.Property.Name</Name>
    <Parent/>
</SpecialistService>
<SpecialistService public-id="svc:pro_ins">
    <Active>true</Active>
    <Code>propinspect</Code>
    <Description displayKey="true">Services.Property.Inspection.Description</Description>
    <Name displayKey="true">Services.Property.Inspection.Name</Name>
    <Parent public-id="svc:pro"/>
</SpecialistService>
<SpecialistService public-id="svc:pro_ins_ind">
    <Active>true</Active>
    <Code>propinspectindependent</Code>
    <Description displayKey="true">Services.Property.IndependentAppraisal.Description</Description>
    <Name displayKey="true">Services.Property.IndependentAppraisal.Name</Name>
    <Parent public-id="svc:pro_ins"/>
</SpecialistService>
<SpecialistService public-id="svc:pro_con">
    <Active>true</Active>
    <Code>propconstrserv</Code>
    <Description displayKey="true">Services.Property.ConstructionServices.Description</Description>
    <Name displayKey="true">Services.Property.ConstructionServices.Name</Name>
    <Parent public-id="svc:pro"/>
</SpecialistService>
<SpecialistService public-id="svc:pro_con_gen">
    <Active>true</Active>
    <Code>propconstrservgencontractor</Code>
    <Description displayKey="true">Services.Property.GeneralContractor.Description</Description>
    <Name displayKey="true">Services.Property.GeneralContractor.Name</Name>
    <Parent public-id="svc:pro_con"/>
</SpecialistService>
<SpecialistService public-id="svc:pro_con_car">
    <Active>true</Active>
    <Code>propconstrservcarpentry</Code>
    <Description displayKey="true">Services.Property.Carpentry.Description</Description>
    <Name displayKey="true">Services.Property.Carpentry.Name</Name>
    <Parent public-id="svc:pro_con"/>
</SpecialistService>
</import>
```

See also

- "Specialist services directory example" on page 173
- "Specialist services example instance relationships" on page 175

Specialist services example instance relationships

The XML example in "Specialist services example XML entries" on page 174 creates a `SpecialistService` instance for each `<SpecialistService>` XML entry. Each entity definition has a public ID, a code, a name, a description, and, if the service has a parent, a pointer to the parent. The following figure shows the entity instances that would be created in ContactManager for the Auto portion of the XML in the previous example.


Services Directory
SpecialistService Instance Relationships
SpecialistService
PublicID = svc:aut
Code = auto
Name = Services.Auto.Name
Description = Services.Auto.Description
Parent = null
Leaf = no
SpecialistService
PublicID = svc:aut_ins
Code = autoinsprepair
Name = Services.Auto.InspectionRepair.Name
Description = Services.Auto.InspectionRepair.Description
Parent = svc:aut
Leaf = no
SpecialistService
PublicID = svc:aut_ins_aud
Code = autoinsprepairaudio
Name = Services.Auto.AudioEquipment.Name
Description = Services.Auto.AudioEquipment.Description
Parent = svc:aut_ins
Leaf = yes
SpecialistService
PublicID = svc:aut_ins_bod
Code = autoinsprepairbody
Name = Services.Auto.AutoBody.Name
Description = Services.Auto.AutoBody.Description
Parent = svc:aut_ins
Leaf = yes
foreign key
foreign key

chapter 9

Vendor document management

ContactManager enables you to attach documents to vendor contacts. For example, there might be a service level agreement that you require for vendors. Using ContactManager, you can attach the agreement document to a vendor contact, and ContactManager then maintains the connection.

If ClaimCenter is integrated with ContactManager, you can see read-only information about the documents attached to a vendor contact in the contact's detail view. These documents remain attached to the vendor contact independently of the current claim. You cannot attach documents to vendor contacts independently of claims in ClaimCenter, but must log in to ContactManager for that purpose.

IMPORTANT: Guidewire recommends integrating with an external document management system rather than using the default demonstration document management system on the ContactManager server. The default system is useful only for demonstration purposes and does not support features of a real document management system, such as document versioning.

See also

- For information on document management in ClaimCenter, see the *Application Guide*.

Overview of vendor documents

The Vendor Documents feature provides a way to attach documents to vendor contacts in ContactManager. In the base configuration, only vendor contacts, contacts that have the Vendor tag, can have documents attached to them. ContactManager maintains the connection between vendor contacts and the documents attached to them. You can see which documents are attached to a vendor contact when you view the contact's detailed information, either in ContactManager or in ClaimCenter.

Document storage in the base configuration

This topic describes how ContactManager stores documents as configured in the base product. You can configure how ContactManager uses metadata properties, stores content, responds to document requests, and so on.

In the base configuration, documents are stored as a combination of:

Metadata

Properties that specify information about a document. In the base configuration, ContactManager stores these properties in the database. For example, there are properties for the document's name, the document's file type, an optional type classification, and so on. When you create a new document, you must specify some of its properties before you can save it.

For example, you see document properties when you click the Info action for a document in the **Details** view for a contact.

Document content

A file that is stored in the ContactManager demonstration DMS. In general, you create and edit document content as a file on your local system by using your editing software. Before uploading the content, you specify the metadata representing the document in ContactManager. You then upload the file to the server, which associates the file with its metadata and stores the document content in the demonstration DMS.

For example, you can view a document's content by clicking the document name in the **Details** view for a contact.

Note: If you are just indicating the existence of a document, the document is hard copy and there is no content to upload. The document metadata is stored by ContactManager and typically the document name and description indicate where the hard copy is located.

See also

- "Document storage overview" in the *Integration Guide*

Document metadata properties

When you create a new document or edit an existing document, you see a set of metadata properties that the base configuration of ContactManager stores in the database. Document search uses a subset of these properties.

You can set the following metadata properties for a document:

Name

The name of the document, a required value. ContactManager uses this name for the document content file. For example, if you download the content for a document, this setting determines the name of the file name sent to the browser. For hard-copy documents, this field provides information helpful for identifying the hard-copy document.

Description

Especially useful for locating hard-copy documents.

File type

The type of content file, also known as a MIME type. This value is set by ContactManager when it initially detects the file you want to upload. This property is required.

IMPORTANT: You can change the file type, but do so with caution. ContactManager uses your setting to set the MIME type for the file. The operating system formats the document content file to match this MIME type when you upload the content.

Note: The **File Type** field does not apply to documents representing hard copy documents because there is no content for this kind of document.

Author

By default, the name of the user who attached the document to the vendor contact. This field can be changed to some other value, such as the sender of a document.

Recipient

The person or business to which the document was sent, if applicable.

Status

A value from the `DocumentStatusType` typelist, such as Final or Draft. You are required to set this value when you create a document. In the base configuration, only Final and Draft are used. The Approving and Approved statuses are not used in the base configuration, but you can implement code that uses them.

Security type

A value from the `DocumentSecurityType` typelist. The default values are Sensitive Document and Unrestricted Document. For example, a document related to a special investigation might be sensitive and might require extra restrictions on users who can view and edit the document.

Document type

A value from the `DocumentType` typelist that classifies the document, such as Email Sent, Letter Received, Letter Sent, Service Level Agreement, or W9. This property is required.

Hidden

Indicates if the document is hidden or visible. This property is required.

See also

- "Edit metadata properties of a vendor document" on page 182
- "List vendor documents in ContactManager" on page 179
- "Indicate the existence of a hard-copy vendor document" on page 181
- "Setting document configuration parameters and MIME types" on page 184

Working with vendor documents

To work with vendor documents, open an existing vendor contact and click the **Documents** card.

To open an existing vendor contact, click the **Contacts** tab and search for contacts with **Tags** set to `Vendor`. Then select the contact in the search results list.

List vendor documents in ContactManager

About this task

You can list the documents associated with a vendor contact.

Procedure

1. Log in to ContactManager and click the **Contacts** tab.
2. In the **Search** screen, set **Tags** to `Vendor`.
3. Set other search criteria, such as setting **Contact Type** to `Contact` and entering the name of the contact you want to find.
4. Select the contact from the **Search Results**.
5. On the details screen for the contact you selected, click the **Documents** card.
6. All documents attached to this contact are listed on this card.

Results

With the documents listed in the **Documents** card, you can download, view, edit, upload, and remove documents. See "Working with the documents card" on page 179.

Working with the documents card

To open the **Documents** card, see "List vendor documents in ContactManager" on page 179.

In the list of documents, you can:

- Click a document **Name** to download the document and view its contents.
  - If the browser can open the document for viewing, a window opens showing the contents.
  - If the browser cannot open the document for viewing, you see a message saying that the file was downloaded for viewing, and you can open it with the appropriate viewer.

Note: If nothing happens when you click the document name, enable pop-ups for ContactManager in your browser.

- Click View Document Properties to see the document's metadata properties on the **Document Properties** screen. On that screen you can edit the properties, download the document content, upload new content, or remove the document from the contact.
- Click Download to download, view, and possibly edit the document's content.
- Click Upload to upload new or edited content.
- Click Remove Document to detach the document from this contact. Removing the document does not delete it from the DMS.

Create a new vendor document

Before you begin

To open the **Documents** card, see "List vendor documents in ContactManager" on page 179.

About this task

You can create a new vendor document when you have a contact open. You can work with documents at any time, whether or not you are editing the contact.

Procedure

1. Navigate to the **Documents** card.
2. Click **New Document**, and then click one of the following:
   - **Upload document**
   - **Indicate existence of a document**

Uploading documents

When you upload a document, you replace the content for a document with a file from your file system. If you are creating a new document, you must specify metadata properties for the document, and the upload becomes the content. You can upload multiple documents at one time.

Upload new documents

Before you begin

To open the **Documents** card, see "List vendor documents in ContactManager" on page 179.

Procedure

1. On the details view for a contact you selected, click the **Documents** card.
2. Click **New Document** > **Upload documents**.
3. To add files that you want to upload, you can:
   - Drag one or more files from your file system window, such as Windows Explorer, to the worksheet.
   - Click **Add Files**, browse to the locations of your documents, and click **Add**.

   You can click **Add Files** multiple times for files in different folders. You can also select more than one document in a folder.
4. Set the properties for the files you want to upload.

- You can set the properties one file at a time in the fields to the right of each file you added to the list.
- You can edit the properties for multiple files by selecting their check boxes and then clicking **Edit Details**.

You must have values for the **Name**, **File Type**, **Status**, **Document Type**, and **Hidden** fields.

IMPORTANT: Do not set the **Name** field for multiple files. Files must have different names. Additionally, ContactManager sets the file type for you based on the MIME type it detects. If you set the **File Type** field, the file will be uploaded in that format regardless of the file type indicated by its contents.

5. Click **Upload** to send the file or files to the server and, for new files, attach the document to the contact.

Replace content for an existing document

Before you begin

To open the **Documents** card, see "List vendor documents in ContactManager" on page 179.

About this task

If a document already exists, you can replace its content.

Procedure

1. You can start the upload to replace a document's content in two ways:
   - On the **Documents** card, for the document whose contents you want to upload, click Upload under **Actions**.
   - On the **Documents** card, click View Document Properties under **Actions**. Then, on the **Document Properties** screen, click Upload.
2. In the **Update Document Content** screen, add the file that has the new content by:
   - Browsing for the content file.
   - Dragging the file from your file system viewer, such as Windows Explorer.
3. Click **Update**.

Indicate the existence of a hard-copy vendor document

If you keep a vendor document as hard copy instead of scanning the hard copy to produce a content file, use this option to describe the document in ContactManager.

Before you begin

To open the **Documents** card, see "List vendor documents in ContactManager" on page 179.

About this task

Because the document is hard copy, it has only metadata properties stored on the system and no associated content file. You have to go to your file cabinet or other storage location to retrieve the document.

This option gives you all the document attribute fields that you have for electronic documents except **File Type**. The properties you set can appear in searches. See "Document metadata properties" on page 178.

Procedure

1. With a contact open, click the **Documents** card and then click **New Document** > **Indicate existence of a document**.
2. The **New Document** screen that opens enables you to set metadata properties for the document. Enter attributes that describe the hard copy document sufficiently to enable a user to find it.
3. Click **Update** to add the document describing the hard copy document to the database.

Edit content for a vendor document

Before you begin

To open the **Documents** card, see “List vendor documents in ContactManager” on page 179.

About this task

You can edit contents of documents in the **Documents** card for a contact. Your user name must have a role with permissions that enable you to edit the content of a document.

Procedure

1. On the **Documents** card for a contact, click Download in the **Actions** column for the document. Alternatively, you can click the same button on the **Document Properties** screen for the document.
   Your browser indicates that it downloaded the file.
2. Edit the document content file in the appropriate editor.

   Most web browsers can be configured to open some types of downloaded files in their native editors.
3. Save your work after you have made all your edits.
4. Make note of the saved file name and location so you can browse for the file when you upload changes to the document. The file you upload becomes the new content for the document.
5. In the **Documents** screen, click **Upload** under **Actions**.
6. On the **Update Document Content** screen, click **Browse**, locate the file you saved, and then click **Update**. Alternatively, you can drag the file from your file system viewer to this screen.

Edit metadata properties of a vendor document

Before you begin

To open the **Documents** card, see “List vendor documents in ContactManager” on page 179.

About this task

If you have sufficient permissions, you can edit metadata properties of documents in the **Documents** card of a contact. These metadata properties are described at “Document metadata properties” on page 178.

Procedure

1. On the **Documents** card of a contact, click View Document Properties in the **Actions** column for the document.
2. On the **Document Properties** screen, click **Edit**.
3. Make your changes, and then click **Update**.

   Note: If you change the **Name** field, ContactManager uses that name in the future for the file it downloads for document content.

Hide a vendor document

Before you begin

To open the **Documents** card, see “List vendor documents in ContactManager” on page 179.

About this task

Hiding a document is a way to remove an obsolete document from your list of documents without removing it. When you hide a document, you no longer see it listed on the **Documents** card unless you indicate that you want to see hidden documents.

> **Note:** Hiding a document is not the same as removing it. Removing a document detaches it from the current contact, and it must be attached again for you to be able to see it. Only users with appropriate permissions can remove documents.

Procedure

1. On the **Documents** card of a contact, click View Document Properties in the **Actions** column for the document.
2. On the **Document Properties** screen, click **Edit**.
3. Set the **Hidden** property to `Yes`.
4. Click **Update**.

Results

Hiding a document sets the `Obsolete` flag on the `Document` entity and does not retire the document in the database. You can view hidden documents by setting the filter in the **Documents** card to **All Documents** or **Hidden Documents Only**. The default value of this filter, **Only Current Documents**, does not show hidden documents.

Remove a vendor document link

Before you begin

To open the `Documents` card, see “List vendor documents in ContactManager” on page 179.

About this task

If you have a role with the `docdelete` permission, you can remove the link between a document and a contact. If you can remove a document, the Remove Document action is available in the document’s **Actions** column. When you remove a document, you detach the document from a vendor contact. Removing a document does not necessarily delete it. See also “Hide a vendor document” on page 182.

Procedure

1. On the `Documents` card of a contact, click Remove Documents in the **Actions** column for the document.
2. In the confirmation dialog, click **OK**.
   The document is no longer visible in the list of documents.

Configuring vendor document management

The base configuration of ContactManager provides Document Management system permissions and configuration parameters that you can configure or manage as an administrator. Additionally, there are base configuration PCF files, classes used internally in ContactManager with contacts and documents, and entity relationships, all of which are configurable.

> **IMPORTANT:** Guidewire recommends integrating with an external document management system rather than using the default demonstration document management system on the ContactManager server. The default system is useful only for demonstration purposes and does not support features of a real document management system, such as document versioning.

See also

- "Vendor document management integration" on page 188
- For information on ClaimCenter document management configuration, see the *Integration Guide*.

Document security

ContactManager provides a set of system permissions to provide security for all documents. You can also use these permissions to define security types for documents and assign permissions to users that relate to these security types.

Note: The `RestrictSearchesToPermittedItems` search parameter in the `config.xml` file determines how permissions work with search results. The parameter determines if you can see a document that you do not have permission to view in the list of documents attached to a contact.

The following system permissions provide security for documents.

Name	Purpose of permission
`doccreate`	Add documents for a contact.
`docdelete`	Remove documents from a contact. Removing a document detaches it from the contact.
`docedit`	Edit documents.
`docview`	View the documents on a contact.

See also

- "Contact search security configuration parameters in ContactManager" on page 92
- For information on access control for exposures, see the *Application Guide*.

Setting document configuration parameters and MIME types

A set of document configuration parameters in the `config.xml` file control the display and editing of files in a document management system. The default values support using the file system on the application server.

In addition to these configuration parameters, there is a section of `config.xml` for defining MIME types, used in a document's File Type property.

Vendor document configuration parameters

The following document configuration parameters in the `config.xml` file control the display and editing of files in a document management system.

Parameter	Description
`DisplayDocumentEditUploadButtons`	Set to `true` to display the Download and Upload buttons, which enable editing and returning edited files to the document management system. Set to `false` if `IDocumentContentSource` is disabled or does not support editing. The default value of this parameter is `true`.
`DocumentContentDispositionMode`	How to display a retrieved document. Set to `inline` to have it appear in a browser. Set to `attachment` to have it opened by its editor, as determined by its MIME type. The default value of this parameter is `inline`.
`FinalDocumentsNotEditable`	When `false`, the default value, this parameter indicates that documents with Final status can be transferred when using the asynchronous implementation of the `IDocumentContentSource` plugin.
`MaximumFileUploadCount`	The maximum number of document content files that can be listed and uploaded at once. The number of files listed for upload can affect the performance of the upload screen.

Parameter	Description
	The default number of files is 200.
`MaximumFileUploadSize`	The maximum allowable size in megabytes for a document file that you can upload to the server. Attempting to upload a file larger than this size results in failure. Because the uploaded document must be handled on the server, this parameter protects the server from possible memory consumption problems. The default value is 25 MB.
`MaximumTotalUploadSize`	The maximum allowable total size in megabytes per session for documents uploads that are pending commitment to the document management system. Because multiple documents can be uploaded at once, this value also provides control of the overall size of an upload and protects the server from possible memory consumption problems. The default value is 25 MB.
`ReCallDocumentContentSourceAfterRollback`	If `true`, if a document has been rolled back, on commit you must call both `isDocument` and `addDocument` again. The default value of this parameter is `false`.

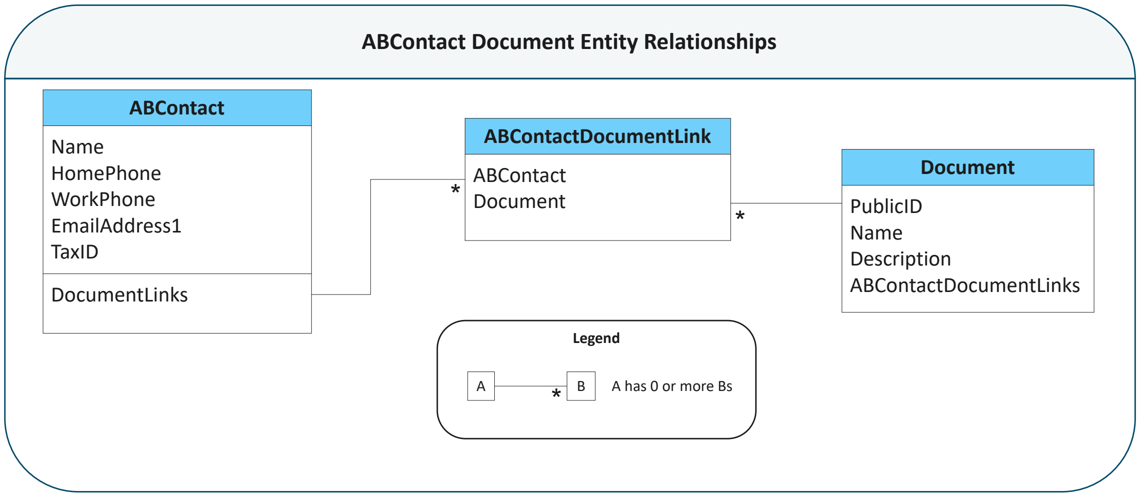
Configuring vendor document MIME types

To configure document file types, also called MIME types, edit the `config.xml` file. A section of `config.xml` maps document MIME types to icons in the user interface and to file extensions. For example:

```
<mimetypemapping>
  <mimetype name="application/msword"
            icon="word_mime"
            extensions=".doc"
  <!-- more mappings -->
</mimetypemapping>
```

ABContact and Document entity relationships

ContactManager can connect an array of documents to an `ABContact`. After you add documents, the contact references the documents as an array, as shown in the following figure.



Note: In the base configuration, a method called by the user interface checks that a contact has a Vendor tag before a document can be attached to it. This behavior is configurable.

See also

- For information on the method supporting the Vendor tag requirement, see “Contact details view PCF file” on page 186.

Working with vendor document PCF files

In ContactManager a user with appropriate permissions can perform a number of actions with documents, such as:

- Viewing all documents
- Attaching a document to a contact
- Removing a document from a contact
- Viewing and editing document metadata properties
- Downloading document contents
- Uploading new or changed document contents

These actions are available on the **Documents** card of the contact detail view.

See also

- "Working with vendor documents" on page 179

Contact details view PCF file

The **Documents** card that supports actions on documents is part of the detail view for a contact. By default, when you choose a contact from the search results, the **Basics** card for the contact opens. The PCF file containing these cards is `ABContactDetailScreen.pcf`, and the actions and documents are in the **Documents** card.

This PCF file defines a variable, `contactDocumentsHelper`, that has class type `gw.web.ContactDocumentsActionsHelper`. This class helps to determine things like:

- Whether to show the **Documents** card for the contact. For example, `ShowDocumentsTabForContact` determines if the contact has the Vendor tag set and the user has the proper permissions.
- If the `IDocumentContentSource` plugin is enabled and available to communicate with the DMS, in `DocumentContentServerAvailable`. The PCF variable `contentActionsEnabled` uses this property.
- Whether to show document content actions based on availability of the document content server, in `isDocumentContentActionsAvailable`. The PCF variable `contentActionsAvailable` uses this method.
- Whether to show document metadata actions based on availability of document metadata, in `isDocumentMetadataActionsAvailable`.
- Whether to show specific actions, such as the **Upload** button or the **Download** button, as defined in `isUploadDocumentContentAvailable` and `isDownloadDocumentContentAvailable`.

Documents card widget

The `DocumentsCard` widget in `ABContactDetailScreen.pcf` displays documents, actions, and buttons like **New Document** and **Remove Selected**.

- The toolbar button `AddDocuments` displays a menu with two choices:

  `IndicateExistence`

  Indicate the existence of a document. Uses the `goInWorkspace` method in the class `pcf.IndicateExistenceDocumentWorksheet` to open the `IndicateExistenceDocumentWorksheet.pcf` worksheet in the Contact workspace. In this worksheet you can add documents to the contact that indicate existence of a hard-copy document.

  `AttachDocument`

  Upload documents. Uses the `goInWorkspace` method in the class `pcf.AttachDocumentWorksheet` to open the `AttachDocumentWorksheet.pcf` worksheet in the Contact workspace. In this worksheet you can add documents to be attached to the contact.

- The toolbar button `RemoveDocuments` calls the method `contact.unlinkDocumentsForUI`, which is defined in the enhancement `GWContactDocumentEnhancement.gsx`.

Note: There is also a **Refresh** button that is available only in asynchronous mode. Use this button to indicate pending documents waiting to be stored in the DMS.

- The list view `ContactDocumentsLV` displays all the documents linked to the current contact.
  - The link cell `Name` displays the name of the document as a link to its content. The name itself is a Link widget.
  - The link cell `Actions` displays the actions available for the document, such as View/Edit document properties, Upload, Download, and Remove.

    Each of the actions is a `Link` widget that performs an appropriate action. For example, the first link, `ViewPropertiesLink`, opens `DocumentPropertiesPopup.pcf`, the edit screen for the document's metadata properties.

See also

- "List vendor documents in ContactManager" on page 179
- "Create a new vendor document" on page 180
- "Indicate the existence of a hard-copy vendor document" on page 181
- "Remove a vendor document link" on page 183

Document permission check expressions

In a page configuration file (PCF), you can control permissions on specific widgets with Gosu expressions that determine if a user has permission to perform an operation.

For example, in the `AddDocuments` toolbar button on the `DocumentsCard` of the `ABContactDetailScreen.pcf` file, the `visible` attribute is set to:

```
perm.Document.create and !isReadOnly
```

The first expression is a permission check expression. It allows access only to users in a role with the permission to upload new document contents, the `doccreate` permission.

Permission check expressions can also be used in Gosu code, such as in the class `gw.web.ContactDocumentsActionsHelper`.

The Document permission check expressions are defined in the following classes:

- `perm.System`
- `perm.Document`

The following table lists the document permission check expressions available in ContactManager:

ContactManager Expression	Description
`perm.Document.create`	Optionally takes a `Document` as a parameter. Verifies that the user has the permission to upload new document contents, the `doccreate` permission.
`perm.System.doccreate`	Verifies that the user has the permission to create a document for a contact, the `doccreate` permission.
`perm.Document.delete`	Takes a `Document` as a parameter and verifies that the user has the permission to remove the document, the `docdelete` permission.
`perm.System.docdelete`	Verifies that the user has the permission to remove a document from a contact, the `docdelete` permission.
`perm.Document.edit`	Takes a `Document` as a parameter and verifies that the user has the permission to edit the document, the `docedit` permission.
`perm.System.docedit`	Verifies that the user has the permission to edit a document on a contact, the `docedit` permission.
`perm.Document.view`	Takes a `Document` as a parameter and verifies that the user has the permission to view the document, the `docview` permission.

ContactManager Expression	Description
`perm.System.docview`	Verifies that the user has the permission to view contact documents, the `docview` permission.

See also

- "Document security" on page 184
- "ContactManager contact and tag permission check expressions" on page 90

Vendor document classes

The following classes and files provide methods used in ContactManager to work with `Document` objects.

`gw.entity.GWContactDocumentEnhancement.gsx`

Extension methods for `ABContact` to unlink documents from a contact.

`gw.web.ContactDocumentsActionsHelper`

Methods used in the PCF files, particularly in `ABContactDetailScreen.pcf`, to determine what to display. For example, there are methods to determine whether to show the **Documents** card or when to show actions for a document.

> **Note:** This class is also called from `DocumentPropertiesPopup.pcf` and `ReviewDuplicateContactsPopup.pcf`.

`gw.document.DocumentCMContext`

Methods that initialize document properties, link a list of documents to a contact, and remove a list of documents from a contact.

`gw.ab.document.DocumentsUtil`

Provides basic methods for creating documents, such as `createDocumentCreationInfo` and `initDocumentExistence`. Also `getDocumentsFor` gets a list of documents.

`com.guidewire.ab.domain.addressbook.impl.ABContactImpl`

A Java class that, for documents, provides methods used by Gosu classes like `DocumentCMContext`. For example, there is a single document version and a list of documents version of the `addToDocuments` method, and there are `getDocuments` and `removeFromDocuments` methods.

See also

- "Contact details view PCF file" on page 186

Vendor document management integration

There are two types of integrations you can perform with vendor documents:

- "Vendor document integration with a document management system" on page 188
- "Vendor document integration with ClaimCenter" on page 190

Vendor document integration with a document management system

External integration with a document management system (DMS) involves writing plugin implementation classes for the document management plugins. This integration can also involve enabling asynchronous event fired messaging rules.

Vendor document management plugins

The following are the main plugin interfaces used to integrate ContactManager with a document management system. In the base configuration, the plugins work with the demonstration document management system. Each plugin interface has a default plugin implementation class.

Interface	Description
`IDocumentMetadataSource`	ContactManager passes metadata properties to the plugin implementation class registered in this plugin registry. The class searches its metadata and returns a list of documents found. In the base configuration, this plugin is not enabled, and the following plugin implementation class is registered: `gw.plugin.document.impl.ServletBackedDocumentMetadataSource` You can implement your own plugin implementation class to interface with a system for storing document metadata, such as name, id, status, author, and so on. If the plugin is not enabled, then the ContactManager stores the metadata in its database. This interface is separate from `IDocumentContentSource` because of different architectural requirements.

`IDocumentContentSource`	ContactManager passes to the plugin implementation class registered in this plugin registry the metadata for one document. The class, which returns the document content, does the following:
	• Interfaces with a document storage system.
	• Contains methods for creating, updating, and retrieving document contents.
	• Supports the following document retrieval modes:
	◦ Document contents.
	◦ Gosu executed by client rules.
	◦ URL to a server content store.
	• In the base configuration, the following plugin implementation class is registered:
	`gw.plugin.document.impl.AsyncDocumentContentSource`
	In the registry, the parameter `TrySynchedAddFirst` is set to `true` and `SynchedContentSource` is set to:
	`gw.plugin.document.impl.ServletBackedDocumentContentSource`
	These parameter values cause the class to first try to use synchronous document management. If it fails, then it uses asynchronous document management.

See also

- For information on asynchoronous document storage and on using document management with an external document management system, see the *Integration Guide*.

ContactManager classes for external integration

External integration with a DMS is described in detail for ClaimCenter document management in the *ClaimCenter Integration Guide*. The primary differences for ContactManager are the names of the classes you can use as a starting point:

ClaimCenter class	ContactManager class
`LocalDocumentContentSource`	`ServletBackedDocumentContentSource`
`LocalDocumentMetadataSource`	`ServletBackedDocumentMetadataSource`

See also

- ContactManager plugins are described at “Vendor document management plugins” on page 188.
- For information on external document management integration, see the *Integration Guide*

Enabling asynchronous event fired document messaging

ContactManager has two event fired messaging rules that can be used with asynchronous document storage on a DMS. These rules are enabled in the base configuration, and they use a message destination and plugin.

Note: These rules and the plugin use sample code that is for demonstration purposes only. There is further work required to integrate ContactManager document management with an external DMS system.

The rules are in the **EventMessage** rule category in the **AsyncDocument** rule set:

Async Document Storage
Sends a Document Store message to message destination 324, the document store transport destination.

Async Document Storage Failed
A sample rule that does error handling if there is a problem with sending the document to the DMS.

The message destination for these two rules is destination ID 324. It is enabled in the base configuration. To edit this destination, open `messaging-config.xml`.

In the plugin registry `documentStoreTransport.gwp`, the class that implements the `documentStoreTransport` plugin in the base configuration is `gw.plugin.document.impl.DocumentStoreTransport`. This class is sample code that works with the demonstration DMS. You must implement a plugin implementation class that works with your external DMS.

See also

- For information on configuring ContactManager event messaging, see "EventMessage rule set category" on page 207.
- For more complete information on configuring asynchronous messaging in ClaimCenter with an external DMS, see the *Integration Guide*

Vendor document integration with ClaimCenter

The integration with ClaimCenter supports retrieving read-only information about documents associated with vendor contacts from ContactManager. You cannot edit vendor documents or attach documents to vendor contacts in ClaimCenter, as you can in ContactManager, but you can see them.

Note: ClaimCenter supports document management for claims and entities associated with claims, including claim contacts and services. That document support is claim-based and does not support attaching documents to vendor contacts independently of their association with a claim. See the *Application Guide*.

ContactManager web service for retrieving contact documents

ContactManager provides a web service method, `ABContactAPI.retrieveDocumentsForContact`, that ClaimCenter uses to retrieve information about contacts. This method's parameters have the following types:

- `gw.webservice.ab.ab1000.abcontactapi.ABContactAPIDocumentSearchCriteria`
- `gw.webservice.ab.ab1000.abcontactapi.ABContactAPIDocumentSearchSpec`

The method returns a search result container of the following type:

```
gw.webservice.ab.ab1000.abcontactapi.ABContactAPIDocumentSearchResultContainer
```

The search result container has an array of `ABContactAPIDocumentInfo` objects, each providing information about a document, including document properties like its name, description, and the URL for the document content. The path for this class is:

```
gw.webservice.ab.ab1000.abcontactapi.ABContactAPIDocumentInfo
```

IMPORTANT: ContactManager cannot make use of extensions to the search classes to support custom search criteria.

See also

- "ABContactAPI methods" on page 303

ClaimCenter retrieval of vendor contact documents

In the base configuration, ClaimCenter uses the method `ABContactSystemPlugin.retrieveDocumentsForContact` to retrieve information on the documents associated with a vendor contact. This method is called when the user selects a vendor contact from the **Address Book** tab, on the **Parties Involved** screen, or in the New Claim wizard.

ClaimCenter uses the contact's `AddressBookUID` to identify the contact for ContactManager, and it also determines if the user has permissions to see documents that have security restrictions. ClaimCenter sends the list of `DocumentSecurityType` objects specifying security permissions for the documents the user can see to ContactManager in the `SearchCriteria`. ContactManager filters the documents based on that set of `DocumentSecurityType` objects.

ClaimCenter then displays information about the documents on the **Documents** card of the contact's detail view.

The ClaimCenter implementation of `ABContactSystemPlugin.retrieveDocumentsForContact` uses the following classes to prepare data for the search sent to ContactManager:

`gw.plugin.contact.ab1000.ContactDocumentsSearchMapper`

Converts ClaimCenter search criteria into ContactManager search criteria. Use this class to map vendor document search criteria to those required by the ContactManager class `ABContactAPIDocumentSearchCriteria`.

`gw.plugin.contact.ab1000.ContactDocumentsHelper`

Sets the search specifications used by the ContactManager class `gw.webservice.ab.ab1000.abcontactapi.ABContactAPIDocumentSearchSpec`.

The ClaimCenter method `retrieveDocumentsForContact` also uses the method `ContactDocumentsHelper.toSearchResults` to format the array of document search results returned from ContactManager. This method uses `gw.plugin.contact.ab1000.ContactDocumentsSearchResultMapper` to format the search results as a `ContactDocumentInfo` object, which ClaimCenter can display in the vendor contact detail view's **Documents** card.

Note: The amount of customization you can do in this search is limited to modifying the existing search criteria and search specifications. Even if you add new criteria both in ContactManager and in these ClaimCenter methods, ContactManager cannot use them.

See also

- "Document metadata properties" on page 178
- For information on the ClaimCenter contact search screen and ClaimCenter document security, see the *Application Guide*.

chapter 10

Linking and synchronizing contacts

You can set up links for contacts and synchronize contact information between Guidewire core applications and the Guidewire ContactManager™ contact management system. You must be managing contacts in an environment where ContactManager is integrated with your Guidewire core applications.

> **Note:** This topic covers synchronizing contact data between a Guidewire core application and ContactManager, which is more complex and configurable in ClaimCenter than it is in PolicyCenter or BillingCenter. From the point of view of the core application, this type of synchronization is *external contact synchronization*. PolicyCenter also defines behavior for synchronizing contact data internally between accounts and policies. That type of synchronization is distinct from external contact synchronization.

See also

- “Locally and centrally managed contacts” on page 14

Linking a contact

Before contact information can be synchronized between ContactManager and a Guidewire core application, it must be linked. Linking ensures that a contact stored locally in the core application is connected to a contact stored in ContactManager. ContactManager uniquely identifies an `ABContact` instance by its `LinkID`. The core applications use `AddressBookUID` for the same purpose. A contact is linked when the core application has verified the contact’s `AddressBookUID` with ContactManager.

Creating and linking a contact

The initial linking of a core application contact and a ContactManager contact typically happens when the core application creates the contact and sends it to ContactManager. In the base configuration, PolicyCenter specifies the unique ID for a new contact, while ClaimCenter and BillingCenter let ContactManager specify the unique ID for a new contact. In all three core applications, `AddressBookUID` holds the unique ID for the contact.

- In the base configuration, ClaimCenter and BillingCenter send the `createContact` request to ContactManager, and ContactManager creates a unique `LinkID` for the contact. ContactManager then returns the new contact data, including the `LinkID`, to the calling application. ContactManager also broadcasts the new contact data to the other applications.
- In the base configuration, PolicyCenter generates a unique ID for the new contact and sends the `createContact` request to ContactManager. ContactManager populates `LinkID` for the new contact with the unique ID specified by PolicyCenter.

See also

- “Linking in ClaimCenter” on page 195
- “Linking in BillingCenter” on page 196
- “PolicyCenter contact creation with external unique IDs” on page 194
- “Linking in PolicyCenter” on page 196

PolicyCenter contact creation with external unique IDs

PolicyCenter must be able to create a new contact and send it both to BillingCenter and ContactManager. PolicyCenter must be able to uniquely identify a contact it sends to BillingCenter, and BillingCenter must be able to link to the contact created in ContactManager. Therefore, in the base configuration, PolicyCenter specifies a unique identifier for any contact it creates and sends to ContactManager.

If PolicyCenter needs to notify BillingCenter that the new contact has been created, PolicyCenter also sends a message to BillingCenter indicating the `AddressBookUID` of the new contact. The value of the `AddressBookUID` is the unique identifier that PolicyCenter created for the contact.

These messages are asynchronous and can arrive at BillingCenter and ContactManager at different times and in no particular order.

When ContactManager receives a contact create request for which a unique ID is specified, it uses that ID for the `LinkID` value of the new contact. ContactManager then sends create messages for update to BillingCenter and ClaimCenter if they are registered.

Depending on when BillingCenter receives the messages, it proceeds down different paths to link the contact:

- BillingCenter receives the message from PolicyCenter before ContactManager creates the new contact.
  1. BillingCenter sends a request to ContactManager for contact data, specifying the `AddressBookUID`.
  2. ContactManager returns `null`, indicating that the contact does not exist.
  3. BillingCenter creates a local copy of the contact with the `AddressBookUID` it received from PolicyCenter.
  4. ContactManager later notifies BillingCenter that the contact was created.
  5. BillingCenter verifies that the `LinkID` sent by ContactManager matches the `AddressBookUID` of the local contact that it created.
  6. BillingCenter updates its local copy of the contact with the data from ContactManager and links the contact. See “Linking in BillingCenter” on page 196.
- BillingCenter receives the message from ContactManager about the new contact before it receives the message from PolicyCenter.
  1. BillingCenter ignores the message from ContactManager, because it does not have a matching contact.
  2. BillingCenter receives a message from PolicyCenter saying that there is a new contact with a specified `AddressBookUID`.
  3. BillingCenter sends a request to ContactManager for contact data, specifying the `AddressBookUID`.
  4. ContactManager sends the data for the contact whose `LinkID` matches the `AddressBookUID` sent by BillingCenter.
  5. BillingCenter updates its local copy of the contact with the data from ContactManager and links the contact. See “Linking in BillingCenter” on page 196.

See also

“PolicyCenter support for creating external unique IDs” on page 195

ContactManager support for handling external unique IDs

ContactManager provides support for handling external unique IDs as follows:

- The ContactMapper class in ContactManager has field mapping definitions for the External_UniqueID fields in each entity that implements ABLinkable. These fields are listed in the topic "Mapping externally specified unique IDs of a ContactManager contact" on page 315.
- If a createContact request comes in that specifies an External_UniqueID field for the new contact, ContactManager uses the value of that field to populate the LinkID for the new contact. If that field is not in the data for the new contact or the field exists but its value is null, ContactManager creates its own LinkID for the new contact. See the description of the createContact method in "ABContactAPI methods" on page 303.
- ContactManager provides a CreatedApproved rule in the EventFired ruleset for each core application. The rule is fired by the event ABContactCreatedApproved, which is added to an ABContact when it is created with an approved status by a call to ABContactAPI. The rule causes ContactManager to broadcast a create message as an update to each configured core application. This message enables any application that has been notified of a new contact that has an external ID provided to ContactManager to receive an update message when the create happens.

  For example, PolicyCenter sends a new contact to BillingCenter with the AddressBookUID specified. When ContactManager creates the new contact, BillingCenter gets the Create for Update message from ContactManager. BillingCenter can then compare the LinkID in the message with the AddressBookUID of the contact that PolicyCenter sent to BillingCenter and update the contact with the data from ContactManager.

PolicyCenter support for creating external unique IDs

PolicyCenter provides support for creating external unique IDs with the following:

- The Contact entity in PolicyCenter is extended with the field ExternalID, which stores the value of the unique ID that PolicyCenter generates.
- The ContactMapper class in PolicyCenter has field mapping definitions for various EXTERNAL_UNIQUE_ID fields. Those definitions are described in "PolicyCenter mapping of externally specified unique IDs" on page 320.
- PolicyCenter has code that generates a new unique ID and assigns it to the ExternalID field of a new contact. PolicyCenter then calls the ABContactAPI.createContact method, passing it the data for that new contact. See the class gw.api.contact.PCContactLifecycle.
- PolicyCenter has an EventFired rule that responds to a ContactAdded event and sends a newly created contact to BillingCenter.

Linking in ClaimCenter

ClaimCenter links a vendor contact automatically with ContactManager if the contact is created by a user that has the permissions abcreate or abedit. If the user does not have these permissions, in the base configuration, new vendor contacts are created, flagged as pending in ContactManager, and linked. These pending creates must be approved by a ContactManager user. After approval, ContactManager removes the pending flag.

This ClaimCenter behavior with vendor contacts is defined in the Gosu class gw.plugin.contact.ContactSystemApprovalUtil. You can edit this class and change how ClaimCenter determines the following:

- If a contact created in ClaimCenter will be created in ContactManager
- If a contact created or updated in ClaimCenter and sent ContactManager requires approval

In ClaimCenter, you can also manually link a local contact to ContactManager. For example, on the **Parties Involved** screen of a claim, you can click the name of a local contact and, below it in the **Basics** tab, click **Link**. After you click **Link**, ClaimCenter calls the ContactManager findDuplicates web service to see if there is a matching contact.

> **Note:** In the base configuration, this link functionality uses the WIDE_MAP variable to try to find a matching contact. See "IFindDuplicatesPlugin plugin interface" on page 328.

Depending on the results of the findDuplicates call, ContactManager and ClaimCenter behave as follows:

- If ContactManager finds an exact match, ClaimCenter links the local contact to the ContactManager contact by populating the local contact's AddressBookUID with the LinkID of the ContactManager contact.

- If ContactManager finds potential matches, ClaimCenter enables you to select one of the potential matches.
- If ContactManager does not find any matches, its behavior depends on the permissions of the ClaimCenter user and the type of contact:
  - If the ClaimCenter user has the `abcreate` permission, ClaimCenter sends a request to ContactManager to create a new contact.
  - If the ClaimCenter user does not have the `abcreate` permission and the contact is a vendor contact, ClaimCenter sends the new vendor to ContactManager with pending status. This new vendor must be approved by a user logged in to ContactManager.
  - If the contact is not a vendor contact, such as `Person`, ClaimCenter sends a request to ContactManager to create the new contact. The ClaimCenter user does not have to have `abcreate` permission.

After a successful link, the contact becomes subject to synchronizing rules, and the **Link** button turns into an **Unlink** button. If you click **Unlink**, the contact is no longer connected to ContactManager. It becomes a local-only contact.

See also

- "Synchronizing ClaimCenter and ContactManager contacts" on page 201
- "Find duplicates behavior" on page 197
- For additional information on working with contacts in ClaimCenter, see the *Application Guide*

Linking in PolicyCenter

In the base configuration, PolicyCenter links contacts with an integrated ContactManager under the following conditions:

- PolicyCenter always links contacts that are on accounts with at least one bound policy. At the time the user adds the contact, PolicyCenter checks to see if there are possible duplicate contacts.
- If the contact initially comes from ContactManager, PolicyCenter links the local contact when it creates the local contact. For example:
  - If you search for contacts and add one that is in ContactManager, that contact becomes linked regardless of the state of the policies on the account.
  - If you click to add a new contact, ContactManager can return a list of exact and potential duplicates. If you choose one of the duplicates as the contact that you want to use, that contact is linked.

PolicyCenter does not send new client data to ContactManager while the client is still a prospect—when there is no bound policy on the account. PolicyCenter does store prospect information locally. If a prospect becomes a customer—at least one policy is bound for the account—PolicyCenter links the contact to the contact management system. You can configure this behavior differently.

The `Contact` entity and its subentities have an `AutoSync` property that controls synchronization with ContactManager. Setting this property to `Allow` enables the contact to be synchronized. After a contact is linked, PolicyCenter ensures that the `AutoSync` property is set to `Allow`.

See also

- "Find duplicates behavior" on page 197
- "Configuring PolicyCenter external contact synchronization" on page 199

Linking in BillingCenter

In the base configuration, BillingCenter links contacts with an integrated ContactManager under the following conditions:

- BillingCenter links contacts that are on accounts and policy periods. At the time the user adds the contact, BillingCenter can check to see if there are possible duplicate contacts.

    - If you click **Add Existing Contact** for an account or policy period and search for a contact, BillingCenter shows a list of matching internal and external contacts.
        - If you add an external contact, one that is stored only in ContactManager, BillingCenter creates a local instance of the contact and links it.
        - If you add a contact that is not external, the contact has a local instance that BillingCenter might already have linked. If the contact instance is not linked, BillingCenter saves it as a new contact in ContactManager and links it.
    - If you click to add a new contact, BillingCenter checks to see if it exists in ContactManager. This check can return a list of exact and potential duplicates. If you choose one of the duplicates as the contact that you want to use, that contact is linked. If you create the contact without choosing from the list, BillingCenter sends it to ContactManager as a new contact, and at that point it gets linked.
- For producer contacts, BillingCenter creates a local instance of the contact. It does not link producer contacts with ContactManager.

BillingCenter find duplicates behavior is defined in the method `gw.plugin.contact.ab1000.ABContactSystemPlugin.findDuplicates`. For example, this method specifies that the only tag to be used to find duplicates is the client tag. If you want to use other tags for finding duplicates, you can modify this method to do so.

The `Contact` entity and its subentities have an `AutoSync` property that affects synchronization with ContactManager. Setting this property to `Allow` for a contact of an account or policy period enables the contact to be synchronized. BillingCenter sets the `AutoSync` property to `Allow` for all new contacts.

See also

- “Find duplicates behavior” on page 197
- “Configuring BillingCenter external contact synchronization” on page 200

Find duplicates behavior

In their base configurations, the Guidewire core applications call the ContactManager web service `ABContactAPI.findDuplicates` to see if there are duplicates for a contact. The match types that ContactManager can return are *exact match* and *potential match*. For example:

- ClaimCenter calls this API when setting up a link for a contact. If there is an exact match, ContactManager establishes the link automatically. If the match is potential, ContactManager sends the potential matches to ClaimCenter, which displays them and enables you to choose one. Linking proceeds as described in “Linking in ClaimCenter” on page 195.
- Both PolicyCenter and ClaimCenter call this API when you click the **Check for Duplicates** button while adding a new contact. They also call this API when you save a new contact and you have not previously clicked **Check for Duplicates**.
    - If you click **Check for Duplicates**, ContactManager checks for exact and potential matches and sends the results to the core application, PolicyCenter or ClaimCenter. The core application shows the results and enables you to choose one.
    - If you click to save a new contact and have not clicked **Check for Duplicates**, the core application and ContactManager participate in the find duplicates exchange as described previously.
    - If you click to save a new contact and have already clicked **Check for Duplicates**, there is no additional find duplicates check. However, if there is an exact match that you have not chosen, the core application shows a warning that an exact match was available.

      You must confirm one of the following:
        - If you want to create a new contact, confirm that you do not want to choose the exact match.
        - If you do not want to create a new contact, click to cancel the operation.
- BillingCenter calls this API when you add a new contact to an account or policy period, as follows:

- The first time you click to save a new contact, BillingCenter and ContactManager participate in a find duplicates exchange. ContactManager checks for exact and potential matches and sends the results to BillingCenter. BillingCenter shows the results and enables you to choose one.
- If you choose a contact from a list of exact or potential matches, BillingCenter links that contact and sets it up to be synchronized.
- You might save a new contact if there were no matches listed or if you do not see a good match on the list of contacts. In this case, there is no additional find duplicates check. However, if there is an exact or potential match on the list that you have not chosen, BillingCenter shows a warning that a matching contact was available.

  You must click one of the following:

  - **Update** to continue and create the new contact
  - **Check for Duplicates** to choose from the list.

- In ContactManager, to configure this matching behavior you can edit the `FindDuplicatesPlugin` plugin, which implements the interface `IFindDuplicatesPlugin`. Additionally, there is a set of Duplicate Finder classes that work with the plugin. You can edit these classes to add new subtypes and fields to the find duplicates process.

See also

- “IFindDuplicatesPlugin plugin interface” on page 328
- “ABContactAPI web service” on page 303

Synchronizing with ContactManager

After a contact has been linked with ContactManager, the core applications and ContactManager work together to keep the contact data synchronized. This topic describes how contacts are synchronized, how the applications communicate changes to contact data, and how they ensure that this data is kept current across your Guidewire applications.

ContactManager plugins for broadcasting of contact changes

ContactManager uses three plugins and an API to broadcast contact updates and changes to the Guidewire core applications. The registries for the plugins are `ClaimSystemPlugin.gwp`, `PolicySystemPlugin.gwp`, and `BillingSystemPlugin.gwp`. The base ContactManager application provides plugin implementations that ContactManager can use to communicate with ClaimCenter, PolicyCenter, and BillingCenter. The plugin implementations are:

- `gw.plugin.claim.cc1000.CCClaimSystemPlugin`
- `gw.plugin.policy.pc1000.PCPolicySystemPlugin`
- `gw.plugin.billing.bc1000.BCBillingSystemPlugin`

To enable ContactManager to communicate with the Guidewire core applications you have installed, you configure the application plugin registry corresponding to each installed application by specifying the corresponding plugin implementation. Then ContactManager can then send updates by calling each application’s `ContactAPI` web service, which implements the interface `ABClientAPI`. At that point, the application handles the update in its `ContactAPI` implementation of `ABClientAPI`.

See also

- “Integrating ContactManager with ClaimCenter in QuickStart” on page 21
- “Integrating ContactManager with PolicyCenter in QuickStart” on page 28

Synchronizing PolicyCenter and ContactManager contacts

PolicyCenter synchronizes contact data with ContactManager by using *external contact synchronization*, which is distinct from PolicyCenter internal synchronization. PolicyCenter separately defines behavior for synchronizing contact data internally between accounts and policies.

PolicyCenter performs external contact synchronization with ContactManager for contacts that are shared across accounts, not for contacts at the policy level. PolicyCenter synchronizes contact data between polices and accounts internally.

In the PolicyCenter integration with ContactManager, linked client data is synchronized with ContactManager at all times. For PolicyCenter, ContactManager is the system of record for linked client data.

PolicyCenter uses the class `gw.plugin.contact.ab1000.ABContactSystemPlugin` to create, retrieve, update, delete, and find duplicates of contacts in ContactManager. This class implements `gw.plugin.contact.ContactSystemPlugin` and calls the methods of the ContactManager web service `ABContactAPI`.

To enable ContactManager to send contact updates to PolicyCenter, PolicyCenter implements the ContactManager interface `ABClientAPI` in the class `gw.webservice.pc.pc1000.contact.ContactAPI`. ContactManager requires that all Guidewire core applications implement the `ABClientAPI` interface. Implementing this interface ensures that ContactManager can broadcast contact changes to the core applications by using standard web services, and that the core applications have logic to handle these updates.

Exceptions to PolicyCenter synchronization updates

There are some cases in which PolicyCenter cannot perform a contact change or update:

- ContactManager notifies PolicyCenter that a contact has been deleted, but the contact is being used on at least one account in PolicyCenter.

  This case is not likely to happen, because ContactManager checks to see if a contact can be deleted before it sends the request to delete the contact to PolicyCenter. However, if this case does happen, PolicyCenter unlinks the contact, does not delete the local instance, and creates an activity warning that an external system tried to delete the contact.
- ContactManager notifies PolicyCenter that an address has been deleted, but the address is being used as a policy address in PolicyCenter.

  In this case, PolicyCenter unlinks the local instance of the address and does not delete it.
- PolicyCenter sends changes to a contact's data to ContactManager, but the contact's data was changed previously by another application or directly in ContactManager. The original data for the contact in PolicyCenter, prior to the change, must match the data currently in ContactManager for the change to be accepted. Since the original data that PolicyCenter sent does not match, ContactManager sends an exception to PolicyCenter and does not update the contact's data.

  In this case, PolicyCenter creates an activity for a user to reapply the updates to the contact's data. The activity contains information about the contact and the data that could not be applied in ContactManager.

Configuring PolicyCenter external contact synchronization

The PolicyCenter `Contact` entity has a typekey attribute called `AutoSync` that controls whether contacts are automatically synchronized with the external contact management system. Any PolicyCenter contact that is linked to ContactManager has the same value in its `AddressBookUID` property as the `LinkID` of the corresponding contact in ContactManager. Guidewire recommends that all contacts that are in both PolicyCenter and ContactManager be linked and have an `AutoSync` setting of `Allow`.

> **Note:** In the default integration, PolicyCenter does not send a new contact to ContactManager until there is at least one bound policy on an account that the contact is associated with. When a contact is not linked to ContactManager, it has an `AutoSync` setting of `null`.

For any `Contact` instance, in addition to `null`, the `AutoSync` value can be one of three codes:

`Allow`	Allow the contact to be synchronized automatically. In a rule, this value would be `TC_ALLOW`.
`Disallow`	Do not allow the contact to be synchronized. In a rule, this value would be `TC_DISALLOW`. In its base configuration, PolicyCenter never sets a contact to `Disallow`.

`Suspended`	The contact was synchronized in the past, but synchronizing is not currently allowed. In a rule, this value would be `TC_SUSPENDED`. For example, this value could be set for all contacts during a ContactManager outage.

PolicyCenter uses the Java method `AccountContactImpl.markContactForAutoSync` internally to set the `AutoSync` value on a contact. This method is scriptable. The method sets `AutoSync` to `Allow` only if it is not already set to `Disallow`. `Disallow` is treated as a terminal state.

Synchronizing PolicyCenter contact fields

When PolicyCenter synchronizes contact data with ContactManager, it updates the fields in the PolicyCenter contact with the centralized ContactManager data. The system overwrites all the fields defined in the `ContactMapper` class. This class also defines the fields that PolicyCenter sends to ContactManager. Use this class to control how contact data is synchronized between PolicyCenter and ContactManager.

See also

- "Working with contact mapping files" on page 116

Synchronizing BillingCenter and ContactManager contacts

BillingCenter performs contact synchronization with ContactManager for contacts of accounts and policy periods, but not for contacts of producers. For BillingCenter, ContactManager is the system of record for linked client data.

BillingCenter uses the class `gw.plugin.contact.ab1000.ABContactSystemPlugin` to create, retrieve, update, delete, search for, and find duplicates of contacts in ContactManager. This class implements `ContactSystemPlugin` and calls the methods of the ContactManager web service `ABContactAPI`.

To enable ContactManager to send contact updates to BillingCenter, BillingCenter implements the ContactManager interface `ABClientAPI` in the class `gw.webservice.bc.bc1000.contact.ContactAPI`. ContactManager requires that all Guidewire core applications implement the `ABClientAPI` interface. Implementing this interface ensures that ContactManager can broadcast contact changes to the core applications by using standard web services, and that the core applications have logic to handle these updates.

Exceptions to BillingCenter synchronization updates

There are some cases in which BillingCenter cannot perform a contact change or update:

- ContactManager notifies BillingCenter that a contact has been deleted, but the contact is being used on at least one account in BillingCenter.

  This case is not likely to happen, because ContactManager checks to see if a contact can be deleted before it sends the request to delete the contact to BillingCenter. However, if this case does happen, BillingCenter unlinks the contact and does not delete the local instance.

- BillingCenter sends changes to a contact's data to ContactManager, but the contact's data was changed previously by another application or directly in ContactManager. The original data for the contact in BillingCenter, prior to the change, must match the data currently in ContactManager for the change to be accepted. Since the original data that BillingCenter sent does not match, ContactManager sends an exception to BillingCenter and does not update the contact's data.

  In this case, BillingCenter creates an activity for a user to reapply the updates to the contact's data. The activity contains information about the contact and the data that could not be applied in ContactManager.

Configuring BillingCenter external contact synchronization

The BillingCenter `Contact` entity has a typekey attribute called `AutoSync` that is important in determining if contacts are automatically synchronized with the external contact management system. Any BillingCenter contact that is linked to ContactManager has the same value in its `AddressBookUID` property as the `LinkID` of the corresponding contact in ContactManager. BillingCenter sets `AutoSync` for all new contacts to `Allow`.

Note: In the default integration, BillingCenter does not synchronize contacts of producers. BillingCenter adds a property to `Contact` entities, `ShouldSendToContactSystem`, that determines if the contact is synchronized.

This `Boolean` property is set in a `get` method in `gw.api.address.ContactEnhancement`, described later in this topic.

For any `Contact` instance, the `AutoSync` value can be one of three codes, in addition to `null`:

`Allow`	Allow the contact to be synchronized automatically. In a rule, this value would be `TC_ALLOW`.
`Disallow`	Do not allow the contact to be synchronized. In a rule, this value would be `TC_DISALLOW`. In its base configuration, BillingCenter never sets a contact to `Disallow`.
`Suspended`	The contact was synchronized in the past, but synchronizing is not currently allowed. In a rule, this value would be TC_SUSPENDED. For example, this value could be set for all contacts during a ContactManager outage.

The previous code descriptions provide values you can set for `AutoSync` in a rule. To see a rule that sets `AutoSync`, open Guidewire Studio™ for BillingCenter. Then, in the **Project** window, navigate to **configuration** > **config** > **Rule Sets** > **Preupdate** > **ContactPreupdate** > **All Contacts sync by default**.

BillingCenter does not link or synchronize contacts of producers. To control this aspect of synchronization, BillingCenter adds a property to `Contact` entities, `ShouldSendToContactSystem`, that determines if the contact is synchronized. This `Boolean` property is set in a `get` method in `gw.api.address.ContactEnhancement`.The code for the method is:

```
property get ShouldSendToContactSystem() : boolean {
  return this.AutoSync == AutoSync.TC_ALLOW
    and not this.ID.Temporary
    and (isOnAccount() or isOnPolicyPeriod())
}
```

For a contact to be sent to ContactManager, it must have an `AutoSync` value of `AutoSync.TC_ALLOW` and it must be connected to either an account or a policy period.

Synchronizing BillingCenter contact fields

When BillingCenter synchronizes contact data with ContactManager, it updates the fields in the BillingCenter contact with the centralized ContactManager data. The system overwrites all the fields defined in the `ContactMapper` class. This class also defines the fields that BillingCenter sends to ContactManager. Use this class to control specifics of synchronizing contact data between BillingCenter and ContactManager.

See also

- "Working with contact mapping files" on page 116

Synchronizing ClaimCenter and ContactManager contacts

To enable ContactManager to send contact updates to ClaimCenter, ClaimCenter implements the ContactManager interface `ABClientAPI` in the class `gw.webservice.cc.cc1000.contact.ContactAPI`. ContactManager requires that all Guidewire core applications implement the `ABClientAPI` interface. Implementing this interface ensures that ContactManager can broadcast contact changes to the core applications by using standard web services, and that the core applications have logic to handle these updates.

ClaimCenter synchronizes contacts as follows:

- If ContactManager sends a contact change, ClaimCenter saves only the contact ID and then uses the contact automatic synchronization mechanism to update all local instances of the contact.

  ClaimCenter can have multiple local instances of any contact, one instance for each claim. ClaimCenter uses its automatic synchronization mechanism to ensure that all instances are synchronized. This behavior is different from the other Guidewire core applications. PolicyCenter and BillingCenter have just one local instance of a linked contact and apply all changes sent from ContactManager to their linked contacts.
- Like the other core applications, ClaimCenter automatically sends updates to ContactManager for linked contacts. In the default configuration, for a non-vendor contact, ClaimCenter calls ContactManager and specifies that the change is to be applied to the ContactManager contact. For a vendor contact, when the ClaimCenter user updates

the contact changes, ClaimCenter determines if the user has contact permissions to perform the action. ClaimCenter then calls ContactManager appropriately:

- If the user has permissions, ClaimCenter makes a call to ContactManager specifying that the vendor contact update is to be applied to the ContactManager contact.
- If the user does not have permissions, ClaimCenter makes a call to ContactManager specifying that the vendor contact update is to be pending. A pending update has to be reviewed by a ContactManager contact administrator.

  Note: For information on defining this vendor contact behavior in the class `ContactSystemApprovalUtil`, see “Linking in ClaimCenter” on page 195.

If the update succeeds, ClaimCenter then updates all local instances of the contact by using the automatic synchronization mechanism. A pending update succeeds if the update is approved in ContactManager and ContactManager notifies ClaimCenter.

- When ContactManager applies a contact change or create that originated from ClaimCenter, ContactManager sends the update to any other Guidewire applications that are integrated with ContactManager. It does not send the update back to ClaimCenter.

Configuring automatic synchronization with ClaimCenter

ClaimCenter contacts that are linked to ContactManager contacts can be synchronized automatically. The synchronizing mechanism ensures data consistency between ClaimCenter and ContactManager. This mechanism updates centrally managed contacts appropriately when a contact changes.

ClaimCenter synchronizing support

There are a number of ways to configure synchronization of contacts in ClaimCenter. You can use the `AutoSync` attribute, configure rules, and configure the `ContactAutoSync` work queue.

ClaimCenter AutoSync attribute of Contact

The ClaimCenter `Contact` entity has a typekey attribute called `AutoSync` that controls whether contacts are automatically synchronized. By default, all contacts are synchronized, an `AutoSync` setting of `Allow`.

For any `Contact` instance, the `AutoSync` value can be one of three codes:

`Allow`	Allow the contact to be synchronized automatically. In a rule, this value would be `TC_ALLOW`.
`Disallow`	Do not allow the contact to be synchronized. In a rule, this value would be `TC_DISALLOW`.
`Suspended`	The contact was synchronized in the past, but synchronizing is not currently allowed. In a rule, this value would be TC_SUSPENDED. ClaimCenter typically sets `AutoSync` to this value when a claim is closed. If the claim is reopened, ClaimCenter sets the value of `AutoSync` to `Allow`.

Rules that affect contact synchronization

In the default configurations of ClaimCenter and ContactManager, all contacts are set to allow automatic synchronization. However, for automatic synchronization to run, you must schedule the contact auto sync work queue.

You can determine which contacts you want your users to be able to synchronize. For example, you might not want to synchronize `Person` and `ABPerson` contacts, but you might want to synchronize all `Vendor` and `ABVendor` contacts. Or, you might want to synchronize only contacts that meet particular criteria.

You can use the Preupdate rules in Guidewire Studio™ for ClaimCenter to set automatic synchronization values, determine how addresses update, and set the minimum required tags.

For example, there are predefined **ContactPreupdate** rules in ClaimCenter. To see them, open ClaimCenter Studio. Then in the **Project** window, navigate to **configuration** > **config** > **Rule Sets** > **Preupdate** > **ContactPreupdate**.

COP01000 - Update Check Address

Updates addresses that have changed for synchronized contacts.

COP02500 - Set all Vendors to auto sync

Ensures that all contacts added to a claim are set to synchronize. For all contacts, `AutoSync` is set to `Allow`.

COP03000 - Add default tags

Ensures that the tags for a contact are the minimum required tags for ClaimCenter. The default tags are defined in the class `gw.api.contact.CCContactMinimalTagsImpl`. By default, all contacts get the Claim Party tag, and all vendor types, such as `PersonVendor` and `CompanyVendor` and their subtypes, also get the Vendor tag.

See also

- "Running and scheduling the contact auto sync work queue" on page 203

Setting up the contact auto sync work queue

To process each change to synchronized contacts, ClaimCenter uses a work queue named `CCContactAutoSyncWorkQueue`. You can configure work queues in `work-queue.xml`.

You can open this file in Guidewire Studio™ for ClaimCenter. In the **Project** window, navigate to **configuration** > **config** > **workqueue**, and then double-click `work-queue.xml` to open it in the editor.

Note: After you edit any of the configuration files described in this topic, you must rebuild and redeploy ClaimCenter for the changes to take effect.

The following code shows the default settings for this work queue:

```
<work-queue
    workQueueClass="com.guidewire.cc.domain.contact.CCContactAutoSyncWorkQueue"
    progressinterval="600000">
  <worker instances="1" maxpollinterval="0"/>
</work-queue>
```

You can change the attributes for a work queue and add additional worker instances in `work-queue.xml`.

The comments at the beginning of the `work-queue.xml` file document the attributes for a work queue and provide some guidelines for adding worker instances.

See also

- For general information on configuring work queues, see the *Administration Guide*

Running and scheduling the contact auto sync work queue

You can have your work queues process changes on a schedule or in real time as they happen. The `InstantaneousContactAutoSync` parameter in the ClaimCenter `config.xml` file controls this behavior. This parameter determines how ClaimCenter updates copies of a contact instance that has changed, either by a user action in ClaimCenter or by a change notification from ContactManager.

To open this file, start Guidewire Studio™ for ClaimCenter. Then, in the **Project** window, navigate to **configuration** > **config** and double-click `config.xml`. After changing this parameter, you must rebuild and redeploy ClaimCenter for the changes to take effect.

By default, the `InstantaneousContactAutoSync` parameter is set to `true`:

```
<param name="InstantaneousContactAutoSync" value="true"/>
```

There are also two configuration parameters in `config.xml` that can improve performance of contact synchronization:

`ContactAutoSyncWorkItemChunkSize`

Specifies the maximum number of contacts linked to a single `AddressBookUID` that each `ContactAutoSync` work item is to process. The default value is 400.

`ContactAutoSyncBundleCommitSize`

Specifies the maximum number of contacts that match a single `AddressBookUID` in each bundle commit. The default value is 15.

For example, if 1600 ClaimCenter local contacts are linked to a single ContactManager contact (one `AddressBookUID` for all 1600 contacts), then with these parameters set to their default values:

- There are four work items with 400 contacts each.
- Each work item does 27 commits (400/15).

IMPORTANT: With `InstantaneousContactAutoSync` set to `true`, updating local instances of a contact can affect system performance if you have a large number of local instances. Additionally, if ClaimCenter receives a contact change notification from ContactManager, ClaimCenter updates all local instances of the contact, even instances open for edit. If a ClaimCenter user is editing a local instance of that contact, the user will be unable to save the changes made in the editing session. The user will have to exit the editing session and start over with the newly updated contact data received from ContactManager.

Setting `InstantaneousContactAutoSync` to `false` causes ClaimCenter to perform synchronized updates on the schedule set for the `ContactAutoSync` work queue in the `scheduler-config.xml` file. To open this file in the editor, start Guidewire Studio™ for ClaimCenter, and in the **Projects** window, navigate to **configuration** > **config** > **scheduler** and double-click `scheduler-config.xml`. You must rebuild and redeploy ClaimCenter for changes to take effect.

By default, `ContactAutoSync` is set to run at 5:00 am and 5:00 pm every day. It is on this schedule to ensure that all contacts are synchronized before the Financials Escalation work queue, `FinancialsEsc`, runs every day at 6:05 am and 6:05 pm. Synchronizing contacts before running the Financials Escalation work queue prevents Financials Escalation from having validation errors for contacts that are not synchronized. If you change the `ContactAutoSync` times, also schedule Financials Escalation to run at an appropriate time afterward.

The default `ContactAutoSync` setting looks like this:

```
<!-- Contact Auto Sync batch process should run every day at 5 am and 5 pm.
     It needs to run before the financials escalations to prevent those
     from triggering validation errors from out of sync contacts.-->
   <ProcessSchedule process="ContactAutoSync">
     <CronSchedule hours="5,17"/>
   </ProcessSchedule>-->
```

- If you set `InstantaneousContactAutoSync` to `false` and do not schedule `ContactAutoSync`, the work queue does not run and the system does not process changes.
- If you set `InstantaneousContactAutoSync` to `true` and you have `ContactAutoSync` scheduled, changes process immediately, and then the scheduled `ContactAutoSync` work queue also runs as scheduled.

See also

- For general information on administering and configuring scheduled work queues with the work queue scheduler, see the *Administration Guide*.
- For more information on `ContactAutoSyncBundleCommitSize` and `ContactAutoSyncWorkItemChunkSize`, see the *Configuration Guide*.

Initiating a manual synchronization from ClaimCenter

You can initiate a contact synchronization in the ClaimCenter user interface. Centrally managed contacts have a status message that informs you when they are not synchronized. For example, you have a claim open and on the **Parties Involved** > **Contacts** screen, the **Basics** card for the contact says:

This contact is linked to the Address Book but is out of sync

When you see this status message, you can click **Copy from Address Book** to synchronize the Address Book data for this one contact. Clicking this button copies the contact data from ContactManager to ClaimCenter.

Configuring contacts from ContactManager

Synchronizing ClaimCenter contact fields

When ClaimCenter synchronizes contact data with ContactManager, it updates the fields in the ClaimCenter contact with the centralized ContactManager data.

The system overwrites the fields defined in the ClaimCenter `ContactMapper` class. This class also defines the fields that ClaimCenter sends to ContactManager. Use this class to control which fields are synchronized between ClaimCenter and ContactManager. See "ContactMapper class" on page 313.

In addition, you can edit the `gw.plugin.contact.ab1000.RelationshipSyncConfig` class in Guidewire Studio™ for ClaimCenter to specify the contact relationships to include or exclude in a synchronization.

Including and excluding contact relationships

Contact relationships retrieved from ContactManager such as primary contact, employer, guardian, and so on can be configured with the ClaimCenter `RelationshipSyncConfig` class. Contact relationships configured in this class apply to three processes:

1. Initial contact retrieval from ContactManager
2. Automatic synchronization of contacts
3. Manual contact synchronization

To include or exclude a contact relationship, use the `includeRelationship` and `excludeRelationship` methods of the ClaimCenter `RelationshipSyncConfig` class.

IMPORTANT: Typically, you include only relationships that appear on the **Related Contacts** card of the ClaimCenter **Parties Involved** screen. In particular, avoid zero-or-more relationship types. These types can grow quickly and can result in performance issues as the system pulls down many contacts from ContactManager.

You open the `RelationshipSyncConfig` class for editing in Guidewire Studio™ for ClaimCenter.

The `RelationshipSyncConfig` class has two methods that support including or excluding relationships when synchronizing a Contact subtype:

`includeRelationship`
Specifies relationships to include in synchronizing the specified `Contact` or `Contact` subtype.

`excludeRelationship`
Specifies relationships to exclude in synchronizing the specified `Contact` or `Contact` subtype.

These two methods have the same parameters:

`contactType`
The `Contact` subtype for which the synchronizable relationship will be added or excluded.

`contactBidiRel`
The relationship `ContactBidiRel`, a valid typecode in the `ContactRel` typelist. The typecode must be specified as an enumerated typelist value, such as `TC_EMPLOYER` or `TC_PRIMARYCONTACT`.

`appliesToSubtype`
If `true`, the relationship exclusion or inclusion applies to all subtypes of `contactType` unless overridden by an `includeRelationship` or `excludeRelationship` call. If `false`, the method applies only to the `Contact` subtype specified in `contactType`.

Note: You use a coding pattern called *method chaining* to add the method calls to the `init` method of the `RelationshipSyncConfig` class. With method chaining, the object returned by one method becomes the object from which you call the next method in the chain.

The `init` method can have multiple `includeRelationship` or `excludeRelationship` method calls chained to it. Add the methods before the final `create` method in the chain.

For example, you might include a relationship for one contact type but exclude it for its child subtypes. The following method call, which is included in the `RelationshipSyncConfig` class in the base configuration, includes the `guardian` relationship for a `Person` entity, but not for its subtypes:

```
.includeRelationship(Person, TC_GUARDIAN, false )
```

The default behavior for an entity's relationships is that the system ignores them. Only in certain cases do you need to use `excludeRelationship`, such as overriding a setting of `includeRelationship` in a parent type. For example, the default `Contact` definition includes the relationship `primarycontact` for all `Contact` entities and subentities. However, the `Person` definition overrides the `Contact` definition and excludes the `primarycontact` relationship only for `Person` entities, as shown in the following code snippet:

```
return new RelationshipSyncConfigBuilder<RelationshipSyncConfig>(
    new RelationshipSyncConfig())
  .includeRelationship(Contact, TC_PRIMARYCONTACT, true)
  .excludeRelationship(Person, TC_PRIMARYCONTACT, false)
  <!-- ... -->
  .create()
```

See also

- "Linking a contact" on page 193
- The topic on chaining query builder methods in the *Integration Guide*

chapter 11

ContactManager rules

ContactManager provides rules to handle business processes. These rules use the same Guidewire Studio™ editors and infrastructure described in the *Gosu Rules Guide*.

ContactManager rule sets

This topic describes the sample rules that Guidewire provides as part of the base ContactManager application. You access these rule sets in Guidewire Studio™ for ContactManager by navigating in the **Project** window to **configuration** > **config** > **Rule Sets**.

ContactManager provides the following rule set categories in the base product:

Rule set category	Description
EventMessage	Rules that handle communication with integrated external applications. See "EventMessage rule set category" on page 207. For information on events and event messaging, see the *Integration Guide*.
Preupdate	Rules triggered any time that a contact or other high-level entity changes. See "Preupdate rule set category" on page 210.
Validation	Rules that check for missing information or invalid data on `ABContact` and `Region` entities. See "Validation rule set category" on page 211.

EventMessage rule set category

In the base configuration, there is a single rule set, **EventFired**, in the **EventMessage** rule category. The rules in this rule set:

- Perform event processing.
- Generate messages about events that have occurred.

ContactManager calls the Event Fired rules if an entity involved in a bundle commit triggers an event for which a message destination has registered interest. As part of the event processing, ContactManager:

- Runs the rules in the Event Fired rule set once for every event for which a message destination has registered interest.
- Runs the Event Fired rule set once for each destination that is listening for that particular event. It is possible for ContactManager to run the Event Fired rule sets multiple times for each event, once for each destination interested in that event.

Messaging events

ContactManager automatically generates certain events, called *standard* events, for most top-level objects. In general, events are generated for any addition, modification, removal, or retirement of a top-level entity.

For example, ContactManager automatically generates the following events on the `ABContact` object:

- `ABContactAdded`
- `ABContactChanged`
- `ABContactRemoved`
- `ABContactResync`
- `ABContactPendingChangeRejected`

It is also possible to create a custom event on an entity by using the `addEvent` method. This method takes a single parameter, ***`eventName`***, which is a `String` value that sets the name of the event:

```
entity.addEvent(eventName)
```

See also

- "ContactManager messaging events" on page 308
- For information on messaging events and custom events, see the *Integration Guide*

Link a message event to a message destination

About this task

You can use the Messaging editor to link an event to a message destination.

- A *message destination* is an external system to which it is possible to send messages.
- A *message event* is an abstract notification of a change in ContactManager that is of possible interest to an external system. For example, events can be adding, changing, or removing a Guidewire entity.

Procedure

1. At a command prompt, navigate to the ContactManager installation folder, and then start Guidewire Studio™ for ContactManager by entering the following command:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Messaging**, and then double-click `messaging-config.xml`.

   In the Messaging editor, you can associate one or more events with a particular message destination.

What to do next

See also

- For information on using the Messaging editor, see the *Configuration Guide*
- For information on message destinations, implementing messaging plugins, and messaging and events, see the *Integration Guide*

Message destinations and message plugins

Each message destination encapsulates all the necessary behavior for an external system, but uses three different plugin interfaces to implement the destination. Each of the following plugins handles different parts of what a destination does:

- The message request plugin handles message preprocessing.
- The message transport plugin handles message transport.
- The message reply plugin handles message replies.

You register new messaging plugins first in the Plugins editor in Guidewire Studio™. After you create a new implementation, Studio prompts you for a plugin interface name, and, in some cases, a plugin name. Use that plugin name in the Messaging editor in Studio to register each destination. You must register your plugin in two different editors in Studio, first in the Plugins registry and then in the Messaging editor.

IMPORTANT: After the ContactManager application server starts, ContactManager initializes all message destinations. ContactManager saves a list of events for which each destination requested notifications. Because this initialization happens at system startup, you must restart the ContactManager application if you change the list of events or destinations.

See also

- For information on using the messaging editor, see the *Configuration Guide*

Generating messages

Use the method `createMessage` in rules to create message text. The message that the rule creates can be either a simple text message or a more involved, constructed message.

The following code is an example of a simple text message that uses the Gosu in-line dynamic template functionality to construct the message. Gosu in-line dynamic templates combine static text with values from variables or other calculations that Gosu evaluates at run time. For example, the following `createMessage` method call creates a message that lists the event name and the entity that triggered this rule set.

```
messageContext.createMessage("${messageContext.EventName} - ${messageContext.Root}")
```

See also

- For information on generating new messages in event-fired rules, see the *Integration Guide*

Modifying entity data in event fired rules and messaging plugins

Be careful about modifying entity data in Event Fired rules and messaging plugin implementations. Use these rules to perform only the minimal data changes necessary for integration code. Entity changes in these code locations do not cause the application to run or rerun validation or preupdate rules. Therefore, do not change fields that might require those rules to run again. Only change fields that are not modifiable from the user interface. For example, you might set custom data model extension flags only used by messaging code.

Important message configuration caveats

ContactManager does not support the following:

- Adding or deleting business entities from Event Fired rules or messaging plugins, even indirectly through other APIs.
- Calling any message acknowledgment or any skipping methods such as the `Message` methods `reportAck`, `reportError`, or `skip` in rules. Use those methods only in messaging plugins. This prohibition also applies to Event Fired rule set execution.
- Creating messages outside the Event Message rule set.

See also

- For an overview of messaging, see the *Integration Guide*
- For information on using the Messaging editor, see the *Configuration Guide*

Preupdate rule set category

Use the preupdate rule sets to perform domain logic or validation that must be committed before the entity in question is committed. Only a change to an entity that implements the `Validatable` delegate can trigger a preupdate rule.

Typically, these rules execute before the validation rule set. Preupdate rules can perform a wide variety of actions as needed. The primary use of the preupdate rules in the base configuration is to keep a history of changes to contacts and addresses.

Additionally, there is a Pending Contact Change Preupdate rule set with rules that handle rejection of either a pending contact creation or a pending contact update.

See also

- For information on how the validation and preupdate rules work together in performing validation, see “Overview of ContactManager preupdate and validation” on page 211.

Entities that trigger preupdate rules

The following entities trigger preupdate rules in the ContactManager base configuration:

- `ABContactContact`
- `ABContact`
- `Address`
- `PendingContactChange`

For an entity to trigger the preupdate rules, it must implement the `Validatable` delegate. All entities that implement the `Validatable` delegate trigger the validation rules as well.

ContactManager runs the preupdate and validation rules if:

- An instance of a validatable entity is created, modified, removed, or retired.
- A subentity of a validatable entity is created, modified, removed, or retired, and the validatable entity is connected to the subentity through a `triggersValidation="true"` link.

IMPORTANT: In running the preupdate rule set, ContactManager first computes the set of objects on which to run the preupdate rules. It then runs the rules on this set of objects. If a preupdate rule modifies an entity, the preupdate rules for that entity do not fire unless the schedule for preupdate rules already includes this entity. In other words, changing an object in a preupdate rule does not cause the preupdate rules to run on that object as well. Additionally, creating a new entity in a preupdate rule does not cause the preupdate rules to run on that entity.

Creating preupdate rules for custom entities

You can create preupdate rules for entities that you create—entities that are not part of the base Guidewire ContactManager configuration.

- The entity must implement the `Validatable` delegate with an `implementsEntity` element that has its **Name** set to `Validatable`.
- The preupdate and validation rule sets that you want the custom entity to trigger must conform to the following naming convention:
    - Put your preupdate rules in the **Preupdate** rule set category and name the rule set `*entity-name*Preupdate`.
    - Put your validation rules in the **Validation** rule set category and name the rule set `*entity-name*ValidationRules`.

  You must create a rule set category with the same name as the extension entity and add the word `Preupdate` to it as a suffix. You then put your rules in this rule set.

  For example, if you create an extension entity named `NewEntityExt`, you need to create a rule set to hold your preupdate rules and name it `NewEntityExtPreupdate`.

See also

- For information on creating new rule sets, see the *Gosu Rules Guide*.

ABContactContact preupdate rule set

Use the `ABContactContact` preupdate rules to modify `ABContactContact` and related entities. This rule set in the base configuration has rules that track the history on changes to a contact's related contacts.

ABContact preupdate rule set

Use the `ABContact` preupdate rules to modify `ABContact` and related entities. This rule set in the base configuration has rules that:

- Set the `BatchGeocode` field for a vendor contact.
- Track history for creation of a contact, removal of a related contact, and changes to a contact's name, phone numbers, addresses, and so on.
- For an `ABContact` entity that has changed, remove entries for that entity from the cache for the **Pending Changes** screen.

See also

- "Pending changes screen cache" on page 279

Address preupdate rule set

Use the `Address` preupdate rules to modify `Address` and related entities. This rule set in the base configuration has rules that track the history on changes to a contact's primary and secondary addresses.

PendingContactChangePreupdate rule set

The pending contact change preupdate rules handle rejection of either a pending contact creation or a pending contact update. Pending contacts are reviewed by a contact administrator and can be either accepted or rejected. Rejected changes trigger an event that sends a message to the core application, which notifies the user who made the change that the change was rejected.

Additionally, for a rejected pending change to an `ABContact` entity, there is a rule that removes entries for that entity from the cache for the **Pending Changes** screen.

See also

- "Review pending changes to contacts" on page 278
- "Pending changes screen cache" on page 279

Validation rule set category

ContactManager uses validation rules to ensure that:

- A region zone is available in the zone lookup for the organization.
- The date of birth of each `ABContact` instance is in the past.
- The country code portion of all non-null phone numbers for each `ABContact` instance is correctly formatted.

Overview of ContactManager preupdate and validation

ContactManager runs preupdate and validation rules every time it does a *database bundle commit*—commits data to the database. The validation rules execute after ContactManager runs all preupdate callbacks and preupdate rules. ContactManager runs the validation rules as the last step before writing data to the database.

Before applying rules during the bundle commit operation, ContactManager builds a validation object graph according to configuration settings. The graph contains possible targets for rule execution and is used by both the preupdate and the validation rules.

Preupdate rules, unlike validation rules, can modify additional objects during execution. Therefore, there can be additional objects that need to be constructed after the preupdate rules run and before the validation rules run.

Following are some examples indicating that the set of entities to be validated can be a super set of the entities for which preupdate rules are run:

Preupdate rules modify an entity that is not in the commit bundle
ContactManager can run additional validation rules if the entity being modified also triggers validation.

Preupdate rules modify only entities that are in the commit bundle
ContactManager runs the validation rules for the same set of entities on which the preupdate rules ran.

Preupdate rules modify an entity that triggers validation for one of the top-level entities
There is no difference from the top-level entity validation rules because they were already going to run.

Note: Preupdate rules are not recursive. They do not run a second time on objects modified during execution of the preupdate rules. ContactManager just adds any objects modified by the preupdate rules to the list of objects needing validation, as described by the validation graph.

Overview of ContactManager validation

For an entity to have preupdate or validation rules associated with it, the entity must be validatable. To be validatable, the entity must implement the `Validatable` delegate. In the base configuration, ContactManager comes preconfigured with the following high-level entities that can trigger validation:

- `ABContact`
- `ABContactContact`
- `Activity`
- `Address`
- `Group`
- `PendingContactChange`
- `Region`
- `User`

ContactManager can validate these entities in no particular order.

See also

- “Top-level ContactManager entities that trigger validation” on page 213
- For information on `<implementsEntity>`, see the *Configuration Guide*

Validation graph in ContactManager

During database commit, ContactManager performs validation on the following entities:

- Any validatable entity that is itself either updated or inserted.
- Any validatable entity that refers to an entity that has been updated, inserted, or removed.

ContactManager gathers all the entities that reference a changed entity into a virtual graph. This graph maps all paths from each type of entity to the top-level validatable entity, such as `ABContact`. These paths are queried in the database or in memory to determine which validatable entities, if any, refer to the entity that was inserted, updated, or removed.

ContactManager determines the validation graph by traversing the set of foreign keys and arrays that trigger validation. For example, suppose that the data model marks the `Address` array on `ABContact` as triggering validation. Therefore, any changes made to an address causes the Rules engine to validate the contact as well.

ContactManager follows foreign keys and arrays that trigger validation through any links and arrays on the referenced entities down the object tree. For example, you might end up with a path like `ABContact` → `ABContactAddress` → `Address`. To actually trigger validation, each link in the chain—`Address` and `ABContactAddress`—must be marked as triggering validation, and `ABContact` must implement the Validatable delegate.

ContactManager stores this path in reverse order in the validation graph. Thus, if an address changes, ContactManager follows the path to find the contact that references a changed address. ContactManager transverses the tree from address to contact address, and finally to the contact—`Address` → `ABContactAddress` → `ABContact`.

Top-level ContactManager entities that trigger validation

To be validatable, an entity, subtype, delegate, or base object must implement the `Validatable` delegate. If an entity has a foreign key or array for which `triggersValidation` is `true`, the referenced entities must also implement the `Validatable` delegate.

The following table lists the entities that trigger validation in the base ContactManager configuration.

Validatable entity	**Foreign key**	**Array**	**Comments**
`ABContact`	`PrimaryAddress`	• `ContactAddresses` • `SourceRelatedContacts` • `TargetRelatedContacts`	In the base configuration, ContactManager runs the preupdate and validation rules on the `ABContact` object during database commit if any objects with `triggersValidation="true"` have been modified.
`User`	• `Contact` • `Credential` • `UserSettings`	`Attributes`	In the base configuration, ContactManager does not have any preupdate or validation rules for the `User` entity.

chapter 12

Configuring personal data destruction in ContactManager

ContactManager supports destruction of some kinds of data. Destruction can mean either purging the data completely from the database or it can mean obfuscating data, making the original contents permanently unreadable.

Guidewire recognizes the need for insurers to be able to destroy personal information both on an on-demand basis or on a time-based basis. Destruction can be mandated by regulation or business practices, within the requirements of regulation, codes of conduct, or other business practices.

Overview of data destruction

Important note

Note: The data destruction features described in these topics provide a set of features that help enable insurers to comply with some of their data destruction requirements. These requirements may be driven by insurers' policies and practices, as well as by their interpretation of various regulatory requirements. Such regulatory requirements may come from, for example, the European Union General Data Protection Regulation (GDPR) or the New York State Cybersecurity Requirements for Financial Services Companies law.

Data destruction terminology

Data destruction is the process of requesting that data be destroyed, making the data impossible to retrieve. Data destruction is typically initiated with a request that specifies a contact or user whose data is to be destroyed. In the base configuration, ContactManager provides a web service that is intended to be called by an external application. You use the external application to manage the destruction of the data across Guidewire applications.

Data destruction can be implemented as either purging or obfuscation of data, depending on the data to be destroyed.

Purging is a form of data destruction that completely removes contact data from ContactManager. There can be multiple objects associated with the contact that are also removed as they are detected by traversing the entity domain graph.

Obfuscation is a form of data destruction that permanently overwrites fields, such as user contact fields, with data that replaces the original data. Some actual removal of data can also be involved, such as deletion of an address referenced only by one user.

Obfuscation might be required if destroying the data affects contacts that cannot be destroyed. For example, purging user data for a former employee could affect hundreds or even thousands of contacts. Therefore it makes more sense to obfuscate the data for the user and leave the other data alone.

Encapsulation of business logic for retention and destruction

Regulations, codes of conduct, and other generally accepted business practices vary from jurisdiction to jurisdiction. Additionally, business policies and interpretation of conflicting legal requirements vary from insurer to insurer. Therefore no single approach meets the needs of all insurers. To accommodate varying needs, ContactManager provides a configurable solution that captures business logic for retention and destruction in one place.

There is a configurable plugin that has access to the business objects to be removed through the `ABContact` root object. The examination of objects to be removed starts with the root object and traverses a graph of objects, enabling detailed examination of the business objects. You can mark requests requiring user review—manual intervention—for those data destruction requests that require special handling, prior to the destruction actually occurring.

See also

- "ContactManager entity domain graph" on page 225

Notification of data protection officer on errors or conflicts

Requirements for destruction and for retention can conflict with each other. While the plugin class might be able to resolve conflicts in a generic way, situations can arise when the two sets of requirements are not reconcilable. Additionally, the data destruction process can encounter errors. In these situations, notification is done through a configurable plugin.

The default behavior of this plugin is that a message is logged that describes the situation.

After the situation has been resolved, the destruction request can be queued again for reprocessing.

See also

- "ContactManager Data Protection Officer" on page 229

Wide-swath data destruction

In many situations, there is a need to destroy the personal data related to a specific business object, a contact. Specifically, the `ABContact` entity and its subentities can require this kind of destruction.

This object can affect many individual data objects. A single call allows the entirety of related data to be removed. In the case where these business objects are nested, a best-effort destruction is performed.

ContactManager components provide the ability to purge rows from the database for business objects such as `ABContact` and related data. This approach is suitable for high-volume data destruction.

See also

- "Data destruction purge configuration" on page 229

Individual-entity data destruction

While wide-swath data destruction meets the needs of the insurer in most cases, there are special cases in which specific personal data cannot be deleted. For example, there might be database integrity concerns, or the data to be deleted, such as data for previous employee, might be related to a large number of contacts.

In such cases, where individual instances of data cannot be deleted, ContactManager provides the ability to obfuscate data. Obfuscation can include wiping a field completely, replacing it with a neutral value, or replacing it with a unique, irreversible value. Additionally, some actual removal of data can also be involved, such as deletion of an address referenced only by one user.

The entities and fields to which obfuscation can applied, as well as the method for determining the replacement value, are configurable.

See also

- "Data obfuscation in ContactManager" on page 233

Integration with Guidewire core applications

ContactManager coordinates with Guidewire core applications on any request to destroy an `ABContact` object or a subobject of `ABContact`. This coordination is the same as with any contact removal. ContactManager cannot purge an `ABContact` until it receives permission from any core applications that are installed.

However, `User` objects are local to ContactManager, so obfuscation of a user can proceed without consulting core applications.

See also

- “Data destruction purge configuration” on page 229
- “Data obfuscation in ContactManager” on page 233

Integration with other systems

ContactManager needs to be able to respond to data destruction requests from external systems, as well as have the ability to notify data consumers of data destruction.

ContactManager provides a web service that:

- Takes a reference to an individual contact.
- Takes application-specific action to destroy the data related to that contact.
- Reports back to the caller on the level of success of the request. Callers can query the status of a given request.

See also

- “Data destruction web service” on page 217

Notification of downstream systems

ContactManager provides a messaging system to assist you in ensuring that the destruction of personal data flows into systems connected with components. Additionally, you might need to notify outside organizations that process data on your behalf. The messaging system supports broadcasting personal data destruction response messages.

These messages are delivered by using the existing ContactManager guaranteed-delivery messaging system.

See also

- “PersonalDataPurge event” on page 231

Data destruction configuration parameters

In the base configuration of ContactManager, data destruction is not enabled. You must set the following configuration parameters in `config.xml` to enable data destruction:

- `PersonalDataDestructionEnabled` – Set this parameter to `true` to enable destruction of personal data. In the base configuration, this parameter is `false`.
- `ContactDestructionRequestAgeForPurgingResults` – Used by the `RemoveOldContactDestructionRequest` work queue to determine the age of `PersonalDataDestructionRequest` objects that have a status of Finished. In the base configuration, this parameter is set to 10 days.

Data destruction web service

ContactManager provides a web service that enables external software programs to initiate and track requests to destroy data. In the base configuration, this web service, `PersonalDataDestructionAPI`, enables an external application to request:

- Destruction of a contact’s data by `AddressBookUID` or by `PublicID`.
- Destruction of a user by user name.

In the base configuration, the `PersonalDataDestroyer` obtained by the class that implements the `PersonalDataDestructionPlugin` uses an `ABContact PublicID`. You can configure the `PublicID` to correspond to an entity other than `ABContact`.

The requests for removal return a unique `requesterID` that can be used to check the status of the request. Additionally, this `requesterID` is available in the plugin notification call when the request has been completed.

Note: The `PersonalDataDestructionAPI` checks for both retired and active contacts when a contact data destruction request is received. When implementing data obfuscation for contacts, you must evaluate the need to look for related objects that are retired in the database. Retired objects can be obfuscated in the same fashion as active objects, but must be retrieved from the database with a query that is specified to look for retired objects. For example: `query.withFindRetired(true)`.

IMPORTANT: The external software program that calls the web service must destroy data for a contact in the core applications before requesting that ContactManager destroy the contact. If you have more than one core application installed, the external application must request that the contact be destroyed in all of them before sending the request to ContactManager. When ContactManager processes a request to destroy a contact, it first verifies with all installed core applications that it can destroy the contact. If any core application indicates that the contact cannot be destroyed, ContactManager does not proceed with the destruction request and notifies the web service that the contact cannot be destroyed.

See also

- "Defining the Destroyer" on page 230
- "Specifying which objects in the graph can be destroyed" on page 231

PersonalDataDestructionAPI web service methods

`PersonalDataDestructionAPI` provides the following methods:

- `requestContactRemovalWithABUID(addressBookUID : String, requesterID: String) : String` – A request to destroy a contact by `AddressBookUID`. In ContactManager, the `AddressBookUID` is checked against the `LinkID` of an `ABContact` object. This method is implemented as an asynchronous process that uses work queues.
- `requestContactRemovalWithPublicID(contactPublicID : String, requesterID: String) : String` – A request to destroy a contact by `PublicID`. In ContactManager, the contact is an `ABContact`. This method is implemented as an asynchronous process that uses work queues.
- `doesABUIDExist(addressBookUID: String): boolean` – Uses `translateABUIDToPublicIDs` as defined in the class that implements the `PersonalContactDestroyer` interface. In ContactManager, this translation method compares the `AddressBookUID` to the `LinkID` of an `ABContact`.
- `doesContactWithPublicIDExist(publicID: String): boolean` – In ContactManager, the contact is an `ABContact` object.
- `currentDestructionRequestStatusByRequestID(uniqueRequestID : String): DestructionRequestStatus`
- `destroyUser(username : String) : boolean` – Given a ContactManager user name, verifies the existence of the credential and the user. This method is a synchronous destruction request that does not involve work queues and does not require coordination with core applications. It works with the `Contact` object and not with `ABContact` because `User` is a subtype of `Contact`.
    - If the credential exists and the user does not, then the method obfuscates the credential, logs that the user does not exist, and returns `credential.IsObfuscated`.
    - If both credential and user exist, the method attempts to obfuscate the user, which obfuscates both the credential and the `UserContact` object. If the obfuscation succeeds, the method returns `true`. If not, it logs that there was a failure and returns `false`.

See also

- "Defining the Destroyer" on page 230

Lifecycle of a personal data destruction request

The lifecycle of a contact removal request depends on the method that the external system calls to start the request. The lifecycle, also called an *asynchronous* personal data destruction request, is started by a call either to `requestContactRemovalWithABUID` or `requestContactRemovalWithPublicID`. For these two web service method calls, the external system has either the `AddressBookUID` or the `PublicID` of the contact whose data to be destroyed. The destroy action performed is defined in the ContactManager plugin class that implements the `PersonalDataDestruction` plugin interface.

> **Note:** In the base configuration, the `destroyUser` method is synchronous and initiates obfuscation. It does not use work queues, and therefore does not participate in the personal data destruction request lifecycle. Additionally, because users of the application are subentities of `Contact`, this method works with `Contact` objects and not `ABContact` objects.

If the web service determines that the request is an existing one, it adds the specified `requesterID` value to the existing destruction request and does not start a new request.

If the web service determines that the request is a new one, the web service:

1. Does the following depending on whether the request is for an `AddressBookUID` or `PublicID`:
   - If the web service call was to `requestContactRemovalWithABUID`, the web service:
     - Creates a `PersonalDataDestructionRequest` object for the `LinkID` of the `ABContact`.
     - Adds a `PersonalDataContactDestructionRequest` object for the related `PublicID` value, obtained from a call to the `PersonalDataDestroyer` implementation.
   - If the web service call was to `requestContactRemovalWithPublicID`, the web service creates a `PersonalDataContactDestructionRequest` object for the `PublicID` of the `ABContact`.
2. Adds a `PersonalDataDestructionRequester` object using `requesterID`.
3. The `DestroyContactForPersonalData` work queue checks for requests in the `ReadyToAttemptDestruction` category, status `New` or `ReRun`, and calls the Destroyer.

   The class `PersonalDataContactDestructionWorkQueue`, which implements this work queue, calls the following method:

   ```
   PersonalDataDestructionController.destroyContact(contactPurgeRequest)
   ```

   - If the request status is in the `DestructionStatusFinished` category, the queue marks the date of destruction for the contact destruction request.
   - If the request status is `ManualInterventionRequired`, you must implement code that notifies the data protection officer. That user must determine what to do and then set the status to `ReRun` so the `DestroyContactForPersonalData` work queue can run it again.
4. The `NotifyExternalSystemForPersonalData` work queue looks at all `PersonalDataContactDestructionRequest` objects that are associated with a `PersonalDataDestructionRequest`. If they all have a status that is in the `DestructionStatusFinished` category, the work queue does the notification.
5. The `NotifyExternalSystemForPersonalData` work queue notifies the external system by using `PersonalDataDestructionRequester` objects. As part of this notification, the work queue calls the `PersonaDataDestruction` plugin method `notifyExternalSystemsRequestProcessed`.
6. The `RemoveOldContactDestructionRequest` work queue removes all requests that satisfy both of the following criteria:
   - The date obtained by adding the value of the configuration parameter `ContactDestructionRequestAgeForPurgingResults` to the value of `PersonalDataContactDestructionRequest.purgedDate` is less than or equal to today's date.

- The PersonalDataContactDestructionRequest object has a typecode that is in the DestructionStatusFinished category.

Personal data destruction request entities

Three kinds of entities are created when a request is made to purge a contact:

PersonalDataDestructionRequest
Holds the AddressBookUID (LinkID) and information regarding the parts and result of this destruction request.

PersonalDataContactDestructionRequest
Holds the PublicID of the ABContact and its current destruction status.

PersonalDataDestructionRequester
Holds the external system RequesterID that requested the purge and a unique ID associated with the request assigned by ContactManager.

Example of a request made with AddressBookUID

If the call is made to the web service method requestContactRemovalWithABUID, only one contact can have the LinkID. The destruction request causes the following instances to be created:

- One PersonalDataDestructionRequest
- One PersonalDataContactDestructionRequest object for the PublicID linked to the AddressBookUID request
- One PersonalDataDestructionRequester

If a LinkID corresponding to the AddressBookUID is not found, an exception is thrown.

If an existing destruction request for AddressBookUID has AllRequestsFulfilled equal to true, then the external system is notified that destruction has already finished.

Example of a request made with PublicID

If two calls are made to the web service method requestContactRemovalWithPublicID for the same PublicID, the destruction request creates:

- One PersonalDataDestructionRequest
- One PersonalDataContactDestructionRequest
- Two PersonalDataDestructionRequester objects.

If the PublicID is not found, an exception is thrown.

If an existing destruction request with PublicD has AllRequestsFulfilled on the PersonalDataDestructionRequest equal to true, then the external system is notified that destruction has already finished.

Typelists for status of personal destruction workflow

The personal destruction workflow uses typecodes to indicate various statuses. These typecodes are defined in three typelists, ContactDestructionStatus, DestructionRequestStatus, and ContactDestructionStatusCategory.

ContactDestructionStatus

When a new contact destruction request is started, the initial status of the destruction object is New. These status values are defined in the ContactDestructionStatus typelist. After a destruction attempt is made on the contact, the destroyer is expected to return a status corresponding to how much it was able to destroy:

- New – The initial status of the destruction object when a new contact destruction request is started.

- `NotDestroyed` – Nothing could be destroyed.
- `Partial` – Some data was destroyed.
- `Completed` – Everything requested was destroyed.
- `ManualIntervention` – There was an error. This status enables inspection by the Data Protection Officer and must be set before setting `ReRun`.

  In the base configuration, ContactManager provides code that notifies the Data Protection Officer in the plugin class that implements `PersonalDataDestruction`. Additionally, after the Data Protection Officer takes action, the method `PersonalDataDestructionController.requeueContactRemovalRequestWithPublicID` must be called. You must configure a way to make that method call, such as a button in the ContactManager user interface.
- `ReRun` – Enables another attempt at destruction.

DestructionRequestStatus

You can retrieve the status of the entire destruction request through the `Status` property on the request itself. The status values are defined in the `DestructionRequestStatus` typelist. The general status of the entire destruction request can be:

- `DoesNotExist` – The object to be destroyed does not exist.
- `Unprocessed` – Everything is still in the New status.
- `InProgress` – All other combinations of contact destruction statuses.
- `Finished` – Everything is in a final state.

ContactDestructionStatusCategory

Every `ContactDestructionStatus` typecode except `ManualInterventionRequired` has one or more categories.

- `DestructionStatusNotProcessed` – Indicates that the request has not gone through the destruction process.
- `ReadyToAttemptDestruction` – Labels the contact purge request as being ready to be sent to the destroyer.
- `DestructionStatusFinished` – Indicates that the request has finished the destruction process.
- `ReadyToBeNotified` – Labels the request as ready to notify to the external system.

`New` and `ReRun` are under the category `ReadyToAttemptDestruction`.

`New` also is included in the category `DestructionStatusNotProcessed`.

`Partial`, `NotDestroyed`, and `Completed` are under both the category `DestructionStatusFinished` and the category `ReadyToBeNotified`.

Work queues used in personal data destruction

The following work queues are used in personal data destruction: `DestroyContactForPersonalData`, `NotifyExternalSystemForPersonalData`, and `RemoveOldContactDestructionRequest`.

DestroyContactForPersonalData work queue

This work queue finds all `PersonalDataContactDestructionRequest` objects that have a status set to `New` or `ReRun` (category `ReadyToAttemptDestruction`). How far the destruction process went for the found contacts is determined by the `ContactDestructionStatus` returned from the Destroyer, the class that implements the `PersonalDataDestroyer` interface.

The contact destruction status is set to the returned status. If the status is `Completed`, `Partial`, or `NotDestroyed` (category `DestructionStatusFinished`), the date of completion is also populated.

An exception is thrown if return status is `New` or if you try to change the status from a typecode in the `DestructionStatusFinished` category.

Note: The class that implements this work queue is `PersonalDataContactDestructionWorkQueue`.

NotifyExternalSystemForPersonalData work queue

This work queue finds all `PersonalDataDestructionRequest` objects that have a status typecode in the `DestructionStatusFinished` category and that have `RequestersNotified` set to `false`. The work queue processes found requests by sending a notification to all associated requesters, and then the work queue marks `RequestersNotified` `true`. If the notification fails, an exception is thrown and `RequestersNotified` remains `false`.

Note: The class that implements this work queue is `PersonalDataDestructionNotifyExternalSystemsWorkQueue`. In your implementation, you must verify that the notification was successful before marking `RequestersNotified` `true`.

A method on the `PersonalDataDestruction` plugin, `notifyExternalSystemsRequestProcessed`, is called by the class `PersonalDataDestructionNotifyExternalSystemsWorkQueue` to notify external systems when a personal data destruction request is completed. The original `RequestID` is passed to the method, which does nothing by default. You are expected to implement this method to notify systems of interest. The `RequestID` is received when the destruction request is originally created through `PersonalDataDestructionAPI`.

See also

- "Personal data destruction plugin implementation classes" on page 232

RemoveOldContactDestructionRequest work queue

This work queue finds all `PersonalDataDestructionRequest`, `PersonalDataContactDestructionRequest`, and `PersonalDataDestructionRequester` objects that have the following values:

- `RequestersNotified` set to `true`
- `PersonalDataContactDestructionRequest.DestructionDate` plus the value of the `ContactDestructionRequestAgeForPurgingResults` configuration parameter is less than or equal to today's date
- Each found request that has `AllRequestsFulfilled` equal to `true` is removed.

Note: The class that implements this work queue is `RemoveOldContactDestructionRequestWorkQueue`.

PersonalDataDestructionController class

This class handles the complete lifecycle of the asynchronous destruction process after a destruction request has been made through the `PersonalDataDestructionAPI` web service. This class attempts to destroy the contact and notify the requesters that the contact has been destroyed.

The class provides the following methods relating to the lifecyle:

`notifyRequesterDestructionRequestHasFinished(destructionRequester: PersonalDataDestructionRequester)`

Takes a requester and notifies the external system that the request with related `AddressBookUID` and `PublicID` values was processed and finished.

`destroyContact(contactDestructionRequest: PersonalDataContactDestructionRequest)`

Called by `PersonalDataContactDestructionWorkQueue`, the class that implements the `DestroyContactForPersonalData` work queue, when attempting destruction. Uses the class that implements the `PersonalDataDestroyer` interface, called the Destroyer, to destroy the contact, and returns a `ContactDestructionStatus`. Verifies status and sets appropriate `PersonalDataContactDestructionRequest` and `PersonalDataDestructionRequest` attributes.

`requeueContactRemovalRequestWithPublicID(publicID : String, bundle : Bundle)`

Sets purge request status to `ReRun` after manual intervention, allowing the purge request to be reattempted for destruction.

See also

- "Data destruction web service" on page 217

- “Defining the Destroyer” on page 230

Data model configuration for data destruction

There are data model configurations that apply to the objects being destroyed. Some are general configurations, and some are specific to purging or obfuscation.

DestructionRootPinnable delegate

An entity that implements this delegate gets a `DoNotDestroy` flag that can be checked as part of the destruction process. An entity that is intended to be the root of an entity graph must implement the `DestructionRootPinnable` delegate if it is to be used in personal data destruction.

Root of entity graph for personal data destruction

The destruction process uses an entity domain graph to determine what to destroy. An object that implements the `DestructionRootPinnable` delegate and the `RootInfo` delegate is the root of an entity domain graph.

ContactManager has one root entity, `ABContact`.

Do Not Destroy flag

A Do Not Destroy flag is provided in the `DestructionRootPinnable` delegate. If an entity implements this delegate, instances of the entity have a `DoNotDestroy Boolean` field. The default value of this field is `false`.

Note: If this field is on a `Contact` entity or a subentity of `Contact`, it is maintained only in ContactManager. Even if the contact is linked, the field is not sent to ContactManager, nor is it updated from ContactManager.

In the base configuration of ContactManager, the `ABContact` entity implements the `DestructionRootPinnable` delegate and therefore has a `DoNotDestroy Boolean` field. The default value of the field is `false`, which permits destruction of the entity. If this field is set to `true`, the entity cannot be purged.

You can set this field for `ABContact` and `Contact` objects. Use the `markDoNotDestroy` method to set the field. For example:

```
ABContact.markDoNotDestroy(true)
```

Do Not Process flag

A Do Not Process flag is provided by the `Operable` delegate. If an entity implements this delegate, the entity than has a `DoNotProcess Boolean` field that can you use as needed in your configuration of ContactManager. This flag is not used in the base configuration.

Display keys for data destruction messages

There are a number of display keys defined for use by the personal data destruction code. You can see display keys for all languages supported by Guidewire in Guidewire Studio, such as the U.S. English display key properties. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor. In this file, press `Ctrl+F`, enter the search string `PersonalData`, and then click the down or up arrow button to go to each search result in the file.

For example:

```
PersonalData.Error.ObfuscateNotImplemented = Obfuscation for '{0}' is not supported
```

Obfuscatable delegate

If you intend to use obfuscation with an entity, it must implement the `Obfuscatable` delegate. This delegate is necessary to support marking fields as personally identifiable information with the `PersonalData` tag.

> **Note:** A `Delegate` is a reusable type that defines database columns, an interface, and a default implementation of that interface. A delegate permits an entity to implement an interface while delegating the implementation of that interface to another class, that of the delegate. Each delegate type provides additional columns on the affected tables.

The `Obfuscatable` delegate has one column, `ObfuscatedInternal`, which cannot be set in Gosu code. This limitation exists because after the `ObfuscatedInternal` column has been set to `true`, it must not be set to `false`.

> **Note:** If the parent of an entity implements the `Obfuscatable` delegate, all child entities inherit that implementation.

The `Obfuscatable` delegate extends the interfaces `Obfuscator` and `ObfuscatablePublicMethods`. These interfaces provide the following methods:

`markAsObfuscated`
: Marks the entity instance as having been obfuscated by setting `ObfuscateInternal` to `true` on the delegate.

`isObfuscated`
: Returns `true` if `obfuscate` has been called and completed successfully. Otherwise returns `false`.

`obfuscate`
: Obfuscates the columns that are marked for obfuscation on this object.

`obfuscateSimple`
: Works the same as `obfuscate`.

Obfuscator interface

If you intend to use obfuscation with an entity, the entity must implement the `Obfuscator` interface. Each entity that implements the `Obfuscator` interface must also designate an implementation class, such as `UserContactObfuscator`.

> **Note:** If the parent of an entity implements the `Obfuscator` interface, all child entities inherit that implementation, including the implementation class. You can override the parent implementation by declaring `implementsInterface` in the child entity.

For example, in the base configuration, `UserContact.etx` has the following definition:

```
<extension xmlns="http://guidewire.com/datamodel"
  entityName="UserContact">
  <implementsInterface
    iface="gw.api.obfuscation.Obfuscator"
    impl="gw.personaldata.obfuscation.UserContactDefaultObfuscator"/>
  <column-override
    name="EmployeeNumber">
    <tag
      name="PersonalData"
      value="ObfuscateDefault"/>
  </column-override>
</extension>
```

In the base configuration, an entity can implement the `Obfuscatable` delegate but have an `Obfuscator` interface implementation of `UnsupportedObfuscator`. In this case, even if the entity is marked as obfuscatable, any attempt to obfuscate it results in an `UnsupportedOperation` exception.

To be able to obfuscate an entity, you must either use an existing obfuscator or write a suitable obfuscator, such as one that extends `PersonalDataObfuscator`.

See also

- “Personal data obfuscation classes” on page 234

- "Implementing the Obfuscator interface in an entity" on page 233

Marking entity fields for obfuscation

An entity that implements the `Obfuscatable` delegate must also have fields tagged for obfuscation. For each field that contains personal data, add the tag `PersonalData`. The value of the tag determines the kind of obfuscation that will be applied to the field's contents. In the base configuration, two values are defined in the typelist `PersonalDataTagValue.tti`:

`ObfuscateDefault`

The field's contents are replaced with the default value or `null` if no default value is defined.

`ObfuscateUnique`

The field's contents are replaced as you define.

In the base configuration, all the entities that implement the `Obfuscator` interface have fields tagged `PersonalData`. Additionally, many of the fields tagged as `PersonalData` are in `.etx` files to enable you to modify them. For example, the `Name` field of `Contact` is defined in `Contact.etx` as follows:

```
<column-override
  name="Name">
    <tag
      name="PersonalData"
      value="ObfuscateUnique"/>
</column-override>
```

In addition, the following delegates have fields tagged `PersonalData`. An entity that implements one of these delegates does not have to implement `Obfuscator` and `Obfuscatable` unless you want the entity to support obfuscation.

- `AddressBookConvertible.etx`
- `GlobalAddress.etx`
- `GlobalAddress.Global.etx`
- `GlobalContactName.etx`
- `GlobalContactName.Global.etx`
- `GlobalPersonName.etx`
- `GlobalPersonName.Global.etx`

See also

- For a list of entities that implement `Obfuscator`, see "Implementing the Obfuscator interface in an entity" on page 233

ContactManager entity domain graph

ContactManager uses an `ABContact` entity domain graph to define the aggregate cluster of associated objects that it treats as a single unit for purposes of data destruction.

Overview of the ContactManager entity domain graph

The aggregate cluster of associated objects in an entity domain graph has a root and a boundary.

- The *root* is a single specific entity that the aggregate cluster contains. The root entity is the main entity in the graph. A root entity is application-specific and must implement `DestructionRootPinnable`. *Pinnable root* is another term for one of these root entities.

  In the base configuration of ContactManager, the root entity is `ABContact`.
- The *boundary* defines what is inside the aggregate cluster of objects. It identifies all the entities that are part of the graph.

In ContactManager, the boundary defines the entities that relate to an `ABContact` object, such as `Address`, `ABContactSpecialistService`, `ABContactContact`, and so on.

The unit of work for the destruction process is a single instance of the domain graph, such as a single `ABContact` object and all its associated entities.

To enforce the boundaries of the domain graph, all objects participating in the destruction process must implement the `Extractable` delegate.

You cannot define a new entity domain graph for use in personal data destruction. However, you can configure an existing entity domain graph. For example, use the `archivingOwner` data model attribute and the `CrossDomainPublicID` data model tag.

At server startup, the entity domain graph is validated. If the domain graph fails validation, the action taken depends on whether or not personal data destruction is enabled.

- If the configuration parameter `PersonalDataDestructionEnabled` is set to `true`, a failed graph validation prevents the server from starting. In this case, you must make graph corrections. Without a proper graph, `ABContact` purge will not work.
- If the configuration parameter `PersonalDataDestructionEnabled` is set to `false`, a failed graph validation logs warning messages indicating that a graph is invalid and needs correction. However, the server does start up. If you do not use personal data destruction, you can safely ignore these warnings.

ContactManager ABContact entity domain graph

The root of the ContactManager entity domain graph is `ABContact`. The entity domain graph includes entities such as `Address`, `ABContactSpecialistService`, `ABContactContact`, and so on. In the base configuration, this entity graph is used for purging `ABContact` data. ContactManager does not support archiving.

ContactManager defines specifics of handling the `ABContact` entity domain graph itself in the Java class `ABDomainGraphSupport`. For example, this class has methods like `register`, `getDomainGraphName`, `getLinkFromRootInfoToRoot`, and so on. You are not likely to need to use any of the methods in this class.

The class `ABContactPurgeMethodsImpl` is provided for special handling of entities that are not directly part of the domain graph but need to be considered in a purge. For example, an entity has a foreign key pointing to an entity in the domain graph, but there is no reciprocal foreign key from the entity in the domain graph.

ABContact domain graph and related entities

There are several ways that an entity can be considered part of the `ABContact` domain graph. There are also entities that require special handling or are explicitly excluded from the graph.

- The entity is included in the purge graph because it implements `Extractable`.

  See “Entities That Implement `Extractable`”.
- The entity is included in the purge graph because it has a foreign key to `ABContact`.

  See “Entities with Foreign Keys to an `ABContact` Object”.
- The entity might qualify for purging, but requires special handling.

  See “Entity Requiring Special Purge Handling”.
- The entity is a shared resource and is therefore excluded from the purge graph.

  See “`ABContact` Graph and Entities that Are Shared Resources”.

Entities that implement Extractable

The following entities are defined as part of the domain graph and are automatically purged along with `ABContact`:

- `PendingContactChange`
- `PendingContactCreate`

- `PendingContactUpdate`
- `ABContactCategoryScore`
- `ContactHistory`
- `TrackedChange`
- `ABContactTag`
- `ABContactSpecialistService`
- `ABContactDocumentLink`
- `EFTData`
- `ReviewSummary`
- `ReviewSummaryCategoryScore`
- `Address`

  Addresses are connected to the domain graph in two ways, directly defined in `ABContact` as a `PrimaryAddress` or as part of `ABContactAddress`. Addresses are not a shared resource among `ABContact` objects and are exclusive to the `ABContact` they are associated with, so they can be safely purged along with the `ABContact`.
- `ABContactAddress`
- `ABContactContact`

  Acting as a specialized edge foreign key, this object references `ABContact` through the `SrcABContactID` and the `RelABContactID` fields. In the base configuration, these fields are cross domain links, which can be safely severed from the other contacts.
- `MergeContactPair`

  Merge contact pairs are automatically purged along with `ABContact`, which will automatically severs relationships between that contact and any contacts that are duplicates. There is special handling for a contact that is marked as a retired `MergeContactPair` object.

  `MergeContactPair` objects represent a sort of versioning system for contacts. In the system there can be a number of contacts that, while different objects, really represent the same thing. The system has the ability to denote one version of the contact as the *kept* version, the official representation of the contact. The other versions are demoted and retired, but the system is able to keep track of them. These obsolete versions are called *retired* versions, and you can think of them as replaced by the kept version. `MergeContactPair` entities are used by ContactManager to help represent this kept and retired versioning.

  There are special conditions that apply to the `ABContact` object itself when a `MergeContactPair` exists:

  - If the contact being purged is the kept contact, then the purge destroys not just it, but also any retired versions of that contact.
  - If the contact being purged is a retired contact, attempting to purge that contact causes an exception to be thrown. The exception informs the user that deleting the contact is prohibited, and that deleting the contact requires explicitly purging the kept version of that contact.

Entities with foreign keys to an ABContact object

The following entities have foreign keys that point to an `ABContact` object, but there is no foreign key from the `ABContact` object to the entity. These entities do not implement `Extractable` and are purged explicitly by ContactManager along with `ABContact`.

- `ABContactScoringWorkItem`

  Deleted during the purge by ContactManager.
- `DuplicateContactWorkItem`

  Any that reference the contact are purged by ContactManager. Even though the entity references `ABContact` by using a non-nullable foreign key, deleting the entity is not flagged as problematic by the graph validator.

- `MessageHistory`

  Deleted during the purge by ContactManager.
- `Message`

  Deleted during the purge by ContactManager.

Entity requiring special purge handling

The `DuplicateContactPair` entity requires special handling. The `ABContact` entity cannot be purged if the contact continues to be referenced by any `DuplicateContactPair`.

`ABContact` is referenced by two non-nullable foreign keys, `ContactID` and `DuplicateContactID`. These two foreign keys are marked as cross domain, which makes `DuplicateContactPair` severable.

Purging will fail and throw an exception if an `ABContact` is referenced by any `DuplicateContactPair` entity, either as the contact or as the duplicate contact. A user of ContactManager must use the **Merge Duplicates** screens to resolve duplicates first. Only when there are no references will purging be allowed.

ABContact graph and entities that are shared resources

Resources that are shared can also be system or administration entities. The distinction does not matter. In terms of the domain graph, these objects are shared resources and are not part of the graph. These objects are not purged and remain in the system under all conditions. The `ABContact` object has the following shared resources:

- `DuplicateContactBatchRun`

  These objects are indirectly part of the `ABContact` graph.
- `SpecialistService`
- `SpecialistServiceParent`
- `Document`

  A document can be purged only if it is not shared by another `ABContact` object. Document metadata entities can be shared by one or more contacts. If only a single contact is related to a document, then the `Document` entity is purged along with the contact. Purging the document in this case is required by a database consistency check in ContactManager.

Table ab_purgedrootinfo

Whenever ContactManager purges an `ABContact`, it deletes the `ABContact` domain graph instance for that contact. To track the contacts it purged, ContactManager creates a `PurgedRootInfo` entity instance at the same time that it purges a contact. This entity records the `publicID` of the purged contact. ContactManager then stores each `PurgedRootInfo` instance as a row in the table `ab_purgedrootinfo`.

Note: ContactManager does not automatically delete `PurgedRootInfo` rows from the table. If you have a high purge volume, Guidewire recommends that you create a batch process to delete old, unwanted `PurgedRootInfo` instances from the `ab_purgedrootinfo` table.

Extensions of ABContact and other entities and the domain graph

If you have extended the `ABContact` data model, personal data purge is able to handle your extensions as long as you address domain graph design guidelines.

You can also create a variety of relationships between entities through foreign keys. Again, to support personal data purge, follow the guidelines for managing entities through the domain graph.

Following are some general guidelines:

- An entity is referenced by the `ABContact` or is part of the direct contact graph. The entity must implement `Extractable`, which makes the entity part of the contact graph. The entity will automatically be destroyed along with the rest of the contact graph when the personal data purge initiated. This guideline also applies to entities that are used as array objects.

- An `ABContact` object or other entity in the contact graph has a foreign key pointing to an entity, but the entity being pointed to has no reciprocal foreign key. The entity being pointed to is not part of the graph and is not deleted as part of personal data purge. Special handling is optional. You can add special handling to perform cleanup during the personal data purge.
- An entity has a foreign key that points to an `ABContact` entity or another entity that is part of the domain graph. However, there is no reciprocal foreign key from the domain graph entity. In this case, special handling is required because deleting the domain graph object leaves nothing for the original entity's foreign key to reference.

You can code your special handling in the `ABContactPurgeMethodsImpl` class.

ContactManager Data Protection Officer

A Data Protection Officer is expected to be available to handle problems with data destruction. Therefore, in the base configuration, there is a Data Protection Officer role to which users can be assigned. The code name for this role is `data_protection_officer`.

Data Protection Officer permissions

In the base configuration of ContactManager, the Data Protection Officer role has the following permissions relating to personal data destruction:

- `requestcontactdestruction`
- `editobfuscatedusercontact`

This role also has additional permissions to enable the user to work with users and groups. These permissions include `groupcreate`, `groupdelete`, `groupedit`, `usereditattrs`, `usereditlang`, `useredit`, `usergrantroles`, `userviewall`, `grouptreereview`, `grouppreview`, and `userview`.

A user named DataProtection Officer with login `dpofficer`, which has the Data Protection Officer role, is available in the sample data.

In the base configuration, ContactManager screens prevent a user from editing obfuscated user contacts if the user does not have permissions to do so. In addition, ContactManager prevents a user without the correct permissions from adding obfuscated user contacts to or removing them from groups and roles. You can create additional permissions and configure ContactManager to further limit editing of obfuscated objects.

Notifying the Data Protection Officer

The `PersonalDataDestruction` plugin interface provides a method that enables notification of the Data Protection Officer, `notifyDataProtectionOfficer`.

For example, in the base configuration, the class that implements the `PersonalDataDestruction` plugin interface, `ABPersonalDataDestructionSafePlugin`, overrides the `notifyDataProtectionOfficer` method. In this class, the `notifyDataProtectionOfficer` method logs messages to the system console if a destruction request fails.

Note: The class `ABPersonalDataDestructionSamplePlugin` has the same implementation for the `notifyDataProtectionOfficer` method.

Data destruction purge configuration

Purging is the process of completely removing `ABContact` data from the ContactManager database. There can be multiple objects associated with a contact that are also removed as they are detected by traversing the entity domain graph.

Note: ContactManager is expected to be the last Guidewire application web service called by the external system to process a personal data destruction request. When ContactManager receives a contact destruction request from the web service, it makes calls to the core application implementation of `ContactAPI.isContactDeletable` to determine if the contact can be destroyed. The contact will be destroyed in ContactManager only if all the applications return a positive response saying the contact can be destroyed.

See also

- “ContactManager entity domain graph” on page 225
- “Data obfuscation in ContactManager” on page 233

Defining the Destroyer

The `ABPersonalDataDestroyer` class implements the `PersonalDataDestroyer` interface and provides methods that determine how a destruction request is carried out. The class provides methods that are called by the `PersonalDataDestructionAPI` web service and its work queues.

A personal data destruction request to delete a contact by `AddressBookUID` corresponds in ContactManager to one `ABContact` with a single `LinkID` value. The method `translateABUIDToPublicIDs` finds the `PublicID` value of the contact and returns it to the web service, which creates a `PersonalDataContactDestructionRequest`. This personal data destruction request enables the work queue to make a `destroyContact` method call to delete the contact by `PublicID` when the work item is processed.

If a personal data destruction request specifies deleting a contact by `PublicID`, ContactManager can directly use the `PublicID` to delete the `ABContact` object.

The following two methods are the primary Destroyer methods used by the web service.

translateABUIDToPublicIDs

This public method overrides the `PersonalDataDestroyer` interface method `translateABUIDtoPublicIDs`. It finds the `ABContact` with a `LinkID` that is the same as the specified `AddressBookUID` and returns the contact’s `PublicID` value.

The method is called by the web service `PersonalDataDestructionAPI`. The web service uses the method to determine if an `AddressBookUID` exists and to get the `PublicID` if the original destruction request was specified by `AddressBookUID`.

destroyContact by Destruction Request

This public method overrides the `PersonalDataDestroyer` interface method `destroyContact`. It is the main entry point for destroying contacts.

> **Note:** The `PersonalDataContactDestructionWorkQueue` class calls this method when it processes a work item. The class calls `PersonalDataDestructionController.destroyContact(contactPurgeRequest)`.

This method takes a `PersonalDataDestructionRequest` and finds the corresponding `ABContact` by `PublicID`.

Once the `ABContact` is found, a call to the `PersonalDataDestruction` plugin is made to determine whether the `Contact` can be purged.

- If the result of the plugin call is `MUST_NOT_DESTROY`, the request is not processed for destruction. The reason is logged in the `DATA_DESTRUCTION_REQUEST` logger and the Destroyer returns the status `ContactDestructionStatus.TC_NOTDESTROYED`.
- If the result of the plugin call is `MAY_DESTROY` or `MUST_DESTROY`, the request is processed for destruction and `destroyContact(contact)` is invoked.
  - If the purge is successful the Destroyer must return the status `ContactDestructionStatus.TC_COMPLETED`.
  - If there is a purge error, the Destroyer must return the status `ContactDestructionStatus.TC_MANUALINTERVENTIONREQUIRED` and log the error or exception in the `DATA_DESTRUCTION_REQUEST` logger.

    > **Note:** The `notifyDataProtectionOfficer` method uses the error log level if there is a purge error. Otherwise it just uses the info log level.

It is possible that the contact was previously purged and cannot be found. For example, a previous `PersonalDataDestructionRequest` resulted in the `ABContact` being purged. In that case, the method returns `ContactDestructionStatus.TC_COMPLETED`.

See also

- “Data destruction web service” on page 217

PersonalDataPurge event

When a personal data purge of certain objects is committed, ContactManager generates a `PersonalDataPurge` event that you can respond to in an Event Fired rule to send a message. For example, you might want to send a message to a downstream system.

ContactManager generates a `PersonalDataPurge` event when an `ABContact` purge is committed as part of a personal data purge. In the base configuration, there is no rule that responds to this event, but you can create an `EventFired` rule in the `EventMessage` rule set category that sends a message.

Your rule could take the following form in the Guidewire Studio Rules editor:

`USES:`

`CONDITION (messageContext : entity.MessageContext):`

```
return messageContext.EventName == "PersonalDataPurge"
```

`ACTION (messageContext : entity.MessageContext, actions : gw.Rules.Action):`

```
var claimInfo = messageContext.Root as ABContact
messageContext.createPersonalDataPurgeMessage(
    "("ABContact with public ID ${contact.PublicID} has been successfully purged")
```

See also

- “ContactManager messaging events” on page 308
- For general information on messaging and message events, see the *Gosu Rules Guide*

Specifying which objects in the graph can be destroyed

The `PersonalDataDestruction` plugin interface provides basic methods that define which objects can be destroyed and how to contact the Data Protection Officer. You define a plugin implementation class to define this behavior and register it in the `PersonalDataDestruction.gwp` plugin registry.

PersonalDataDestruction plugin interface

This interface provides the following methods for implementation by a ContactManager plugin implementation class:

```
    PersonalDataDisposition shouldDestroyRoot(
           Pinnable root,
           Collection<Pinnable> descendants, Pinnable origin);
    PersonalDataDisposition shouldDestroyUser(UserContact userContact);
    void notifyDataProtectionOfficer(
           Pinnable root, String title, String message, Date dateOfError);
    void notifyExternalSystemsContactHasBeenPurged(
           String AddressBookUID, String requestor, String requestID);
    PersonalDataPurgeContext createContext(PersonalDataPurgeContext context);
    void prepareForPurge(PersonalDataPurgeContext context);
    void postPurge(PersonalDataPurgeContext context);
    PersonalDataDestroyer getDestroyer();
```

In the base configuration, ContactManager registers the `ABPersonalDataDestructionSafePlugin` class as the default class that implements `PersonalDataDestruction`, which prevents destruction of any of the pinnable root entities. A more realistic starting point for your purge definition is the `ABPersonalDataDestructionSamplePlugin` class.

See also

- “Personal data destruction plugin implementation classes” on page 232
- “ContactManager entity domain graph” on page 225

Personal data destruction plugin implementation classes

In the base configuration of ContactManager, the `ABPersonalDataDestructionSafePlugin` class is registered as the class that implements the `PersonalDataDestruction` plugin interface. This class provides default handling for destruction of pinnable root entities in the base configuration. It prevents destruction of any personal data.

`ABPersonalDataDestructionSamplePlugin` is the class you can use as an example when you implement your own personal data destruction class to define both `getDestroyer` and how specific pinnable roots are handled. You must then register your implementation class with the plugin registry `PersonalDataDestruction.gwp`.

These two classes define methods that control destruction of pinnable root entities by returning one of the values defined in the `enum PersonalDataDisposition`:

- `MUST_NOT_DESTROY` – The object must not be destroyed. If this value is in conflict with a `MUST_DESTROY` value in the domain graph, the Data Protection Officer must get involved.
- `MUST_DESTROY` – The object must be destroyed.
- `MAY_DESTROY` – The object can be destroyed.

ABPersonalDataDestructionSafePlugin

In the base configuration, `ABPersonalDataDestructionSafePlugin` calls `getDestroyer` to obtain the destroyer defined in `ABPersonalDataDestroyer`. Additionally, this class prevents data destruction by returning `MUST_NOT_DESTROY` for all calls to destroy pinnable root entities. For example:

```
override function shouldDestroyRoot(
    root: DestructionRootPinnable,
    descendants: Collection<DestructionRootPinnable>,
    origin: DestructionRootPinnable): PersonalDataDisposition
{
  notifyDataProtectionOfficer(
        root, "Safe plugin implementation always returns MUST_NOT_DESTROY")
  return MUST_NOT_DESTROY
}
override function shouldDestroyUser(
    userContact: UserContact): PersonalDataDisposition
{
  notifyDataProtectionOfficer(
        userContact, "Safe plugin implementation always returns MUST_NOT_DESTROY")
  return MUST_NOT_DESTROY
}
private function notifyDataProtectionOfficer(
    contact : DestructionRootPinnable, message : String)
{
  notifyDataProtectionOfficer(contact, null, message, null)
}
override function notifyDataProtectionOfficer(
    root: DestructionRootPinnable, title: String, message: String, dateOfError: Date)
{
  ABPersonalDataLogUtil.logInfoNotDestroyed(root, message)
}
```

ABPersonalDataDestructionSamplePlugin

You can use the class `ABPersonalDataDestructionSamplePlugin` as a guide for writing your own personal data destruction code. This class can return values other than `MUST_NOT_DESTROY` for a pinnable root entity.

The class returns `MUST_NOT_DESTROY` in the following circumstances:

- The contact is an `ABContact` with `DoNotDestroy` set to `true`.
- The subtype of the contact is `ABCompany` or `ABPlace`.
- Core applications were checked for permission to destroy the contact, and at least one application did not permit the destruction.

The class returns `MUST_DESTROY` if the contact:

- Is an `ABContact` for which `DoNotDestroy` is `false`, the subtype is not `ABCompany` or `ABPlace`, and all core applications permit destruction of the contact.
- Is a `UserContact`.

See also

- “Do Not Destroy flag” on page 223
- “DestructionRootPinnable delegate” on page 223
- “ContactManager entity domain graph” on page 225

Data obfuscation in ContactManager

In general, personal data destruction in ContactManager is done through removal of database records. However, the entities associated with an employee who works in your installation of ContactManager are not conducive to deletion because of the way data creation and changes are recorded. In particular, objects in the database are connected to the user that created them, and, in many cases, the user that last modified them.

Because an employee is likely to create or modify hundreds of thousands of objects, it would be computationally expensive to locate all those objects in the database. It would also be expensive to change those references to something else. It is not necessary to destroy the relationship between all the work that the employee performed and the fact that it was performed by a specific employee. If the employee's personally identifiable data is destroyed, the set of objects associated with the employee can remain and not violate the need to destroy personal data.

Therefore, in the base configuration, ContactManager obfuscates data related to `UserContact` objects.

Obfuscated objects

Each object can detect if it has been obfuscated or not through its `Obfuscated` flag. The system has no special handling for objects that have gone through obfuscation. Obfuscated objects act like any other active object in the system regarding search results, batch processes, and so on. You can implement additional functionality to filter obfuscated beans, according to ContactManager configuration capabilities. In your obfuscation implementation, you must take into account how your custom obfuscation might affect existing processes in ContactManager.

Preupdate rules

Data obfuscation works the same as a normal entity editing, so changes made during obfuscation will trigger preupdate rules for entity types that have rules registered.

Implementing the Obfuscatable delegate in an entity

To be obfuscatable, an entity must implement the `Obfuscatable` delegate. If the entity implements `DestructionRootPinnable`, the `DoNotDestroy` field is set to `false` by default, which enables the entity to be obfuscated.

If an entity implements this delegate, the fields to be obfuscated must be tagged `PersonalData`.

> **Note:** Tagging is not supported for array references, nor does obfuscation cascade automatically through foreign keys. Arrays and cascading through foreign keys must be handled in Gosu code in the `Obfuscator` implementation class.

See also

- “Marking entity fields for obfuscation” on page 225
- “Implementing the Obfuscator interface in an entity” on page 233
- “Obfuscatable delegate” on page 224
- “Do Not Destroy flag” on page 223

Implementing the Obfuscator interface in an entity

To be obfuscatable, an entity must implement the `Obfuscator` interface and specify an implementation class other than `UnsupportedObfuscator`. For example, use a class that extends `DefaultPersonalDataObfuscator`.

The entities that have `Obfuscator` implementations that support obfuscation are:

- `Credential` – `CredentialDefaultObfuscator`

  Has fields and typekeys marked `PersonalData`.
- `OfficialID` – `DefaultPersonalDataObfuscator`

  Has fields and typekeys marked `PersonalData`.
- `User` – `UserDefaultObfuscator`

  Has fields and typekeys marked `PersonalData`.
- `UserContact` – `UserContactDefaultObfuscator`

  This entity is a subtype of `Person`, which is a subtype of `Contact`. It inherits the `Contact` implementation of the `Obfuscatable` delegate and overrides the `Contact` implementation of the `Obfuscator` interface. In the base configuration, the `EmployeeNumber` field is marked `PersonalData`.

The following entities implement the `Obfuscatable` delegate, and in the base configuration their `Obfuscator` interface implementation is `UnsupportedObfuscator`:

- `Address`
- `Contact` – Has fields and typekeys that are marked `PersonalData`.
- `ContactCategoryScore`
- `Contact.Global`
- `Person` – Inherits obfuscation settings from `Contact`. Has fields and typekeys that are marked `PersonalData`.

See also

- "Obfuscator interface" on page 224

Personal data obfuscation classes

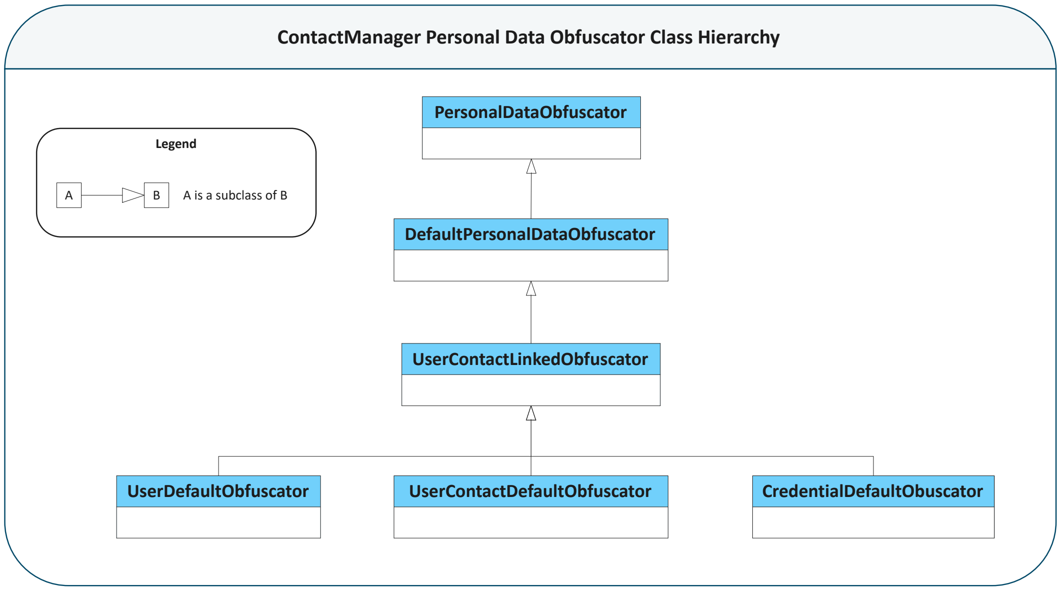
The personal data obfuscation classes, all of which ultimately inherit from `PersonalDataObfuscator`, define obfuscation for specific entities. The specific class that an entity uses is defined in its `implementsEntity` element for `gw.api.obfuscation.Obfuscator`.

See also

- "Implementing the Obfuscator interface in an entity" on page 233
- "Obfuscator interface" on page 224
- "Marking entity fields for obfuscation" on page 225

Personal data obfuscation class hierarchy

ContactManager provides the following class hierarchy for personal data obfuscation:


ContactManager Personal Data Obfuscator Class Hierarchy
Legend
A
B
A is a subclass of B
PersonalDataObfuscator
DefaultPersonalDataObfuscator
UserContactLinkedObfuscator
UserDefaultObfuscator
UserContactDefaultObfuscator
CredentialDefaultObuscator

`DefaultPersonalDataObfuscator` and its subclasses define base configuration behavior to enable obfuscating `User`, `UserContact`, and `Credential` and related objects. For example:

- `DefaultPersonalDataObfuscator` overrides the method `getObfuscatedValueForPersonalDataField` and defines default obfuscation behavior for fields that are tagged `OfuscateUnique`. The method handles `ObfuscateUnique` by calling `PersonalDataObfuscatorUtil` to get an MD5 hash value for the field.
- `UserContactLinkedObfuscator` overrides the `shouldObfuscate` method. That method calls the `shouldDestroyUser` method defined in the `PersonalDataDestruction` plugin to get the setting for `UserContact`. For example, in the base configuration the setting is `MUST_NOT_DESTROY`.
- `UserDefaultObfuscator` overrides a `beforeObfuscate` method. That method throws an exception if the user's credential is active because that means the user is still active. If the user is not active, the method calls `user.Credential.obfuscate` and `user.Contact.obfuscate` and removes the arrays of join entities `AttributeUser` and `UserRegion`.
- `UserContactDefaultObfuscator` attempts to obfuscate or remove any contact addresses, official IDs, category scores, and tags associated with the `UserContact` that can be destroyed.
- `CredentialDefaultObfuscator` overrides the `beforeObfuscate` method, which stops the obfuscation if the credential is active.

PersonalDataObfuscator

`PersonalDataObfuscator` is the parent class for the obfuscator classes. It is a general class that obfuscates the fields for any entity. You extend this class or one of its subclasses when implementing obfuscation for entities that do not have obfuscator classes defined.

`PersonalDataObfuscator` handles setting the obfuscation for marked fields. If you have tagged all the entity fields `PersonalData` with value `ObfuscateDefault`, and there is no need to do any special preprocessing of the fields, you can use this class.

The following methods are provided:

- `isObfuscated` – Checks the field `ObfuscatedInternal` and returns `true` or `false` depending on the value.
- `obfuscate` – Finds all the columns that are marked for obfuscation on this object.
  - The method sets the field value with the `obfuscatedValue`.
  - If the column is marked `PersonalData` with a value of `ObfuscateDefault`, the method sets the value to either `null` or the default value.

- ◦ If the column is marked `PersonalData` with a value of something other than `ObfuscateDefault`, the method calls `getObfuscatedValueForPersonalDataField`. In the base configuration, `getObfuscatedValueForPersonalDataField` is defined in `DefaultPersonalDataObfuscator` and uses the tag value. You can override the definition of this method to change how it obfuscates the field.
- ◦ At the end, the method sets the `ObfuscateInternal` column to `true` by using `ObfuscatablePublicMethod.setObfuscated`.

- `obfuscateSimple` – Works the same as `obfuscate`.

If you want to do preprocessing before obfuscation, you can extend `PersonalDataObfuscator` or a subclass of this class and override the `beforeObfuscate` method. If you want to change the value of the personal data field before obfuscation, you can override `getObfuscatedValueForPersonalDataField`.

See also

- "Marking entity fields for obfuscation" on page 225

UnsupportedObfuscator

This Java class provides a default implementation for any entity that implements the interface `Obfuscator` and declares `UnsupportedObfuscator` as the implementation. When `obfuscate` is called, this class throws `unsupportedOperationException` for the field.

PersonalDataObfuscatorUtil

This Gosu class is in the same package as the personal data obfuscation classes, `gw.personaldata.obfuscation`. The class implements the method `computeMD5Padding`. If a personal data field has a `PersonalData` tag with value `ObfuscateUnique`, this method is called to obfuscate the field.

The method computes an MD5 `String` based on the type of entity and the `PublicID`, and then returns that string so the field can be obfuscated with that value.

For example, `DefaultPersonalDataObfuscator` calls this method in its `getObfuscatedValueForPersonalDataField` method when the field's `PersonalData` tag has the value `PersonalDataTagValue.TC_OBFUSCATEUNIQUE.Code`.

See also

- "Marking entity fields for obfuscation" on page 225

chapter 13

ClaimCenter service provider performance reviews

Overview of service provider performance reviews

You can request services in a claim from vendor service providers such as body shops, assessors, attorneys, and medical clinics. You can evaluate your service providers by gathering review information on them in service provider performance reviews.

Note: Service provider performance reviews are available only if ClaimCenter is integrated with ContactManager.

ClaimCenter enables you to evaluate your service providers, which are vendor contacts, by gathering review information on them. Having this information helps in selecting the best providers, controlling your claim costs, increasing customer satisfaction, and increasing claim processing efficiency.

The central element of service provider performance reviews is the review. A review consists of sets of scorable questions in a questionnaire evaluating a vendor for work performed on a specific claim. You use the review's score to rank vendors and determine which vendors you will use.

Note: ContactManager aggregates performance scores only from questionnaires that evaluate performance of vendors and not from service request metrics.

In particular, you can:

- Conduct post-service reviews on any type of vendor subtype of `Contact`.
- Score each review as part of the claim associated with the vendor’s work.
- Score each vendor by combining its individual review scores.

After you have created reviews for your vendors, you can:

- Define lists of preferred vendors based on their past performance, as quantified by their reviews.
- Search for nearby vendors with high review scores.
- Assign nearby and high-rated vendors to provide services.
- Remove poorly performing vendors and steer business to high performers.
- Negotiate contracts with vendors for future services based on objective past performance standards.

See also

- “Integrating ContactManager with Guidewire core applications” on page 21
- “Vendor services” on page 163
- For information on services in ClaimCenter, see the *Application Guide*

Service providers

A service provider is a vendor contact (separate from your insurance company) that provides a service during the resolution of a claim. For example

- Auto repair shop, auto glass shop, auto body shop
- Property assessor, auto damage assessor
- Attorney, law firm
- Medical clinic, home care worker, nurse, physician, physical therapist

Note: The sample data available in the base product includes a review for auto repair shops.

Typically, you contact the service provider and request services, but the insured party or another claim contact can also contact a service provider directly.

ClaimCenter considers all service providers to be vendor contacts and stores them in ContactManager. You can categorize your service providers. For example, you create different `Contact` types for auto glass repair, auto body repair, and transmission repair, and then create reviews containing sets of questions using the appropriate vendor types.

See also

- “Data model for service provider performance reviews” on page 244
- “Configure service provider performance reviews” on page 247

Reviews of service providers

ClaimCenter scores each review and stores it with the associated claim. When a review is completed, ClaimCenter sends the results to ContactManager. ContactManager averages the review’s overall and category scores with the scores of all reviews for the service provider and saves the result with the service provider’s contact information.

Searching for service providers can return results from ContactManager that include these average scores. For example, you might specify Auto Repair Shop contacts and an accident site location in your search. ClaimCenter then provides a list of auto repair shops close to the accident site ranked by average review score.

Note: In the base configuration, ClaimCenter is not set up to send review scores to ContactManager, and ContactManager is not set up to process reviews. To configure this behavior, you enable two separate work queues to run, one in each application.

See also

- “Using service provider performance reviews” on page 238

Using service provider performance reviews

You use the features of service provider performance reviews in ClaimCenter. From the ClaimCenter **Contacts** screen, you can work directly with reviews on claims. You can view, edit, complete, or delete them. You can use the ClaimCenter **Search** tab to search ContactManager for contacts with review scores above a minimum value that you set. You can also view the full `ReviewType` review form and see each contact’s score averages. The average scores include both review scores and scores by category for all review types associated with the contact.

Working with reviews

The sample data includes a service provider performance review for an auto repair shop contact. You can extend ClaimCenter and ContactManager to provide reviews for other types of service providers.

The ClaimCenter **Reviews** card is the starting point for working with reviews. You can access it by opening a claim and navigating to **Parties Involved** > **Contacts**.

See also

- "Configure service provider performance reviews" on page 247

Begin a service provider performance review

You start a review of a service provider in ClaimCenter from a claim on the **Contacts** screen.

Procedure

1. Navigate to a claim.
2. Select **Parties Involved** > **Contacts**.
3. Select the contact you want to review, and then click the **Reviews** tab and click **Edit**.
4. If there are review types defined for the contact's type, you can click **Add New Review**.

   Review types are based on the service provider's `Contact` subtype.
5. Select a review type from the drop-down list.

Results

After starting a review, you can answer the review questions at any time. If you navigate away from the review, ClaimCenter saves the review with its current answers. You can return to a review, and, if it is not complete, you can reopen the review and answer additional questions or edit your answers.

What to do next

See also

- "View or edit a service provider performance review in ClaimCenter" on page 239
- "Complete a service provider performance review" on page 240

View or edit a service provider performance review in ClaimCenter

Before you begin

Before you can view or edit a review, you must have started one, as described in "Begin a service provider performance review" on page 239.

About this task

After a review is complete, you can view it, but you cannot edit it. See "Complete a service provider performance review" on page 240.

Procedure

1. Select a claim and click **Parties Involved** > **Contacts**.
2. Select an auto repair shop from the list and then click **Reviews** in the contact details screen below the list to show a list of reviews.

   The **Reviews** tab appears only if a review type is defined for that type of contact.
3. Select a review.
4. You can see the details of the review below the list of reviews.
5. If the review is not complete, you can click **Edit** to continue filling it out.

Complete a service provider performance review

Before you begin

Before you can complete a review, you must have created one, and you must be able to open the list of reviews. See "View or edit a service provider performance review in ClaimCenter" on page 239.

About this task

After you finish editing a review in ClaimCenter, you must complete it. Completing a review closes it for further editing and saves it in ContactManager, which updates the average scores for the contact.

Procedure

1. Display a list of reviews.
2. Check the check box for each review you want to complete.
3. Click **Complete Selected** at the top of the list.

Results

ClaimCenter calculates the review's scores and sends the result to ContactManager, which uses them to calculate the vendor contact's average scores. ClaimCenter does not immediately change the vendor's scores to reflect the new scores. See "Scheduling service provider performance reviews for processing" on page 241.

View a vendor's service provider performance scores in ContactManager

Before you begin

Before you can see a vendor's service provider performance review scores in ContactManager, at least one review has to have been completed in ClaimCenter. Additionally, ClaimCenter must have sent the review to ContactManager, and ContactManager must have processed the review. See "Complete a service provider performance review" on page 240. See also "Scheduling service provider performance reviews for processing" on page 241.

Procedure

1. On the **Contacts** tab, click **Search** in the sidebar, and then search for a contact that has been reviewed in ClaimCenter.
2. Select the contact and click the **Reviews** tab.
3. Select a review.
4. You can see information on the review below the list of reviews.

Delete a service provider performance review

Before you begin

Before you can delete a service provider performance review, you must have created one, and you must be able to open the list of reviews. See "View or edit a service provider performance review in ClaimCenter" on page 239.

About this task

You can delete both complete and incomplete reviews in ClaimCenter. There are separate permissions for deleting incomplete reviews and completed reviews. You can use these permissions, for example, to allow adjusters to delete reviews that have been started but not scored. For a list of permissions, see "Service provider performance review permissions" on page 257.

Procedure

1. Display a list of reviews.
2. Click the check box for each review you want to delete.
3. Click **Delete Selected**.

Search for a vendor by review score in ClaimCenter or ContactManager

Before you begin

This topic requires that ContactManager has processed at least one service provider performance review for a vendor. See "Complete a service provider performance review" on page 240.

About this task

You can search for contacts by minimum service provider performance review score in ClaimCenter or ContactManager. On the search screen, you select a **Minimum Score** value from a drop-down list. You can configure the drop-down list by changing the PCF files.

Procedure

1. Log in to ClaimCenter or ContactManager.
   - In ClaimCenter, on the **Address Book** tab, click **Search**.
   - In ContactManager, on the **Contacts** tab, click **Search**.
2. Select a **Contact Type**.
3. If there is a review type for the contact type, you see the **Minimum Score** field. For example, if you have imported the sample data, selecting an Auto Repair Shop shows the **Minimum Score** list.
4. Enter the data required to search for the contact, such as company name, tax ID, or postal code.
5. Click **Search**.

Results

The results are review values that are equal to or greater than the search value. Search results include a column of overall scores. You can sort the list by clicking the column's title. Scores are not used in the default sort order.

Scheduling service provider performance reviews for processing

In the base configuration, ClaimCenter does not send service provider performance reviews to ContactManager, and ContactManager does not compute average scores and save them with the associated contact information. To get both sides of this feature working, you set up ClaimCenter and ContactManager to run batch processes.

You can optionally configure the work queues defined in the ClaimCenter and ContactManager `work-queue.xml` files.

You can also run the processes manually, as described in "Manually processing service provider performance reviews" on page 243.

See also

- For information on work queue schedule specifications, see the *Administration Guide*

Configuring service provider performance review work queues

Service provider performance reviews use the Guidewire work queue infrastructure and the process scheduler for asynchronous processing. The work queues are enabled by default in each application's `work-queue.xml` file and do not necessarily require configuration. To run the work queue writers asynchronously, you must schedule them.

ClaimCenter review sync work queue and writer

- The ClaimCenter `ReviewSync` writer collects reviews that have been completed but not sent to ContactManager, and then creates work items for them. The ClaimCenter `ReviewSync` worker processes each review indicated by a work item and sends it to ContactManager.
- The ClaimCenter `work-queue.xml` file configures one `ReviewSyncWorkQueue`. You can open this file in Guidewire Studio™ for ClaimCenter by navigating in the **Project** window to **configuration** > **config** > **workqueue** and double-clicking `work-queue.xml`. The entry in this file for `ReviewSyncWorkQueue` is as follows:

```
<work-queue
    workQueueClass="com.guidewire.cc.domain.contact.ReviewSyncWorkQueue"
    progressinterval="600000">
  <worker instances="1"/>
</work-queue>
```

ContactManager ABContact scoring work queue and writer

- The ContactManager `ABContactScoring` writer collects contacts that need to be updated and creates work items for them. The ContactManager `ABContactScoring` worker processes each contact indicated by a work item and updates the scores for each contact.
- The ContactManager `work-queue.xml` file configures one `ABContactScoringWorkQueue`. You can open this file in Guidewire Studio™ for ContactManager by navigating in the **Project** window to **configuration** > **config** > **workqueue** and double-clicking `work-queue.xml`. The entry in this file for `ABContactScoringWorkQueue` is as follows:

```
<work-queue
    progressinterval="600000"
    workQueueClass="com.guidewire.ab.domain.contact.ABContactScoringWorkQueue">
  <worker instances="1"/>
</work-queue>
```

See also

- “Schedule ClaimCenter review sync work queue” on page 242
- “Schedule ContactManager ABContact scoring work queue” on page 243
- For more information on work queues and the work queue scheduler, see the *Administration Guide*

Schedule ClaimCenter review sync work queue

About this task

You can schedule the ClaimCenter work queue that sends service provider performance reviews to ContactManager.

Procedure

1. Start ClaimCenter Studio.

   At a command prompt, navigate to the ClaimCenter installation folder and enter the following command:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **scheduler**.
3. Double-click `schduler-config.xml` to open the file in an editor.
4. Find the following the `ReviewSync` entry and remove the comment markers:

   ```
   <!--
     <ProcessSchedule process="ReviewSync">
       <CronSchedule minutes="20"/>
     </ProcessSchedule>
   -->
   ```

   This entry causes `ReviewSync` batch processing to run every hour at 20 minutes after the hour.

Schedule ContactManager ABContact scoring work queue

About this task

You can schedule the ContactManager work queue that processes service provider performance reviews sent by ClaimCenter.

Procedure

1. Start Guidewire Studio™ for ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **scheduler**.
3. Double-click `scheduler-config.xml` to open the file in an editor.
4. Find the following `ABContactScoring` entry and remove the comment markers:

   ```
   <!-- <ProcessSchedule process="ABContactScoring">
       <CronSchedule hours="0"/>
   </ProcessSchedule> -->
   ```

   This entry causes the `ABContactScoring` work queue to run at midnight every day.

What to do next

See also

- For more information on work queues and the work queue scheduler, see the *Administration Guide*

Manually processing service provider performance reviews

You can manually run the ClaimCenter and ContactManager work queues that handle service provider performance reviews. For example, you might want to test a change you have made.

See also

- “Overview of service provider performance reviews”
- “Data model for service provider performance reviews” on page 244
- “Configure service provider performance reviews” on page 247
- “Service provider performance review plugins” on page 258

Manually run completed review sync batch processing in ClaimCenter

About this task

If there are completed service provider performance reviews in ClaimCenter that you want to process immediately, you can run the batch process manually.

Procedure

1. Log in to ClaimCenter as an administrator, such as username `su` and password `gw`.
2. Press `Alt+Shift+T` and then click the **Server Tools** tab.
3. Click **Batch Process Info** in the sidebar.
4. Locate **ClaimCenter (SPM) Completed Review Sync** and click **Run** in the column on the right.

Manually run ABContact score aggregator batch processing in ContactManager

About this task

If there are service provider performance reviews in ContactManager that you want to process immediately, you can run the work queue manually.

Procedure

1. Log in to ContactManager as an administrator, such as username `su` and password `gw`.
2. Press `Alt+Shift+T` and then click the **Server Tools** tab.
3. Click **Batch Process Info** in the sidebar.
4. Locate **AB Contact Score Aggregator** and click **Run** in the column on the right.

Data model for service provider performance reviews

All reviews consist of the following parts:

Header

Contains information common to all `ReviewType` objects.

Question

The review is a set of questions. The questions can be divided into groups called categories.

Answer

After the review has been given, each question has an answer.

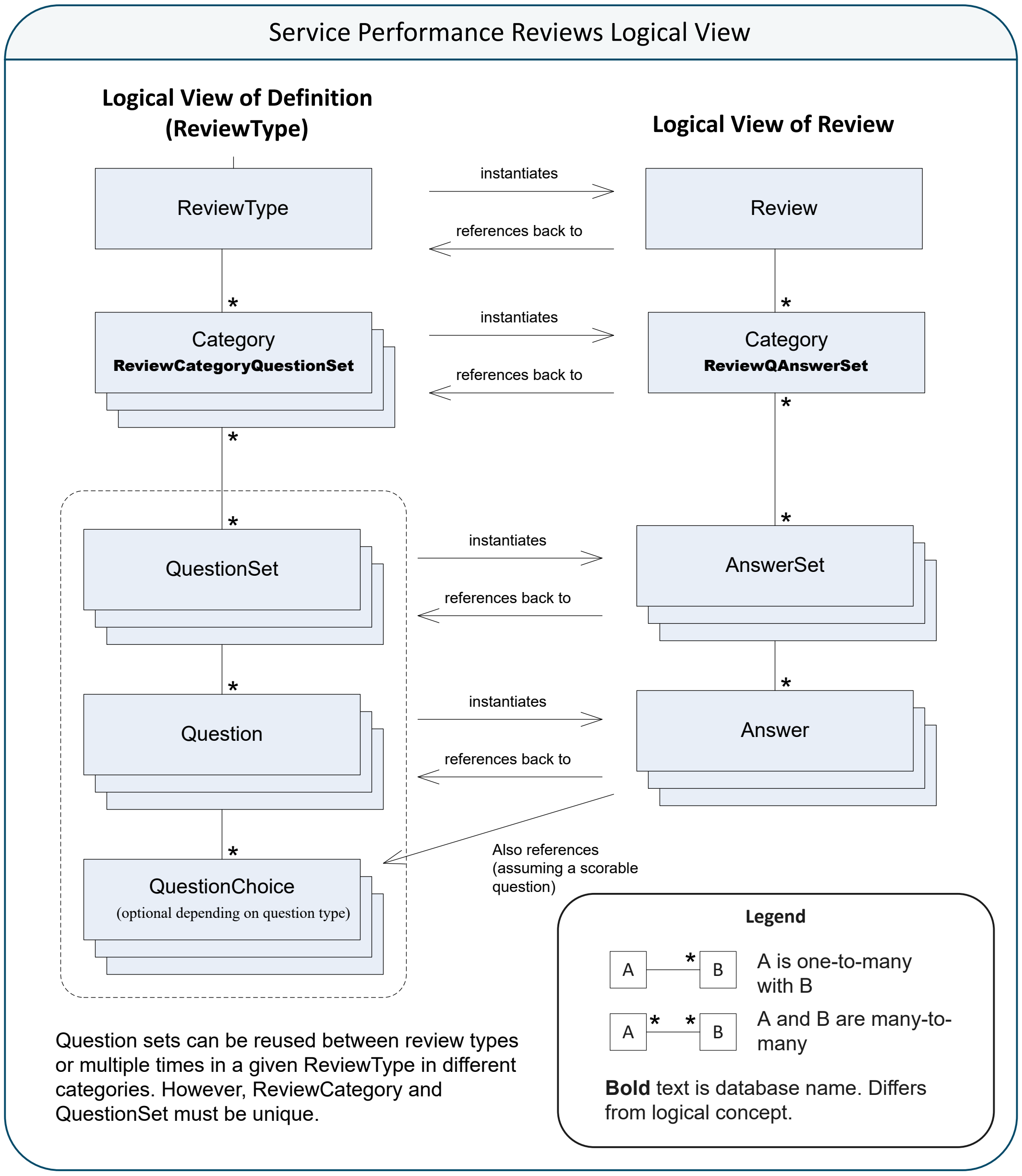
Score

After the review is complete, it is scored. There is a score for the entire review and a score for each category if the review has categories.

Service providers of any contact type or subtype can have review scores, both overall scores and category scores, associated with them. Additionally, each claim can store a review’s answers and scores. In general, questions, sometimes with answer choices, are grouped into sets of questions, which are further grouped into categories, which are associated with a review type. Finally, review types are associated with defined contact subtypes.

ClaimCenter service provider performance review entities

Two entities, `Review` and `ReviewType`, form the basis of the data model for ClaimCenter service provider performance reviews. On the left side of the following diagram are abstract entities used to create objects shown on the right side of the diagram. `ReviewType` and `Review` entities are related in the same way that activity patterns and activities are related.


Service Performance Reviews Logical View
Logical View of Definition (ReviewType)
Logical View of Review
ReviewType
Review
instantiates
references back to
Category
ReviewCategoryQuestionSet
Category
ReviewQAnswerSet
QuestionSet
AnswerSet
Question
Answer
QuestionChoice
(optional depending on question type)
Also references (assuming a scorable question)
*
Legend
A
B
A is one-to-many with B
A and B are many-to-many
Bold text is database name. Differs from logical concept.

Question sets can be reused between review types or multiple times in a given ReviewType in different categories. However, ReviewCategory and QuestionSet must be unique.

`ReviewType`

A digital questionnaire type, associated with a particular contact subtype and all its child subtypes, that you use to collect feedback about contacts. An example, included with each release, is an Auto Repair Evaluation.

`ReviewCategoryQuestionSet`

A join table that relates a `ReviewType` to a `QuestionSet` and assigns a `ReviewCategory` to the `QuestionSet`. You can divide a questionnaire into one or more sections, or categories. Categories of this form might include Quality of Work, Delivery, and General Satisfaction. Typecodes in the `ReviewCategory` typelist define categories to enable ClaimCenter and ContactManager to calculate scores for them.

`QuestionSet`

A group of questions.

`Question`

A member of a `QuestionSet`.

`QuestionChoice`

Optionally, potential answers to a particular question, such as true or false or a list of choices.

After someone creates a review, the system creates a parallel set of entities:

`Review`

One entire questionnaire, containing one or more `AnswerSet` objects.

`ReviewAnswerSet`

Pointers to filled-in `QuestionSet` objects, each of which is an `AnswerSet`.

`AnswerSet`

A set of answers for a `QuestionSet`.

`Answer`

One reply to a `QuestionSet`.

See also

- *Configuration Guide*

Typelists for service provider performance reviews

Two typelists, which must be present and identical in both ClaimCenter and ContactManager, affect service provider performance reviews:

`ReviewCategory`

The set of categories to which you can assign groups of questions, and from which ClaimCenter generates category scores.

`ReviewServiceType`

The set of possible service types to which the user can assign individual `Review` objects. Service types are a reporting category and appear in a drop-down list in the `Review` header.

ContactManager service provider performance review data model

In ContactManager, service provider performance reviews use the entity `ReviewSummary` and an extension of the ContactManager `ABContact` entity.

ReviewSummary entity

`ReviewSummary` is an entity that captures a summary of a `Review` object's information, sent from ClaimCenter for completed service provider performance reviews. A `ReviewSummary` entity contains the `Review` object’s header information, including the overall score, and a list of category scores. However, it does not include the actual questions and answers in the associated service provider performance review.

Each `ReviewSummary` belongs to a particular `ABContact` instance and uses a claim number to associate it with the claim from which it was obtained.

ABContact entity extensions for service provider performance reviews

Service provider performance reviews use two extensions of the `ABContact` entity in ContactManager.

`Score`

An `integer` column containing the overall score for this `ABContact`, calculated from `ReviewSummary` entities by the `ABContactScoring` work queue.

`CategoryScores`

An array of `ABContactCategoryScore` objects, containing the category scores for this `ABContact` instance, calculated from `ReviewSummary` entities by the `ABContactScoring` work queue.

Configure service provider performance reviews

To configure service provider performance reviews, export `questions.xml`, add, edit, or retire review types, and then import `questions.xml` back into ClaimCenter.

About this task

ClaimCenter provides one `ReviewType` in its sample data. To configure review types and their associated entities, you must log in as an administrator and export **Questions** data, initially from the sample data.

The exported XML file includes sample entities for `ReviewCategoryQuestionSet`, `QuestionSet`, `Question`, `QuestionChoice`, `QuestionFilter`, and `ReviewType`. You can make your changes in the exported XML file and import the changed file.

In addition to working with reviews, you can also create or reuse categories and associate them with `ReviewType` and `QuestionSet` objects.

See also

- For information on installing sample data, see the *Installation Guide*

Export questions and review types

Before you begin

To be able to export a service provider performance review, you must have previously imported sample data or a `questions.xml` file.

About this task

Review types for service provider performance reviews are grouped with questions in the administration **Export Data** screen. To configure questions, categories and review types, you must export them as a unit.

Procedure

1. Log in to ClaimCenter as an administrator. For example, log in with username `su` and password `gw`.
2. Click the **Administration** tab, and then navigate to **Utilities** > **Export Data** in the Sidebar.
3. At the top of the screen under **Export Administrative Data**, click **Export**.
4. Click **Data to Export** and choose **Questions** from the drop-down list, and then click **Export**.
5. If there is a prompt to save `questions.xml`, choose **Save**, and then choose a directory to save it to.

   You later import the file from this directory if you make changes to the file.
6. Open the saved file in an editor. The sample Auto Repair Service review type is at the bottom of the file, after the question and category definitions. The XML code for this element is:

```
<ReviewType public-id="SPMReviewType:1">
```

Add a service provider performance review type

Before you begin

To add a service provider performance review type, you must have previously exported sample review data or manually created a `questions.xml` file. See “Export questions and review types” on page 247.

Note: If you want to change an existing review type, you must retire it and add a new review type. See “Retire a Service Provider Performance Review Type” on page 248

Procedure

1. Find a `ReviewType` in the `questions.xml` file and copy it.
   For example, the following review type is the sample Auto Repair Service review type definition:

```
<ReviewType public-id="SPMReviewType:1">
  <ContactSubtype>AutoRepairShop</ContactSubtype>
  <Description>Sample Review Type for Auto Repair Shops</Description>
  <Name>Auto Repair Service</Name>
  <ShouldRetireFromImportXML>false</ShouldRetireFromImportXML>
</ReviewType>
```

2. Give the `ReviewType` a public ID, a name, and a description, and specify the vendor contact subtypes to which it applies.

   For a list of vendor contact subtypes in the ClaimCenter base application, see “ContactManager and core application contact entity hierarchies” on page 113.

   For example, the following review type defines a review for an auto towing company:

```
<ReviewType public-id="SPMReviewType:2">
  <ContactSubtype>AutoTowingAgcy</ContactSubtype>
  <Description>Review Type for Auto Towing Agencies</Description>
  <Name>Auto Towing Agency</Name>
  <ShouldRetireFromImportXML>false</ShouldRetireFromImportXML>
</ReviewType>
```

3. Open Guidewire Studio™ for ClaimCenter and Guidewire Studio™ for ContactManager, and add new `ReviewCategory` or `ReviewServiceType` typecodes or both, as needed.

   For information on adding these typecodes, see “Configuring review service type and review category typecodes” on page 250.

4. Use administrative file import to add the XML file containing the `ReviewType` to your installation.

   See “Import questions and review types” on page 249.

Retire a Service Provider Performance Review Type

Before you begin

To be able to retire a service provider performance review type, you must have previously exported a `questions.xml` file that has the review type in it. See “Export questions and review types” on page 247.

About this task

If you no longer need a review type, or if you want to change an existing one, you must retire it. You cannot delete or change an existing review type because there might be existing reviews that use it. To change a review type, you must retire it and add a new review type with a new public ID. After importing the XML file containing the retired review type, you can no longer use it to create new reviews in ClaimCenter. However, existing reviews that use the type continue to work.

IMPORTANT: You can retire only review types. Do not retire `QuestionSet`, `Question`, `QuestionChoice`, or `QuestionFilter` entities because existing reviews might still be using them.

Procedure

1. Open the exported `questions.xml` file.
2. Find the review type you want to retire. For example, the following review type is the sample Auto Repair Service review type definition:

```
<ReviewType public-id="SPMReviewType:1">
  <ContactSubtype>AutoRepairShop</ContactSubtype>
  <Description>Sample Review Type for Auto Repair Shops</Description>
  <Name>Auto Repair Service</Name>
  <ShouldRetireFromImportXML>false</ShouldRetireFromImportXML>
</ReviewType>
```

3. Change the setting of `<ShouldRetireFromImportXML>` to `true`.

```
<ReviewType public-id="SPMReviewType:1">
  <ContactSubtype>AutoRepairShop</ContactSubtype>
  <Description>Sample Review Type for Auto Repair Shops</Description>
  <Name>Auto Repair Service</Name>
  <ShouldRetireFromImportXML>true</ShouldRetireFromImportXML>
</ReviewType>
```

4. Import the `questions.xml` file.

   See "Import questions and review types" on page 249.

Import questions and review types

Before you begin

To be able to import a service provider performance review type, you must have previously exported sample review data or manually created a `questions.xml` file.

Procedure

1. Log in to ClaimCenter as an administrator. For example, log in with username `su` and password `gw`.
2. Click the **Administration** tab, and then click **Utilities** > **Import Data** in the Sidebar.
3. On the **Import Administrative Data** screen, click **Browse** and navigate to the `questions.xml` file that you previously exported or created, and then click **Open**.
4. Click **Finish** to import the file.

Configuring categories for service provider performance reviews

You define categories for service provider performance reviews in an exported `questions.xml` file. After defining the categories, you import the file.

A `ReviewCategoryQuestionSet` entity connects a review type to a question set and applies a review category typecode to the question set. The typecode, defined in the `ReviewCategory` typelist, enables you to associate sets of questions with a category. ClaimCenter can then produce category scores for these sets of questions.

You can apply the same `ReviewCategory` typecode to more than one set of questions by defining multiple instances of `ReviewCategoryQuestionSet` entities. However, you cannot apply more than one `ReviewCategory` typecode to any single question set. If you want to use the same question set in another review, you must duplicate the question set and rename it. Then create a `ReviewCategoryQuestionSet` instance that specifies the review type, the review category, and the new question set.

A `ReviewCategoryQuestionSet` entity has the following elements:

`ReviewType`
The review type that uses this review category and question set. A `ReviewCategoryQuestionSet` can reference one review type. If you want to use a `ReviewCategory` with more than one `ReviewType`, define separate `ReviewCategoryQuestionSet` entities.

`QuestionSet`
The question set to which the review category applies. A `ReviewCategoryQuestionSet` can reference one question set.

`ReviewCategory`
A typecode defined in the `ReviewCategory` typelist. Assigning a review category to a question set enables the question set to be scored and the category score to be averaged.

`ElementOrder`

An integer that determines the order in which the categories appear in the `ReviewType`. Numbering starts at 0. You do not have to number categories sequentially, but doing so makes them easier to read and maintain.

The following code sample shows the XML definition of a `ReviewCategoryQuestionSet` entity used in the sample review type definition. This entity applies the `ReviewCategory` typecode `timeliness` to the question set `QuestionSetSPM:1` and adds it to the review type `SPMReviewType:1`. Additionally, the category has an element order of 1, making it the first category in the list.

```
<ReviewCategoryQuestionSet public-id="ReviewCatQSet:1">
  <ElementOrder>1</ElementOrder>
  <QuestionSet public-id="QuestionSetSPM:1" />
  <ReviewCategory>timeliness</ReviewCategory>
  <ReviewType public-id="SPMReviewType:1" />
</ReviewCategoryQuestionSet>
```

See also

- “Export questions and review types” on page 247
- “Import questions and review types” on page 249
- “Configuring review service type and review category typecodes” on page 250

Configuring review service type and review category typecodes

There are two typelists in ClaimCenter and ContactManager in which you define typecodes that affect service provider performance reviews, `ReviewServiceType` and `ReviewCategory`.

`ReviewCategory`

Typecodes assigned by the `ReviewCategoryQuestionSet` entity to a question set, enabling the system to calculate category scores for that question set.

`ReviewServiceType`

Typecodes used in a drop-down list on the review screen to enable you to choose the kind of service being provided by the service provider.

Note: You must define the same typecodes in both ClaimCenter and ContactManager.

Configure review category typecodes

About this task

`ReviewCategory` typecodes are categories that you assign to question sets to enable category scores to be calculated for reviews. You assign a typecode to a question set in a `ReviewCategoryQuestionSet` entity, as described in “Configuring categories for service provider performance reviews” on page 249. Before importing a review that uses a new typecode, add the typecode to ClaimCenter and ContactManager.

Procedure

1. Start Guidewire Studio™ for ClaimCenter.

   At a command prompt, navigate to the ClaimCenter installation folder and enter:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Typelist** and double-click `ReviewCategory.ttx` to open it in the Typelist editor.
3. Right-click an existing typecode and choose **Add new** > **typecode**.
4. Give the typecode a **code**, which you use for the `ReviewCategory` in a `ReviewCategoryQuestionSet`.

For example, there is a typecode for `timeliness`. The sample auto repair shop review associates this category with a question set containing questions that ask how quickly the auto repair shop was able to schedule and complete repairs.

5. Give the typecode a **name** and a **desc**, a description.

   ClaimCenter uses the **name** in screens that show category scores. The description is useful in deciding which categories to apply to question sets.

6. Start Guidewire Studio™ for ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter:

   ```
   gwb studio
   ```

7. Add the same typecode in ContactManager that you added for ClaimCenter, as described previously in step 2 through step 5.

8. Restart ClaimCenter and ContactManager.

   Note: Adding or changing a typecode is a data model change, and therefore requires a restart of the applications.

   a) If necessary, stop ClaimCenter.

      Open a command prompt in the ClaimCenter installation folder and then enter the following command:

      ```
      gwb stopServer
      ```

   b) Regenerate the ClaimCenter *Data Dictionary* to ensure that your changes are correctly formatted:

      ```
      gwb genDataDict
      ```

   c) Open a command prompt in the ContactManager installation folder and then enter the following command:

      ```
      gwb stopServer
      ```

   d) Regenerate the ContactManager *Data Dictionary* to ensure that your changes are correctly formatted:

      ```
      gwb genDataDict
      ```

   e) Start the ContactManager application:

      ```
      gwb runServer
      ```

   f) In the **Project** window, navigate to **configuration** > **gsrc** and then to `wsi.remote.gw.webservice.ab`.

   g) Double-click `ab1000.wsc`.

   h) Click **Fetch** to get the latest WSDL for the ContactManager review summary web service.

   i) Start the ClaimCenter application:

      ```
      gwb runServer
      ```

Overview of review service type typecodes

`ReviewServiceType` typecodes display in the **Service Type** drop-down list on the ClaimCenter review screen for new or incomplete reviews. This drop-down list enables the user to choose the kind of service performed by the service provider. If a service provider performs only one type of service, there might be no need to set up these typecodes for your review.

Part of defining one of these typecodes is specifying the contact subtype that the typecode applies to. You specify the contact subtype in the typecode's **Category**. The category is a typecode from the `Contact` typelist, and it matches the

`ContactSubtype` defined in the associated `ReviewType`. The application then uses this typecode category to determine which typecodes to show in the **Service Type** list for a specific review.

For example, when you start an auto repair shop review, the **Service Type** list shows six service types, **Body**, **Brakes**, **Other**, **Paint**, **Suspension**, and **Transmission and Engine**. All these service types are defined as typecodes in the `ReviewServiceType` typelist, and the **Category** for each of them is `Contact` typecode `AutoRepairShop`.

See also

- For a sample review type definition showing a `ContactSubtype`, see "Configure review service type typecodes" on page 252.

Configure review service type typecodes

About this task

Review service type typecodes are used in a drop-down list on the ClaimCenter service provider performance review screen to enable you to choose the kind of service being provided. See "Overview of review service type typecodes" on page 251.

Procedure

1. Start Guidewire Studio™ for ClaimCenter.

   At a command prompt, navigate to the ClaimCenter installation folder and enter:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Extensions** > **Typelist** and double-click `ReviewServiceType.ttx`.
3. Right-click an existing typecode and choose **Add new** > **typecode**.
4. Give the typecode a **code**, used internally to reference the typecode.
   For example, the service is wheel service, so the code can be `wheels`.
5. Give the typecode a **name** and a **desc**, a description. ClaimCenter shows the name in the drop-down list of service types on the review screen. The description is useful for documenting the typecode.
   For example, name the service `Wheels`. and give it the description `Alignment, balancing, tires`.
6. Right-click the new typecode and choose **Add new** > **Add Categories**.
7. In the **Add Categories** dialog box, choose the typecode you just added and click **Next**.
   For example, choose `wheels` and click **Next**.
8. For **Typelist**, choose `Contact`.
9. Choose a contact subtype that matches the `ContactSubtype` defined in the associated `ReviewType`.
   For example, choose `AutoRepairShop`.
10. Click **Finish**.
11. Start ContactManager studio.

    At a command prompt, navigate to the ContactManager installation folder and enter:

    ```
    gwb studio
    ```

12. Add the same typecode you added for ClaimCenter, as described previously in this topic

    Note: You do not need to add categories for these typecodes in ContactManager, so the steps for adding a typecode category do not apply for that product.

13. Generate the data dictionaries for ClaimCenter and ContactManager to ensure that the changes you made are correct.
14. Restart ContactManager and ClaimCenter, and then refresh the WSDL for `ABReviewSummary` in ClaimCenter studio. For details, see "step 8" in "Configure review category typecodes" on page 250

Configuring question sets, questions, choices, and filters

You can configure question sets, questions, and other features of service provider performance reviews.

See also

- For a generic description of question sets, questions, question choices, and question filters, see the ClaimCenter *Configuration Guide*.

Configuring question sets

A `QuestionSet` provides a way to group `Question` entities and is part of the mechanism for setting up review categories. A question set does not point to its member questions. Instead, each question in the set designates the question set it belongs to.

To enable a `QuestionSet` to work with service provider reviews, you must set its `QuestionSetType` to `spmreview`.

Note: This type is not the only one. ClaimCenter also uses other question set types in other areas.

Additionally, to improve readability, you can:

- Make the `Name` match the `ReviewCategory` in the corresponding `ReviewCategoryQuestionSet`.
- Set the `Priority` to be the same as the `ElementOrder` defined in the corresponding `ReviewCategoryQuestionSet`.

The following code samples are in the exported file `questions.xml`.

For example, the following question set is in the sample auto repair shop review:

```
<QuestionSet public-id="QuestionSetSPM:1">
  <Name>Timeliness</Name>
  <Priority>1</Priority>
  <QuestionSetType>spmreview</QuestionSetType>
  <ShouldRetireFromImportXML>false</ShouldRetireFromImportXML>
</QuestionSet>
```

The `ReviewCategoryQuestionSet` example `ReviewCatQSet:1` references this question set and sets the question set's review category to the typecode `timeliness`.

```
<ReviewCategoryQuestionSet public-id="ReviewCatQSet:1">
  <ElementOrder>1</ElementOrder>
  <QuestionSet public-id="QuestionSetSPM:1" />
  <ReviewCategory>timeliness</ReviewCategory>
  <ReviewType public-id="SPMReviewType:1" />
</ReviewCategoryQuestionSet>
```

See also

- “Export questions and review types” on page 247
- “Configuring categories for service provider performance reviews” on page 249

Configuring questions and question choices

Service provider performance reviews do not require special characteristics for defining questions and question choices. However, you typically use certain features of these entities for service provider reviews.

Service provider performance reviews are typically scored. Therefore, each question belongs to a question set and designates the question set it belongs to. Additionally, to enable scoring, each question typically has a question type of `choice`, which requires a set of `QuestionChoice` answers. Each `QuestionChoice` designates which question it answers.

Naming conventions can make it clear that the questions and question choices are for service provider reviews. For example, you might give the questions IDs like `QuestionSPM:1`, the choices IDs like `QuestionChoiceSPM:1`, and so on.

The question set `QuestionSetSPM:1` has a set of questions associated with it that ask about the timeliness of an auto repair. See “Configuring question sets” on page 253.

The following code, which is in the sample data exported to file `questions.xml`, defines the first question in this set and its answers, its `QuestionChoices`:

```
<Question public-id="QuestionSPM:1">
  <DefaultAnswer/>
  <Indent>0</Indent>
  <Priority>0</Priority>
  <QuestionFormat/>
  <QuestionSet public-id="QuestionSetSPM:1"/>
  <QuestionType>Choice</QuestionType>
  <Required>false</Required>
  <ShouldRetireFromImportXML>false</ShouldRetireFromImportXML>
  <Text>How quickly was the service provider able to schedule the repair?</Text>
</Question>
  <QuestionChoice public-id="QuestionChoiceSPM:1">
    <Code>win1day</Code>
    <Description>Within 1 day of initial contact</Description>
    <Name>Within 1 day of initial contact</Name>
    <Priority>1</Priority>
    <Question public-id="QuestionSPM:1"/>
    <Score>100</Score>
  </QuestionChoice>
  <QuestionChoice public-id="QuestionChoiceSPM:2">
    <Code>win3day</Code>
    <Description>Within 3 days of initial contact</Description>
    <Name>Within 3 days of initial contact</Name>
    <Priority>2</Priority>
    <Question public-id="QuestionSPM:1"/>
    <Score>50</Score>
  </QuestionChoice>
  <QuestionChoice public-id="QuestionChoiceSPM:3">
    <Code>more3days</Code>
    <Description>More than 3 days from initial contact</Description>
    <Name>More than 3 days from initial contact</Name>
    <Priority>3</Priority>
    <Question public-id="QuestionSPM:1"/>
    <Score>0</Score>
  </QuestionChoice>
```

This question, `How quickly was the service provider able to schedule the repair?`, has a `Priority` of `0`. This priority causes ClaimCenter to put the question at the top of the list when it shows the question set. Since its `QuestionType` is `Choice`, the question can be scored and has a series of `QuestionChoice` answers defined for it that have the scores `100`, `50`, and `0`. The answers display in order according to their `Priority` values.

See also

- "Configuring question sets" on page 253
- "Export questions and review types" on page 247

Configuring question filters

Question filters are a way to conditionally show questions based on the answer to another question.

In the sample auto repair shop service provider performance review, there are two filters that show additional questions if the answer to `QuestionSPM:11` is `yes`. That question asks if there were any requotes from the auto repair shop for an auto repair. If the answer is no, this question's score is 100 and no further questions are shown. If the answer is yes, this question's score is 0, and the user sees `QuestionSPM:12` and `QuestionSPM:13`, in that order. These questions ask how many requotes there were and the percentage of additional cost. The answers are choices, with better scores for a lower number of requotes and for a lower percentage of additional cost.

The following XML, from sample code in the exported file `questions.xml`, shows the two filters followed by the three questions and their answers:

```
<QuestionFilter public-id="SPMQFilter:1">
  <Answer>yes</Answer>
  <FilterQuestion public-id="QuestionSPM:11"/>
  <Question public-id="QuestionSPM:12"/>
</QuestionFilter>
<QuestionFilter public-id="SPMQFilter:2">
  <Answer>yes</Answer>
  <FilterQuestion public-id="QuestionSPM:11"/>
  <Question public-id="QuestionSPM:13"/>
</QuestionFilter>
<!-- ... -->
<Question public-id="QuestionSPM:11">
  <DefaultAnswer/>
```

```
    <Indent>0</Indent>
    <Priority>0</Priority>
    <QuestionFormat/>
    <QuestionSet public-id="QuestionSetSPM:5"/>
    <QuestionType>Choice</QuestionType>
    <Required>false</Required>
    <ShouldRetireFromImportXML>false</ShouldRetireFromImportXML>
    <Text>Were there any requotes?</Text>
  </Question>
    <QuestionChoice public-id="QuestionChoiceSPM:34">
      <Code>no</Code>
      <Description>No</Description>
      <Name>No</Name>
      <Priority>1</Priority>
      <Question public-id="QuestionSPM:11"/>
      <Score>100</Score>
    </QuestionChoice>
    <QuestionChoice public-id="QuestionChoiceSPM:35">
      <Code>yes</Code>
      <Description>Yes</Description>
      <Name>Yes</Name>
      <Priority>2</Priority>
      <Question public-id="QuestionSPM:11"/>
      <Score>0</Score>
    </QuestionChoice>
  <!-- ... -->
  <Question public-id="QuestionSPM:12">
    <DefaultAnswer/>
    <Indent>0</Indent>
    <Priority>1</Priority>
    <QuestionFormat/>
    <QuestionSet public-id="QuestionSetSPM:5"/>
    <QuestionType>Choice</QuestionType>
    <Required>false</Required>
    <ShouldRetireFromImportXML>false</ShouldRetireFromImportXML>
    <Text>How many requotes?</Text>
  </Question>
    <QuestionChoice public-id="QuestionChoiceSPM:36">
      <Code>one</Code>
      <Description>1</Description>
      <Name>1</Name>
      <Priority>1</Priority>
      <Question public-id="QuestionSPM:12"/>
      <Score>25</Score>
    </QuestionChoice>
    <QuestionChoice public-id="QuestionChoiceSPM:37">
      <Code>twothree</Code>
      <Description>2 to 3</Description>
      <Name>2 to 3</Name>
      <Priority>2</Priority>
      <Question public-id="QuestionSPM:12"/>
      <Score>10</Score>
    </QuestionChoice>
    <QuestionChoice public-id="QuestionChoiceSPM:38">
      <Code>morethan3</Code>
      <Description>More than 3</Description>
      <Name>More than 3</Name>
      <Priority>3</Priority>
      <Question public-id="QuestionSPM:12"/>
      <Score>0</Score>
    </QuestionChoice>
  <!-- ... -->
  <Question public-id="QuestionSPM:13">
    <DefaultAnswer/>
      <Indent>0</Indent>
      <Priority>2</Priority>
      <QuestionFormat/>
      <QuestionSet public-id="QuestionSetSPM:5"/>
      <QuestionType>Choice</QuestionType>
      <Required>false</Required>
      <ShouldRetireFromImportXML>false</ShouldRetireFromImportXML>
      <Text>% of difference between initial quote and final payment?</Text>
    </Question>
    <QuestionChoice public-id="QuestionChoiceSPM:39">
      <Code>atmost5</Code>
      <Description>Within 5% of original quote</Description>
      <Name>Within 5% of original quote</Name>
      <Priority>1</Priority>
      <Question public-id="QuestionSPM:13"/>
      <Score>25</Score>
    </QuestionChoice>
    <QuestionChoice public-id="QuestionChoiceSPM:40">
      <Code>fiveten</Code>
      <Description>Between 5% and 10%</Description>
      <Name>Between 5% and 10%</Name>
      <Priority>2</Priority>
      <Question public-id="QuestionSPM:13"/>
      <Score>10</Score>
    </QuestionChoice>
    <QuestionChoice public-id="QuestionChoiceSPM:41">
      <Code>tentwenty</Code>
      <Description>Between 10% and 20%</Description>
      <Name>Between 10% and 20%</Name>
      <Priority>3</Priority>
      <Question public-id="QuestionSPM:13"/>
      <Score>5</Score>
    </QuestionChoice>
    <QuestionChoice public-id="QuestionChoiceSPM:42">
      <Code>over20</Code>
      <Description>Greater than 20%</Description>
      <Name>Greater than 20%</Name>
      <Priority>4</Priority>
      <Question public-id="QuestionSPM:13"/>
      <Score>0</Score>
    </QuestionChoice>
```

See also

- "Configuring questions and question choices" on page 253
- "Export questions and review types" on page 247

Overview of service provider performance review scoring

After the completion of a service, typically the insured and claimant provide information to the claims adjuster or another user, who answers the questions in the service provider performance review. This process results in a scored review that evaluates the services provided by a vendor on a single claim.

Each review can have several scores. There is a total score and a score for each category of questions it contains. ClaimCenter scores the review and attaches it to the claim. A claim can have multiple reviews attached to it.

If you have set up the ClaimCenter scheduler to run the `ReviewSync` work queue, ClaimCenter queues the review's scores to send to ContactManager when the review is complete. Additionally, if you have set up the ContactManager scheduler to run the `ABContactScoring` work queue, ContactManager processes reviews it receives.

ContactManager creates vendor scores consisting of the average total score and average score for each category for all reviews from all claims for which the vendor provided services. These vendor scores are the main tool for managing categories of service providers.

The questions in the sample review have answers that range from zero to one hundred points. Scores, simple averages of the answered questions, are also in this range.

Service provider performance review scoring mechanisms

ClaimCenter provides two mechanism to score individual service provider performance reviews, one to score sets of questions—categories—and another to score the entire review. These scoring mechanisms are similar. Both are simple arithmetic averages, with each answered question given equal weight.

The overall score of an individual review is the arithmetic average of all answered questions. Unanswered questions do not affect this average. To make particular questions always affect the score, you can define those questions as mandatory and provide default answers for them. Similarly, each category score of a single review is the average of all answered questions in that category.

ClaimCenter scores an incomplete review each time it saves the review. After you complete and submit the review by clicking **Complete Selected**, ClaimCenter scores the review and sends it to ContactManager. ClaimCenter also attaches the completed review to the claim and never rescores it, and the review becomes uneditable.

Contact scoring for service provider performance reviews

Like service provider performance reviews themselves, ContactManager can give each vendor two kinds of scores, an overall score and a set of category scores.

Overall score

The arithmetic average of all overall scores from reviews pertaining to that contact. To be included, a review must be associated with the individual contact.

Category scores

The average of all scores from the same category in individual reviews that pertain to that contact.

Since different review types can reuse the same categories, the category score average can include category scores taken from all review types that include the category. For example, you might have review types for auto glass repair and for auto repair, both of which have the category of glass installation. A vendor's category scores are the averages of the same categories from all reviews.

Unlike review scores, which ClaimCenter attaches to the claim when the user makes the review complete, aggregated vendor scores update asynchronously through either a batch process or an API call.

Configuring scores for service provider performance reviews

By default, ClaimCenter scores are arithmetic means with each question given an equal weight. If you want to weight questions differently, you can either include unanswered questions in the scoring or calculate overall or category scores other than as arithmetic averages.

To configure scoring differently, you must write your own implementation of the `IContactReviewPlugin` plugin interface in ClaimCenter. The base configuration provides the plugin implementation `gw.plugin.spm.ab1000.ContactReviewPlugin`.

See also

- "Scheduling service provider performance reviews for processing" on page 241
- "Overview of service provider performance review scoring" on page 256
- "Service provider performance review plugins" on page 258

Service provider performance review permissions

In the base configuration, the following permissions are the primary ones associated with service provider performance reviews in ClaimCenter:

- `reviewviewlist` – Permission to view the list of reviews in the **Address Book** tab
- `reviewviewdetail` – Permission to view the details of reviews in ClaimCenter
- `reviewedit` – Permission to edit the **Review** screen that shows the scores for each claim
- `reviewcreate` – Permission to create a new review
- `reviewdeletecompleted` – Permission to delete a completed review that has been sent to ContactManager for calculation
- `reviewdeleteincomplete` – Permission to delete a review that has not been not completed

ContactManager has the following review summary permissions:

- `revsumviewdetail` – Permission to view the **Review Summary** screen and see category scores for each summarized review.
- `revsumviewlist` – Permission to view the list of review summaries and to see the **Reviews** card for a contact.

Adding Permissions for Contacts

If you need more granular control over who gets to view, edit, create, and delete contacts, do not use the simple view and edit permissions. For example, you can designate users to be contact managers that manage certain subtypes of contacts. For this purpose, you would want the system to enforce permissions at the contact subtype level. You can also enforce permissions at the contact tag level.

See also

- "Securing access to contact information" on page 85
- "Configuring ClaimCenter contact security" on page 102

Service provider performance review plugins

There are two plugin interfaces and two plugins that implement these interfaces that support service provider performance reviews.

See also

- For general information on plugins, see the *Integration Guide*.

ClaimCenter service provider performance review plugin

The ClaimCenter plugin class `gw.plugin.spm.ab1000.ContactReviewPlugin` implements the `IContactReviewPlugin` plugin interface.

The `IContactReviewPlugin` interface defines the following functionality:

- A hook for overriding the default scoring functionality.
- The ClaimCenter side of the integration with ContactManager.

The `ContactReviewPlugin` plugin implementation class implements the methods that follow.

`scoreReview`

This method scores the review for a given set of answers. If the review has category scores, the method creates or updates those scores on the `Review` object as necessary. It returns the overall score for the review.

`submitReview`

This method submits a summary of the completed review to ContactManager. It returns the `addressBookUID` for the submitted `ReviewSummary`.

Never call this method directly. If you want to extend the header information in `ReviewSummary`, you must edit this method to pass that information to the `ABVendorEvaluationAPIReviewSummary` web service. The method to call is `toReviewSummary(bundle : Bundle) : ReviewSummary`.

Note: For information on the `ABVendorEvaluationAPIReviewSummary` web service, open the class in Guidewire Studio™ for ContactManager. In the **Project** window, press `Ctrl+N` and enter `ABVendorEvaluationAPIReviewSummary`, and then double-click the class name in the search results.

`deleteReview`

This method deletes the summary corresponding to a completed review from ContactManager. Do not call this method manually.

`updateScores`

This method enables ClaimCenter to trigger a score update on an `ABContact` instance corresponding to the given local `Contact` without waiting for a run of the `ABContactScoring` work queue. The local `Contact` must be linked to ContactManager. You can use the method in extensions or rules, but you cannot call it directly.

See also

- "Register a vendor performance review plugin class in ClaimCenter" on page 258

Register a vendor performance review plugin class in ClaimCenter

If you implement your own service provider performance review plugin, you must register your new plugin in `IContactReviewPlugin.gwp`.

About this task

For information on the default service provider plugin implementation class, see "ClaimCenter service provider performance review plugin" on page 258.

Procedure

1. Start Guidewire Studio™ for ClaimCenter.

   At a command prompt, navigate to the ClaimCenter installation folder and enter:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Plugins** > **registry** and double-click `IContactReviewPlugin.gwp`.
3. In the Registry editor, you can see that the registered implementation is the Gosu class `gw.plugin.spm.ab1000.ContactReviewPlugin`.
4. In the Registry editor, click Remove Plugin —.
5. Click Add Plugin + and, in the drop-down menu, choose either **Add Gosu Plugin** or **Add Java Plugin**, depending on your implementation language.
6. In the **Gosu Class** or **Java Class** field, either search for your class or enter the class with the full package name.
7. Add the password and username parameters you expect the plugin to use to communicate with ContactManager. The default settings are:

Name	Value
`password`	`gw`
`username`	`ClientAppCC`

What to do next

See also "Configuring core application authentication with ContactManager" on page 41.

ContactManager contact scoring plugin for service providers

The ContactManager plugin implementation class `gw.plugin.spm.impl.ABContactScoringPlugin` implements the interface `IABContactScoringPlugin`. The plugin implementation class implements the methods that follow.

`scoreABContact`

This method calculates the average score and average category scores for a contact. It returns the overall average review score. ClaimCenter scripting calls it through `ContactReviewPlugin` and `gw.webservice.ab.ab1000.abvendorevaluationapi.ABVendorEvaluationAPIReviewSummary`. The `ABContactScoring` work queue worker also calls this method.

You can implement a different version of this method to change how scoring works. For example, you might want to weight scores in some manner.

`findContactsToScore`

This method finds the set of `ABContact` objects to be scored for the `ABContactScoring` work queue writer. It returns the iterator key for the contacts it finds.

This method is called by the work queue writer. You can reimplement this method to change how scoring works. The method is useful if, in your scoring system, some contacts need to be rescored without having received new reviews.

For example, you might want to weight scores by recency of reviews or to cancel scores that are based on old reviews.

See also

- "Register a contact scoring plugin class in ContactManager" on page 260

Register a contact scoring plugin class in ContactManager

If you implement your own contact scoring plugin for service provider performance reviews, you must register your new plugin in `IABContactScoringPlugin.gwp`.

About this task

For information on the default contact scoring plugin implementation, see “ContactManager contact scoring plugin for service providers” on page 259.

Procedure

1. Start Guidewire Studio™ for ClaimCenter.

   At a command prompt, navigate to the ContactManager installation folder and enter:

   ```
   gwb studio
   ```

2. In the **Project** window, navigate to **configuration** > **config** > **Plugins** > **registry** and double-click `IABContactScoringPlugin.gwp`.
3. In the Registry editor, you can see that the default implementation is `gw.plugin.spm.impl.ABContactScoringPlugin`.
4. In the Registry editor, click Remove Plugin —.
5. Click Add Plugin + and, in the drop-down menu, choose either **Add Gosu Plugin** or **Add Java Plugin**, depending on your implementation language.
6. In the **Gosu Class** or **Java Class** field, enter the full package name of the new plugin implementation class.

chapter 14

Working directly in ContactManager

You can work directly with contact data in ContactManager by using screens that manage contact data and ContactManager functionality.

See also

- For information on directory structure and important configuration directories like `module` and `configuration,` see “ContactManager configuration files” in the *Configuration Guide*.
- For information on defining the server environment, see “Understanding the ContactManager server environment” in the *Administration Guide*.
- For information on setting up logging, see “Overview of application logging” in the *Administration Guide*.
- For a list of command-prompt administrative tools, such as the `maintenance_tools` command, see “Command prompt tools” in the *Administration Guide*.

Log in to ContactManager

Before you begin

Logging in to ContactManager requires the following:

The URL (web address) for connecting to ContactManager

- In the base configuration, the URL is `http://localhost:8280/ab`
- You can set up a **Favorite** link to the URL or a create a shortcut on your computer desktop that starts your web browser with that URL.

  Note: Guidewire supports Firefox, Google Chrome, and Microsoft Internet Explorer.

A user name and password

You must have one or more roles assigned to your user name by a system administrator. Roles determine the screens you can access and what you can do in ContactManager.

See also

- “Installing ContactManager” on page 17.

Procedure

1. Launch ContactManager by running a web browser and using the appropriate web address, such as:

```
http://localhost:8280/ab
```

2. Enter your **User Name** and **Password** on the login screen.

Results

If your login is successful, ContactManager shows your startup view, or landing page. In the default configuration, ContactManager initially opens the **Search** screen on the **Contacts** tab.

Note:

Because ContactManager generates screens dynamically:

- You cannot create **Favorites** to screens other than the login screen.
- The **Back** button of the browser is not supported.

Change your password in ContactManager

Before you begin

Log in as described at “Log in to ContactManager” on page 261.

About this task

After you log in to ContactManager, you can change your password.

Procedure

1. On the Options menu ⚙, click **Preferences**.
   The **Preferences** worksheet opens below the main work area.
2. Enter your old password.
3. Enter the new password.
4. Enter the new password again in the **Confirm New Password** field.
5. Click **Update**.

Change your user preferences in ContactManager

Procedure

1. Log in to ContactManager.
2. On the Options menu ⚙, click **Preferences**.
   The **Preferences** worksheet opens below the main work area.
3. Change your preferences, such as your password, regional formats, default country, and default phone region.

What to do next

See “Preferences worksheet” on page 262.

Preferences worksheet

In the **Preferences** worksheet, you can change your password, set your regional formats, your default country, and your default phone region. You set the last three if you want your personal settings to be different from the ones set for all users of ContactManager.

- **Password** – Reset your password. See “Change your password in ContactManager” on page 262.
- **Regional Formats** – Set the regional formats that ContactManager uses to enter and display dates, times, numbers, monetary amounts, and names.
- **Default Country** – Determine the settings for names and addresses.
- **Default Phone Region** – Determine how phone number entries are handled, especially the country code setting.

Selecting international settings in ContactManager

In ContactManager, each user can set the following:

- The language that ContactManager uses to display labels and drop-down menu choices
- The regional formats that ContactManager uses to enter and display dates, times, numbers, monetary amounts, and names.

You set your personal preferences for display language and for regional formats by using the Options menu ⚙ at the top, right-hand side of the ContactManager screen. On that menu, click **International**, and then select one of the following:

- **Language**
- **Regional Formats**

See also

- “International settings in ContactManager” on page 266

Changing user interface settings in ContactManager

You can change user interface settings to control the behavior of certain functions in the ContactManager user interface.

On the top tab bar, in the **Options** ⚙ menu, click **Settings** to open the **Settings** screen.

Settings include font size, screen spacing, themes, date entries, general browser behavior, changing the screen for debugging, and some currency features. For complete descriptions, see the following topics that describe all ContactManager settings.

See also

- “Changing interface settings” in the *Application Guide*
- “Change the visual theme” in the *Application Guide*
- “Highlight changed values” in the *Application Guide*

Changing the screen layout

You can adjust some aspects of the screen layout to fit your preferences. You can adjust list views, change the sidebar width, and manage layout preferences. The instructions for doing so in ContactManager are the same as those for the core applications.

See also

- For more information on changing the screen layout, see the *Application Guide*

Data entry support for ContactManager input fields

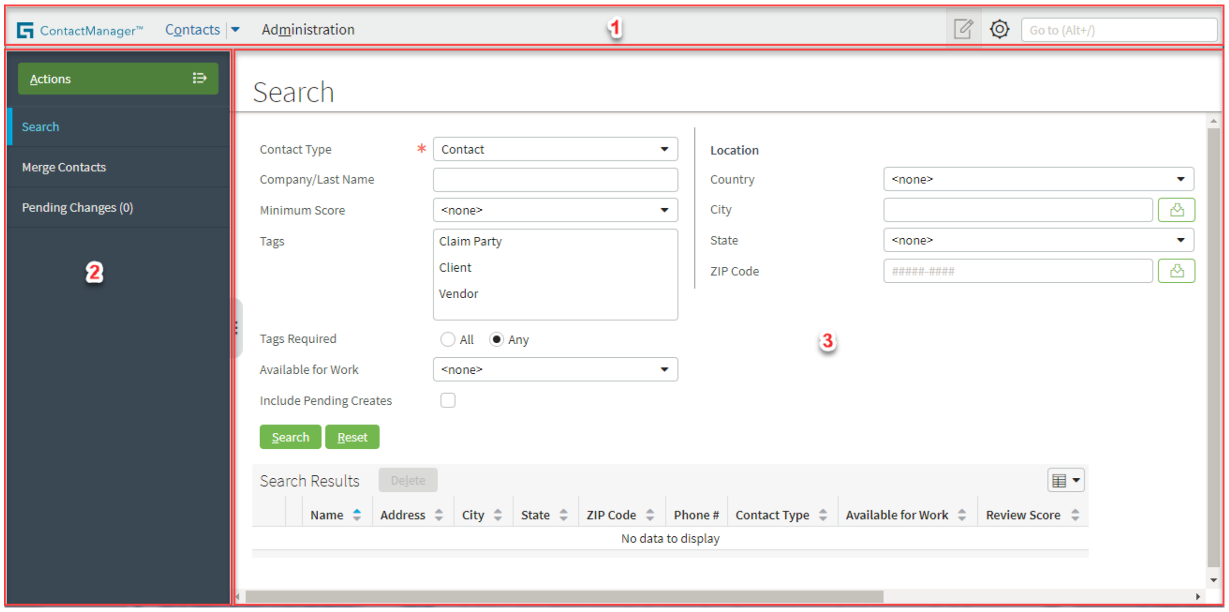
As you type in data for some types of input fields, ContactManager formats the data appropriately for the field. The following topics describe this input field support for ContactManager, which is the same as that for ContactManager.

See also

- "Using the currency macro in currency fields" in the *Application Guide*
- "As-you-type formatting support for input fields" in the *Application Guide*
- "Highlight changed values" in the *Application Guide*

ContactManager user interface

Like the Guidewire core applications, ContactManager has tabs at the top of the screen and an **Actions** button and a menu on the left for navigation.



The ContactManager main user interface contains the following areas:

Area	Description
1	The Tab Bar contains: • Tabs, which group major areas of work for you, providing main screens, actions on the **Actions** menu, and Sidebar menu links. The tabs you see depend on your user permissions. All users can see the **Contacts** tab. • Unsaved Work button. This button is active if you have work you have not saved. You can click the button and go to any screen in which you have not saved your changes. This feature is useful if you must navigate away from a screen in which you are making changes, but you intend to return to it later. After you complete and save your work, ContactManager removes that item from the Unsaved Work drop-down menu. • The Options menu. The selections available on this menu depend on where you are in the ContactManager user interface. Typical menu choices are **International**, **Help**, **About**, **Preferences**, **Clear Layout Preferences**, and **Log Out** *Username*. • QuickJump box. The text box that displays **Go to (Alt+/)**. In the base configuration of ContactManager, this box enables you to run batch processes with the command `RunBatchProcess` *`batchProcess`*. You can configure it.
2	The Sidebar contains menu links and the **Actions** menu. Use the Sidebar menu links to navigate to screens where you can do your work. The items in the Sidebar are contextual and change depending on tab and menu selections.
3	The Screen Area shows most of your business information and is where you interact with that information.

See also

- For a description of the **International** menu and its submenu selections, **Regional Settings** and **Language**, see “International settings in ContactManager” on page 266.
- For information on using and configuring **QuickJump**, see the *Application Guide*.

Contacts tab

The **Contacts** tab enables you to:

- Click **Actions** to create a new person, company, or place to store in ContactManager, if you have permissions to do so.
- Click **Search** to find contacts by using search criteria, such as contact subtype and location.
- Edit and delete any contact entities for which you have permissions.
- Click **Merge Contacts** to merge duplicate contacts that have been discovered by Duplicate Contacts Finder batch processing, if you have permissions to do so.
- Click **Pending Changes** to review changes to contacts that were made by core application users who did not have permissions to make the changes. You need the correct permissions to do so.

See also

- “Detecting and merging duplicate contacts” on page 271
- “Review pending changes to contacts” on page 278

Administration tab

On the ContactManager **Administration** tab, you can manage roles, permissions, login name and password, and group membership for your ContactManager users. Additionally, you can manage message queues, export and import data, and work with script parameters.

Note: To see this tab, you must be logged in to ContactManager as an administrator.

- The **Actions** button on the left enables you to create new ContactManager users and groups.
- Also on the left in the Sidebar is a hierarchical list of all your groups and users.

Additionally, you can choose the following items from the Sidebar:

- **Users and Security** – Groups the following administrative tasks:
    - **Users** – Search for users by role, location, username, and first and last name.
    - **Groups** – Search for groups by name and type.
    - **Roles** – Add and delete roles, add permissions to and remove permissions from roles, and assign roles to ContactManager users.
    - **Regions** – Add, delete, and edit the current regions for ContactManager contacts. *Regions* are geographical areas you can use to define groups’ areas of responsibility. A region can contain one or more states, ZIP codes, counties, or other address elements, such as Canadian provinces. You can assign users and groups to cover one or more regions, and then define business rules to provide location-based assignment.
- **Monitoring** – Groups the following administrative tasks:
    - **Message Queues** – Suspend and resume event messages sent by ContactManager and restart the messaging engine. One use of the **Message Queues** screen is to restart message queues that have been suspended. For example, the message queue for a core application might have been suspended because the core application was not running when ContactManager tried to send contact data.
- **Utilities** – Groups the following administrative tasks:
    - **Import Data** – Import certain types of data through the ContactManager interface.

- **Export Data** – Export certain types of data through the ContactManager interface.
- **Script Parameters** – View existing ContactManager script parameters. To create new script parameters, you must use ContactManager Studio.
- **Vendor Services Onboarding** – Add vendor services for contacts.
- **Data Change** – Enables you to push data changes to the production server. Use this feature sparingly and only to update mission-critical data on running production systems.
- **Runtime Properties** - Provides the ability to add or change application properties in real-time without restarting the application server.

See also

- For information on regions, see the *Application Guide*.
- For information on roles and security, see "Securing access to contact information" on page 85.
- "ContactManager messaging events" on page 308
- "Troubleshooting the ClaimCenter connection with ContactManager" on page 27
- "Troubleshooting the PolicyCenter connection with ContactManager" on page 34
- "Troubleshooting the BillingCenter connection with ContactManager" on page 41
- "Vendor services onboarding" on page 281
- For information on data change, see the *Administration Guide*.
- For information on script parameters, see the *Configuration Guide*
- For information on runtime properties, see the *Administration Guide*

Internal Tools and Server Tools tabs

You can log in to ContactManager as the super user. You can then press `Alt+Shift+T`, click the **Internal Tools** tab, and load sample data.

Alternatively, you can click the **Server Tools** tab and perform administrative tasks, such as starting batch jobs manually.

See also

- "Load sample data for ContactManager" on page 18
- "Start work queues and batch processes" on page 279
- "Working directly in ContactManager" on page 261
- For more information on the **Internal Tools** and **Server Tools** tabs, see the *Administration Guide*

International settings in ContactManager

You can set the following international settings:

- The language in which ContactManager displays labels and drop-down menu choices
- The regional formats that ContactManager uses to enter and display dates, times, numbers, monetary amounts, and names

You set your personal preferences for display language and for regional formats by using the Options menu ⚙ at the top, right-hand side of the ContactManager screen. On that menu, click **International**, and then select one of the following:

- **Language**
- **Regional Formats**

To set international settings in the application, you must configure ContactManager with more than one region. Your configuration of regions and languages determines what you see on this menu, as follows:

- ContactManager hides the **Language** submenu if only one language is installed.
- ContactManager hides the **Regional Formats** submenu if only one region is configured.
- ContactManager hides the **International** menu option entirely if a single language is installed and ContactManager is configured for a single locale.

ContactManager indicates the current selections for **Language** and **Regional Formats** by placing a check mark to the left of the selected options and displaying them in gray.

Options for language

In the base configuration, Guidewire has a single display language, English. To view another language in ContactManager, you must install a language pack and configure ContactManager for that language. If your installation has more than one language, you can select among them from the **Language** submenu. The `LanguageType` typelist defines the set of language choices that display on the menu.

If you do not select a display language from the **Language** submenu, ContactManager uses the primary display language. The configuration parameter `DefaultApplicationLanguage` specifies this language. In the base configuration, the primary display language is U.S. English.

Options for regional formats

If your installation contains more than one configured region, you can select a regional format for that region from the **Regional Formats** submenu. At the time you configure a region, you define regional formats for it.

Regional formats specify the visual layout of the following kinds of data:

- Date
- Time
- Number
- Monetary amounts
- Names of people and companies

The `LocaleType` typelist defines the names of regional formats that you can select from the **Regional Formats** submenu. The base configuration defines the following locale types:

• Australia (English)	• Germany (German)
• Canada (English)	• Great Britain (English)
• Canada (French)	• Japan (Japanese)
• France (French)	• United States (English)

If you do not select a regional format from the **Regional Formats** menu, ContactManager uses the regional formats of the default region. The configuration parameter `DefaultApplicationLocale` specifies the default region. In the base configuration, the default region is `en_US`, United States (English). If you select your preference for region from the **Regional Formats** menu, you can later use the default region again only by selecting it from the **Regional Formats** menu.

See also

- For an introduction to globalization and more information on working with regional formats and languages, see the *Globalization Guide*

Importing and exporting administrative data

While much administrative data is entered directly into ContactManager, there are times when it is useful to transfer this information in bulk.

The **Export Data** and **Import Data** screens available on the **Administration** tab provide a convenient way of moving data to and from XML files. You can export and import administrative data, role definitions, and vendor service trees.

Note: In ClaimCenter, you can also export and import vendor service review types and questions and vendor service details. Adding or changing vendor services in ContactManager requires that you use administrative import. Adding or changing vendor services, vendor service details, and vendor service review types and questions in ClaimCenter also requires that you use administrative import.

Additionally, you can export the security dictionary as either HTML or XML.

You can also import or export other types of data, in either XML or CSV format, by using an API. Also, you can use a command on the command line to import files in either format, but not to export them.

Method	Import	Export	File Formats
user interface	`admin.xml`, `vendorservicetree.xml`, `questions.xml`	`admin.xml`, `roles.xml`, `vendorservicetree.xml`, `questions.xml`, `securitydictionary`*`number`*`.zip`	XML, XML or HTML in compressed ZIP
command line	any	no	XML, CSV
API	any	any	XML, CSV

See also

- "Administration tab" on page 265
- "Importing and exporting vendor service data" on page 164
- "Export questions and review types" on page 247
- For more information on how to import a service, see the *Configuration Guide*

Import administrative data and vendor service trees

About this task

You can import administrative data in the **Administration** tab. For more information, see the *Administration Guide*.

Procedure

1. Click the **Administration** tab and then navigate to **Utilities** > **Import Data** to select a file to import.
2. You can click the **Browse** button to find the file.
   For example, you have created a file of administrative data called `admin.xml`. The file must be either in XML or compressed XML format, with an XSD compatible with the XML files you can import. If you want to import administrative data, you can import any subset of this type of data, such as users or regions.

   As another example, you have created a file of vendor service data called `vendorservicetree.xml`. You can use this file to instantiate vendor services in ContactManager.
3. Click **Next** and follow the prompts to resolve differences between the data in the imported file and data already in the database.

   Data not yet in the database is imported without question. If the imported data differs from what is already in the database, these prompts enable you either to accept the imported data or to keep what is in the database.
4. Click **Finish** to complete the import.

Exporting data in the administration tab

To export administrative data, click the **Administration** tab and then navigate to **Utilities** > **Export Data** to see the following categories. Each category has one or more types of data you can export. Each file contains all the data of a certain type in your installation. These export categories are:

Export Administrative Data

- **Admin** – Exports all administrative data to `admin.xml`, including data of the following types:

  - `Group`, `GroupRegion`, `GroupRuleSet`, `GroupUser`
  - `Questions`
  - `Region`
  - `Role`, `Privileges`, `RolePrivilege`, `Permission`
  - `User`, including `AttributeUser`, `UserRole`, `UserSettings`
  - `UserPreference`
- **Roles** – Exports all data that maps system permissions to roles to the file `roles.xml`. If you choose **Admin** as the export type, the same role data is exported with the other administrative data. The file has data of the following types:
  - `Role`
  - `Privileges`
  - `RolePrivilege`
  - `Permission`
- **Vendor Service Tree** – Exports the entire vendor service hierarchy to the XML file `vendorservicetree.xml`.

Export security dictionary

You can export the security dictionary as a compressed XML or HTML file.

Importing and exporting either with Gosu or from the command prompt

You might want to import or export other types of data or use files in formats other than XML. For example, if you receive new information from an external system, you might want to import this new data into ContactManager in a single step. Gosu classes and command-prompt commands are your two alternatives to using the administrative user interface. Gosu classes enable you both to import and to export, but the command-prompt commands support only importing, not exporting.

Importing from the command prompt

There are command-prompt commands for importing, but not exporting, XML and CSV files containing any kind of data, not just the types of data supported by the **Administration** tab. See the *Administration Guide*.

Importing and exporting with Gosu classes

You can import and export administrative data by using Gosu classes. See the *Integration Guide*.

Using inbound files integration

The base configuration of ContactManager includes a framework for configuring multiple integrations with external systems by processing file-based data. ContactManager provides the framework with a general processing mechanism and the `InboundFileHandler` interface which describes how to process data in the files. You must provide configuration details and a class implementing the `InboundFileHandler` interface.

In the **Administration** tab, the **Utilities** > **Inbound Files** menu link enables you to configure the feature.

> **Note:** The instructions for configuring inbound files in ContactManager and the framework itself is the same as those for the core applications.

See also

- *Integration Guide*

Using outbound files integration

The base configuration of ContactManager includes a framework that supports creating files for external systems. The files are created from records in a database. ContactManager provides the framework with a general processing mechanism and the `OutboundFileHandler` interface which describes how to process data in the records.

You must provide configuration details and a class implementing the `OutboundFileHandler` interface.

In the **Administration** tab, the **Utilities** > **Outbound Files** menu link enables you to configure the feature.

Note: The instructions for configuring outbound files in ContactManager and the framework itself is the same as those for the core applications.

See also

- *Integration Guide*

Managing contact data

You can use Guidewire core application screens to manage contacts stored in ContactManager.

ClaimCenter

In the ClaimCenter **Address Book** tab, you can search for and view existing contacts. You can change ContactManager data in ClaimCenter by using screens for managing or adding claim contacts, such as the **New Claim** wizard or a claim's **Parties Involved** screen.

PolicyCenter

In the PolicyCenter **Contact** tab, you can create new contacts, search for existing contacts, select a recently viewed contact, and change contact information. You can also create an account for the contact. PolicyCenter additionally enables you to work with contacts in its **Account** and **Policy** screens, where you can add, remove, and update contacts in various roles.

BillingCenter

In the BillingCenter **Account** and **Policy** tabs, you can click **Contacts** to open the **Contacts** screen. On this screen you can add new contacts, search for existing contacts, select a recently viewed contact, and change contact information. Additionally, you can search for contacts by clicking the **Search** tab and choosing **Contacts** from the drop-down list.

To work with ContactManager from a Guidewire core application, you must install both ContactManager and the Guidewire core application and integrate them.

You can also manage contacts stored in ContactManager by logging in to ContactManager as a user with the appropriate role. For example, you must log in to ContactManager to merge contacts or approve pending contact changes. Additionally, if you want to add contacts that are not on a claim, such as vendor contacts, you must do so in ContactManager.

You log in as a user with the Contact Manager role, which has permissions supporting viewing, searching for, merging, adding, editing, and deleting contacts and reviewing pending contacts. These permissions include:

- **View merge** permission, with code `abviewmerge`, which enables you to see **Merge Contacts** screens
- **View pending** permission, with code `abviewpending`, which enables you to see **Pending Contacts** screens

Other reasons to log in to ContactManager are to manage its users, to manage vendor services, or to manage the server. For example, you can assign the User Admin role to a user. That user can then log in to ContactManager and manage its users.

See also

- "Installing ContactManager" on page 17
- "Integrating ContactManager with Guidewire core applications" on page 21.

- “Vendor services onboarding” on page 281
- “Working with vendor documents” on page 179

Detecting and merging duplicate contacts

Because creating contacts can result in duplicate contact records, there are features in ContactManager to detect duplicate contacts and enable merging them. A user with appropriate privileges in ContactManager can run Duplicate Contacts Finder batch processing to detect duplicate contacts, and then merge each duplicate contact in the **Merge Contacts** screen.

IMPORTANT: The first time you run Duplicate Contacts Finder batch processing, it can perform a large number of comparisons requiring considerable application resources. You can limit the number of contacts it processes by setting a processing time and date before which contacts are ignored. For more information, see “Configuring Duplicate Contacts Finder batch processing” on page 271. Additionally, Guidewire recommends that you set the work queue to run initially at a time when general access to your server is restricted. You might also need to allocate time for a ContactManager user to resolve what is likely to be a large number of duplicate contacts.

Configuring Duplicate Contacts Finder batch processing

Duplicate Contacts Finder batch processing compares a set of contacts that have recently changed against the rest of the contacts in the ContactManager database. *Recently* is defined by either of the following:

- Contact changes that occurred after the last time the batch process ran
- Contact changes that occurred after the date and time set in the configuration parameter `DuplicateContactsEarliestModificationDate`

For example:

1. Duplicate Contacts Finder batch processing ran today at 1:00 am. There were 100 contacts in the database at that time.
2. Duplicate Contacts Finder found 5 possible duplicate contacts.
3. The contact administrator checked today for duplicate contacts and merged 5 of them, leaving 95 contacts in the database.
4. After Duplicate Contacts Finder last ran at 1:00 am today, 10 new contacts were created.
5. Duplicate Contacts Finder runs at 1:00 am tomorrow. It compares each of those 10 new contacts against 104 contacts—the other 9 new contacts and the 95 contacts that were already in the database.

Duplicate Contacts Finder checks the value of `DuplicateContactsEarliestModificationDate`, which you can set in `config.xml`. Duplicate Contacts Finder ignores any contacts created or modified before the date and time in this configuration parameter. You can use this setting to avoid processing every contact in the database the first time you run the batch process.

See also

“Set the DuplicateContactsEarliestModificationDate configuration parameter” on page 271

Set the `DuplicateContactsEarliestModificationDate` configuration parameter

Before you begin

Set this configuration parameter for Duplicate Contacts Finder. See “Configuring Duplicate Contacts Finder batch processing” on page 271.

About this task

For example, you imported a large number of contacts into the database on February 18, 2018, and you set `DuplicateContactsEarliestModificationDate` to `02/19/2018 12:00 AM`. Duplicate Contacts Finder does not process any contacts from your February 18 import unless you modified them after 12:00 am on February 19, 2018. Duplicate Contacts Finder does process contacts modified after that date and time.

The date and time for this entry must have the following format:

```
mm/dd/yyyy hh:mm am
```

Procedure

1. Start Guidewire Studio™ for ContactManager.

   At a command prompt, navigate to the ContactManager installation folder and enter the following command:

   ```
   gwb studio
   ```

2. Navigate in the **Project** window to **configuration** > **config** and double-click `config.xml` to open this file in the editor.
3. Press `Ctrl+F` and enter `DuplicateContactsEarliestModificationDate` to find this parameter setting in the file.
4. Enter a date and time value.
   `DuplicateContactsEarliestModificationDate="02/19/2018 12:00 AM"`
5. Save your changes.
6. If ContactManager is running, stop ContactManager and restart it to pick up the configuration change.
7. Run Duplicate Contacts Finder, or set it up to run automatically.

What to do next

See also

- “Running the Duplicate Contacts Finder work queue manually” on page 272
- “Setting Duplicate Contacts Finder to run automatically” on page 273

Running the Duplicate Contacts Finder work queue manually

You can run Duplicate Contacts Finder batch processing manually in the **Batch Process Info** screen or from the command line. For example, you might run it to support testing of duplicate contacts.

Run Duplicate Contacts Finder from the batch process info screen

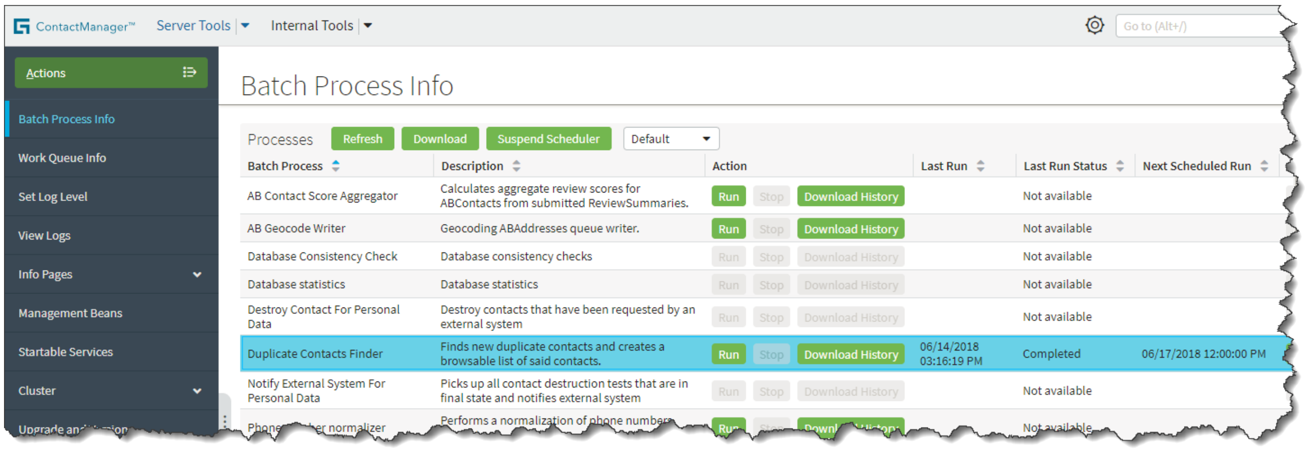
About this task

You can run Duplicate Contacts Finder in the ContactManager user interface.

Procedure

1. Log in as a user with the Tools View role. This role has the permission `toolsBatchProcessview`, which enables you to work with work queues and batch processes. For example, log in with username `su` and password `gw`.
2. Press `Alt+Shift+T` to open the screen showing the **Server Tools** tab and the **Internal Tools** tab.
3. Click the **Server Tools** tab.
4. From the Sidebar on the left under **Actions**, choose **Batch Process Info**.
5. In the **Batch Process Info** screen, scroll down to **Duplicate Contacts Finder** and click the **Run** button in that row.

   When the process completes, the **Last Run Status** changes to Completed and the **Last Run** column updates to show the run date and time.



Batch Process Info

Batch Process	Description	Action	Last Run	Last Run Status	Next Scheduled Run
AB Contact Score Aggregator	Calculates aggregate review scores for ABContacts from submitted ReviewSummaries.	Run Stop Download History		Not available	
AB Geocode Writer	Geocoding ABAddresses queue writer.	Run Stop Download History		Not available	
Database Consistency Check	Database consistency checks	Run Stop Download History		Not available	
Database statistics	Database statistics	Run Stop Download History		Not available	
Destroy Contact For Personal Data	Destroy contacts that have been requested by an external system	Run Stop Download History		Not available	
Duplicate Contacts Finder	Finds new duplicate contacts and creates a browsable list of said contacts.	Run Stop Download History	06/14/2018 03:16:19 PM	Completed	06/17/2018 12:00:00 PM
Notify External System For Personal Data	Picks up all contact destruction tests that are in final state and notifies external system	Run Stop Download History		Not available	
Phone er normalizer	Performs a normalization of phone numbers	Run Stop Downl		Not available	

Run Duplicate Contacts Finder from a command prompt

About this task

You can run Duplicate Contacts Finder from a command prompt by using the `maintenance_tools` command.

Procedure

1. At a command prompt, navigate to `ContactManager/admin/bin`.
2. Enter the following command:

   ```
   maintenance_tools -startprocess duplicatecontacts
   ```

What to do next

See also

- For information on the `maintenance_tools` command, see the *Administration Guide*.

Setting Duplicate Contacts Finder to run automatically

In the base configuration, **Duplicate Contacts Finder** batch processing is set to run once a week on Sunday at 12:00 pm. You can:

- **Change this setting to run the work queue on a schedule** – See “Scheduling work queue writers” on page 273.
- **Set up the work queue itself** – See “Changing work queue settings” on page 274.
- **Limit the number of search results** – See “Changing match results and search scope settings” on page 274.
- **Set the scope of searches** – See “Changing match results and search scope settings” on page 274.

See also

- For information on administering batch processing and on work queues, see the *Administration Guide*

Scheduling work queue writers

In Guidewire Studio™ for ContactManager, navigate in the **Project** window to **configuration** > **config** > **scheduler** and double-click `scheduler-config.xml` to open the file in the editor. In this XML file, you can change the settings for the `ProcessSchedule` named `DuplicateContacts`. The default entry sets the process to run at noon every Sunday.

```
<ProcessSchedule process="DuplicateContacts">
  <CronSchedule dayofmonth="?" dayofweek="SUN" hours="12"/>
</ProcessSchedule>
```

For example, the following entry would cause the process to run at midnight every night:

```
<ProcessSchedule process="DuplicateContacts">
  <CronSchedule hours="0"/>
</ProcessSchedule>
```

See also

- For complete information on the scheduling settings for work queue writers, see the *Administration Guide*.

Changing work queue settings

In Guidewire Studio™ for ContactManager, navigate in the **Projects** window to **configuration** > **config** > **workqueue** and double-click `work-queue.xml` to open it in the editor. In this XML file, you can change the settings for the `workQueueClass` named `com.guidewire.ab.domain.contact.DuplicateContactsFinderWorkQueue`:

```
<work-queue
    progressinterval="600000"
    workQueueClass="com.guidewire.ab.domain.contact.DuplicateContactsFinderWorkQueue">
  <worker batchsize="100" instances="4"/>
</work-queue>
```

The settings are described in the comments at the beginning of the `work-queue.xml` file. The base configuration settings for `com.guidewire.ab.domain.contact.DuplicateContactsFinderWorkQueue` are:

`minpollinterval="0"`
: The worker does not sleep after completing a work item.

`maxpollinterval="60000"`
: The worker wakes up and polls for work every 60,000 milliseconds, or one minute.

`progressinterval="600000"`
: The amount of time the system gives the worker to do the work before assuming that it ran into a problem and reassigning the work to another worker. By default this time is 600,000 milliseconds, or 10 minutes.

`batchsize="100"`
: Each worker takes 100 new contact names at a time and compares each one to the contacts in the database.

`instances="1"`
: The number of workers for the process.

See also

- For more information on work queues and work queue settins, see the *Administration Guide*

Changing match results and search scope settings

You can set the maximum number of matches that can be found for a contact and how wide the search is. In Guidewire Studio™ for ContactManager, navigate in the **Project** window to **configuration** > **config** and open `config.xml`. Change the following parameters under the heading `DuplicateContactsFinderWorkQueue`:

- `MaxDuplicateContactsFinderWorkQueueResults` – By default, each `DuplicateContactsFinderWorkQueue` worker finds up to one thousand potential duplicates for a single contact. This parameter is intended to catch searches that have too loose a set of search fields and, therefore, match too many contacts in the database. If the worker finds more than the maximum allowed number of duplicates, it does not create the list of duplicates in its results list. Instead, it logs an exception that indicates the `LinkID` of the new contact. In addition, the new contact has an exception entry in the results list. The default setting is:

  ```
  <param
    name="MaxDuplicateContactsFinderWorkQueueResults"
    value="1000"/>
  ```

- DuplicateContactsWideSearch – By default, this parameter is true, causing the process to perform a wide search using the match settings for ABPerson, ABCompany, and ABPlace. The default setting is:

  ```
  <param
    name="DuplicateContactsWideSearch" value="true"/>
  ```

  Note: This parameter applies only to duplicate finder batch processing. It does not set general searching behavior in the plugin FindDuplicatesPlugin.

When you specify wide search for duplicate finder processing, it can return matches for a larger number of subtypes than matching that uses DEFAULT_MAP (a setting of false for DuplicateContactsWideSearch). WIDE_MAP uses the contact's supertypes, ABPerson, ABCompany, and ABPlace. This type of search aids in finding a contact that was added erroneously to the database as more than one subtype.

In the base configuration, DEFAULT_MAP additionally returns results for ABPersonVendor and ABCompanyVendor, which are subtypes of ABPerson and ABCompany. For example, if the subtype of the contact being compared against the database is ABDoctor, a DEFAULT_MAP search returns only matches for the ABPersonVendor subtype. This subtype is the closest supertype specified in the map. The wide search in the base configuration returns matches for all ABPerson subtypes, not just ABPersonVendor.

The plugin FindDuplicatesPlugin defines the ABContact subtypes used in this search in the variable WIDE_MAP.

See also

- "IFindDuplicatesPlugin plugin interface" on page 328

Merge duplicate contacts

Before you begin

Merging duplicate contacts requires that Duplicate Contacts Finder batch processing has run. See:

- "Configuring Duplicate Contacts Finder batch processing" on page 271
- "Running the Duplicate Contacts Finder work queue manually" on page 272
- "Setting Duplicate Contacts Finder to run automatically" on page 273

After Duplicate Contacts Finder batch processing has run, you can see any potential duplicate contacts that this process found by clicking **Merge Contacts** on the **Contacts** tab.

Procedure

1. Log in as a user with the Contact Manager role.
   For example, log in with username su and password gw.

   The Contact Manager role includes the permission to view the merge screen, abviewmerge. To merge contacts, there are additional permissions to edit and delete contacts that are also part of the Contact Manager role, such as abedit, abdelete, and anytagedit.
2. Click the **Contacts** tab.
3. In the Sidebar, click **Merge Contacts**.
4. In the **Merge Contacts** screen, you see all the duplicate contacts detected by the last run of Duplicate Contacts Finder batch processing. The duplicates are grouped into pairs. You typically click **Review** to resolve one pair at a time, but you can mark multiple contact pairs in this screen and click **Ignore** for all of them.
   - You can select the check box next to one or more pairs that you determine are not duplicates and click **Ignore** to ignore all the checked pairs. Clicking **Ignore** saves both contacts in each pair and then removes each duplicate contact entry. These duplicate entries will not show up in future runs of the batch process unless a future edit makes them duplicates again.
   - You can search for specific duplicate contacts. If you do a search, you can use the **Match Type** list to filter the results by exact match, potential match, or all matches.

- You might search for a specific contact's duplicates on this screen and then want to see all the duplicates again. To do so, click **Reset** and then click **Search**.

5. If you do not see any potential duplicate contacts listed, it is possible that Duplicate Contacts Finder has not run. It could also have run more than once since the last time you viewed the list.
   - If you clear the **Last Run Only** check box, you can see all duplicates that remain from any run of Duplicate Contacts Finder.
   - If that does not work, you need to run Duplicate Contacts Finder batch processing.
6. For any pair of contacts, if you click the **Review** button, you can compare the two contacts on the **Review Contacts for Merging** screen.

   The **Review Contacts for Merging** screen has four actions you can perform on the two contacts you are comparing:
   - **Merge** – The contacts are duplicates. After you have completed all the comparisons on all tabs and the data is correct for both versions of the contact, click **Merge** to save the data in one contact. Clicking **Merge** removes the contact that is a duplicate and this duplicate contact entry. This duplicate entry will not show up in future runs of the batch process.
   - **Merge Then Edit** – The same as **Merge**, except that the merged contact opens in an editor after ContactManager completes the merge.
   - **Ignore** – The contacts are not duplicates. Clicking **Ignore** saves both contacts and removes this duplicate contact entry. This particular duplicate entry will not show up in future runs of the batch process unless a future edit makes them duplicates again.
   - **Cancel** – You do not want to decide if these two contacts are or are not duplicates. Clicking **Cancel** preserves this duplicate contact entry for this run of the batch process. However, the next run of the batch process will not pick up these duplicates. To see them in the **Merge Contacts** screens after any subsequent run of the batch process, you clear the **Last Run Only** check box, as described previously in "step 5".
7. On the **Review Contacts for Merging** screen, there are multiple tabs. Complete your work on all tabs before clicking **Merge** or **Merge Then Edit** at the top of the screen.
   - Initially you see the **Contact Detail** tab with four columns. The columns show the field names, the data for the contact to be kept, the data for the contact to be retired, and the data resulting from the merge.
   - The **Addresses** tab enables you to choose replacement addresses and to choose more than one address for the merged contact. You can also change the primary address and set the address type for each address. Check **Include** for any address that you want to keep that is not the primary address. If you do not include an address, you must indicate which address duplicates it in the address's **Duplicate Address** list. You can choose **None** from this list if the address is not duplicated by another address.
   - The **Documents** tab enables you to choose which documents to keep. Set **Include** for documents you want to be attached to the merged contact. Clear **Include** for documents you want to remove from the merged contact.
   - The **Related Contacts** tab enables you to choose which related contacts to keep.
   - The **EFT Information** tab enables you to choose which electronic funds transfer accounts to keep.
   - The **Vendor Data** tab is visible if one of the contacts has a vendor tag. This tab enables you to determine which tags, availability settings, and services to keep.
   - If there are any review scores for the contacts, the scores merge and update the next time the AB Contact Score Aggregator batch process runs.

   Review scores are a ClaimCenter feature described in "ClaimCenter service provider performance reviews" on page 237.
8. When the **Updated Contact** column has all the correct data, you can click **Merge** or **Merge Then Edit**.
   - **Merge** – Save the contact with the information you have chosen and exit the merge screens for this contact. ContactManager saves the **Kept** contact and retires the **Retired** contact. ContactManager then notifies all integrated core applications that the contact has changed. On receiving this notification, the core applications

can change references to the retired contact and make them references to the kept contact. See “Merging contacts and notifying core applications” on page 277.

- **Merge Then Edit** – Save the contact with the information you have chosen, and then open the **Kept** contact in the editor. ContactManager first saves the **Kept** contact and retires the **Retired** contact. When the editor opens, you see the newly merged data for the **Kept** contact in the editor. You can make further changes and save your changes.

ContactManager notifies all integrated core applications that the contact has changed. On receiving this notification, the core applications can change references to the retired contact and make them references to the kept contact. See “Merging contacts and notifying core applications” on page 277.

Results

If you merge vendor contacts that have vendor review scores, you do not see the updated average scores until AB Contact Score Aggregator batch processing runs. If you use vendor review scores, it is likely that you have scheduled the writer for this process to run regularly. If you want to see the updated average scores right away, you can run AB Contact Score Aggregator as described under “Start work queues and batch processes” on page 279.

Merging contacts and notifying core applications

When you merge duplicate contacts, ContactManager keeps one contact and retires the other. As with any contact change, ContactManager then notifies the integrated core applications that a contact has been merged. ContactManager calls the core application implementation of the `ABClientAPI` interface method `mergeContacts` to notify the application of the kept contact and retired contact `AddressBookUID` values.

The application can then update the contact’s `AddressBookUID` with that of the kept contact. Each core application handles the change appropriately for the application. For example, the core application updates local copies of the retired contact to use the `AddressBookUID` of the kept contact. It then retrieves the data for the kept contact from ContactManager.

ContactManager provides two `ABContactAPI` web service methods, `getReplacementAddress` and `getReplacementContact`, that the core application can call to update the contact data by using the `AddressBookUID` of the kept contact.

In the base configuration, the core applications implement the `ABClientAPI` interface in the class `ContactAPI`. You can see how each application implements `mergeContacts` by opening the class in Guidewire Studio™ for that application. Each application uses a different path for the class:

BillingCenter

`gw.webservice.bc.bc1000.contact.ContactAPI`

ClaimCenter

`gw.webservice.cc.cc1000.contact.ContactAPI`

BillingCenter

`gw.webservice.pc.pc1000.contact.ContactAPI`

See also

- “Merge duplicate contacts” on page 275
- For a description of the method `ABClientAPI.mergeContacts`, see “ABClientAPI interface” on page 310.
- For descriptions of the methods `ABContactAPI.getReplacementAddress` and `ABContactAPI.getReplacementContact`, see “ABContactAPI methods” on page 303.

Review pending changes to contacts

About this task

Pending changes are saved in ContactManager when a ClaimCenter user who does not have `abcreate` or `abedit` permissions creates or edits a vendor contact. For a pending create or pending update to become a new contact or to update a contact, an authorized user must log in to ContactManager and approve it.

- For a pending create, ContactManager first creates the contact and then flags it as Pending Create. Approving the create just removes the flag. Disapproving the create removes the pending contact.
- For a pending update, ContactManager stores the update information, but does not apply it to the contact until you approve it. If you disapprove the update, this information is deleted and is never applied to the contact.

Procedure

1. Log in to ContactManager in a role that has permissions to create, update, view, delete, and search for contacts and to view pending contacts. For example, log in as a user with the Contact Manager role, such as the sample user `aapplegate/gw`.

   For information on contact permissions and security, see "ContactManager contact security" on page 85.
2. Click the **Contacts** tab and then click **Pending Changes**.
3. On the **Pending Changes** screen, click either the **Updates** card to work with pending contact changes or the **Creates** card to work with new pending contacts.
4. Review each pending Update or Create request.
   - **Updates** – There is a list of contacts with information on the contact name, the ClaimCenter user who made the change, and the associated claim. When you select a contact, then, under **Entity or Property**, you can navigate to the fields that were changed. For each field, you can see the **Current Value** and the submitted pending change, the **New Value**.
   - **Creates** – There is a list of contacts with information on the contact name, the type of contact, the tags, the requesting user, and the claim number. When you select a contact, the contact details display below on the **Basics**, **Addresses**, and **Related Contacts** cards.
5. You can **Approve**, **Reject**, or **Approve Then Edit** the entire pending update or create for a contact. You can also click **Find Duplicates** for a pending create.
   - **Reject** – If you reject the pending change or create, you see a screen asking for a reason and a text explanation. A reason is required. After selecting a reason and optionally entering an explanation, click **Reject Change** to complete the rejection.

     ContactManager sends the rejection to ClaimCenter, which creates an activity for the user with the information you chose. ClaimCenter also changes the status message for the contact to indicate that the change did not go through.
   - **Approve** – If you approve the change, the contact is updated or created, as needed.
   - **Approve Then Edit** – If you want to make further changes to the contact data, click **Approve Then Edit**. The contact is updated or created, as needed, and then ContactManager opens the contact in an editor. At this point, the approved data is saved with the contact, and the contact is open in an editor so you can make further changes.

     If the contact approval was for a pending update, ContactManager shows you both the old data and the new data in a worksheet below the **Edit** screen.

     Note: If you cancel the edit or click **Return to Pending Updates**, you are effectively deciding not to edit the contact data. Canceling the edit is the same as approving the contact update. The old data is no longer available on the worksheet, but you can still access it. To see the old data, search for the contact and open its detail screen, and then click the **History** card to see the list of changes. For a particular change, click **Changes** to see the specifics.

- **Find Duplicates** – This button is available for a pending create only. If you want to check for duplicate contacts for the current contact that is pending creation, click **Find Duplicates**. If there are exact or potential matches, you see a list of the contacts in a worksheet. If one of the contacts is a duplicate of the pending create, you can click **Accept** for that contact.

  ContactManager keeps the selected duplicate contact and retires the pending contact. ContactManager then sends the same message to ClaimCenter that it sends for any other duplicate contact merge. The message indicates the `AddressBookUID` of the retained contact and the `AddressBookUID` of the retired contact. ClaimCenter uses this information to update the link for its contact to the retained contact's `AddressBookUID`.

Results

Note: If you do not check for duplicates and you approve creation of this contact, ContactManager does not automatically check for duplicate contacts for you. The new contact is created. If the new contact is a duplicate of an existing contact, you can make the correction in the **Merge Contacts** screens after Duplicate Contacts Finder batch processing runs. See "Merge duplicate contacts" on page 275.

Pending changes screen cache

The **Pending Changes** screen in ContactManager uses a cache for `DiffDisplay` objects to improve performance of this screen. These cached objects contain the display information for the changes being made as part of a pending update from an external system.

There are two configuration parameters for this cache in the `config.xml` configuration file, which you can edit in Guidewire Studio™ for ContactManager:

`MaxDiffDisplaysInCache`

Controls the maximum size of the cache. The default value is 60 entities.

`DiffDisplaysCacheTimeoutInMinutes`

Sets the amount of time since last access, in minutes, that a `DiffDisplay` object times out of the cache. The default value is 10 minutes.

The following rules remove entries in the cache:

- The rule Pending Contact Cache Update in the `ABContactPreupdate` rule set runs when an `ABContact` entity has changed. The rule removes entries in the cache for that entity.
- The rule Pending Contact Cache Update in the `PendingContactChangePreupdate` rule set runs when a pending change for an `ABContact` entity has been rejected. The rule removes entries in the cache for that entity.

See also

- "Review pending changes to contacts" on page 278
- "ABContact preupdate rule set" on page 211
- "PendingContactChangePreupdate rule set" on page 211

Working with work queues and batch processes

To get information about work queues and batch processes and start and stop them manually, use the **Batch Process Info** screen.

Start work queues and batch processes

Before you begin

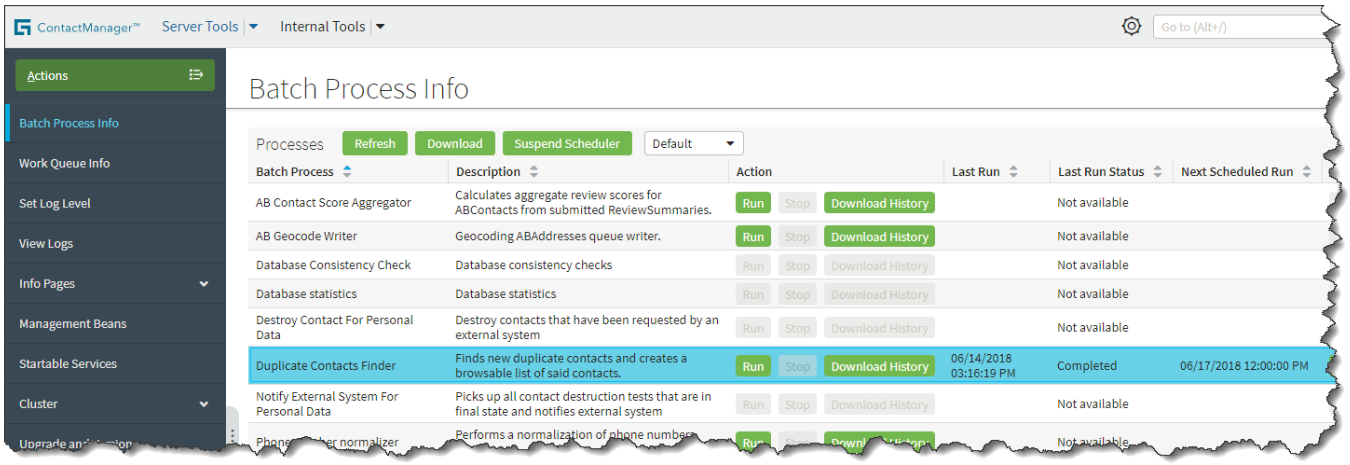
To start a work queue or batch process manually, you first log in to ContactManager as a user with an administrative role. The role must have the `toolsBatchProcessview` permission. See "Log in to ContactManager" on page 261.

Procedure

1. In ContactManager, press `Alt+Shift+T`.
2. Click the **Server Tools** tab.
3. In the Sidebar on the left, click **Batch Process Info**.

Batch Process Info screen

The following figure shows the **Batch Process Info** screen with **Duplicate Contacts Finder** selected:



Batch Process	Description	Action	Last Run	Last Run Status	Next Scheduled Run
AB Contact Score Aggregator	Calculates aggregate review scores for ABContacts from submitted ReviewSummaries.	Run Stop Download History		Not available	
AB Geocode Writer	Geocoding ABAddresses queue writer.	Run Stop Download History		Not available	
Database Consistency Check	Database consistency checks	Run Stop Download History		Not available	
Database statistics	Database statistics	Run Stop Download History		Not available	
Destroy Contact For Personal Data	Destroy contacts that have been requested by an external system	Run Stop Download History		Not available	
Duplicate Contacts Finder	Finds new duplicate contacts and creates a browsable list of said contacts.	Run Stop Download History	06/14/2018 03:16:19 PM	Completed	06/17/2018 12:00:00 PM
Notify External System For Personal Data	Picks up all contact destruction tests that are in final state and notifies external system	Run Stop Download History		Not available	

The **Batch Process Info** screen enables you to run work queues and batch processes manually, without having to wait for the scheduled writers to run. You can start ContactManager processes and view information about them, including writers for work queues. This information includes the **Name**, **Description**, **Status**, **Last Run** time, **Last Run Status**, **Next Scheduled Run** time, and scheduling information.

To start a work queue or batch process manually, click **Run** in the **Action** column.

There are several batch processes you might find useful to run from this screen, especially during development.

Batch Process	Description
AB Contact Score Aggregator	Processes service provider review scores sent by ClaimCenter. For more information, see “ClaimCenter service provider performance reviews” on page 237.
AB Geocode Writer	Geocodes addresses. For more information, see “Geocoding and proximity search for vendor contacts” on page 73.
Duplicate Contacts Finder	Compares new contacts to the existing contacts in the database to see if there are entries that might be duplicates of the contact. After Duplicate Contacts Finder runs, all new contacts that have been processed are marked to prevent them from being processed again. Possible duplicate contacts are saved to a list that a user with the appropriate permissions can process in the Merge Duplicates screen. For more information, see “Detecting and merging duplicate contacts” on page 271.

These batch processes are writers for work queues and have entries in the file `scheduler-config.xml` that you can uncomment and set to run at specific times.

See also

- “Start work queues and batch processes” on page 279
- “Scheduling work queue writers” on page 273
- “Schedule geocoding” on page 78.

- For information on administering batch processing, see the *Administration Guide*

Improve query performance for large batch jobs

About this task

If you run a large batch job on more than ten percent of the addresses in the ContactManager database, query performance can be impacted. To improve query performance, after running the batch job you can run a series of database statistics statements from the command line.

Procedure

1. At a command prompt, navigate to `ContactManager/admin/bin`.
2. Run the following command:

   ```
   maintenance_tools -password password -getdbstatisticsstatements | grep -i ab_abaddress > filename
   ```

   In this command, *`password`* is your administration password and *`filename`* is the file where you want the output of the command to be saved. The `grep` command is available on UNIX or Linux systems. On Windows systems, you can run `grep` in a UNIX or Linux emulator like Cygwin.
3. Open the output file and run the database statistics statements in it.

What to do next

See also

- For information on database statistics, see the *Administration Guide*

Vendor services onboarding

By using the **Vendor Services Onboarding** screen, you can add vendor services to your vendor contacts. This process is useful for initially updating your database of contacts that do not yet have vendor services. For example, you might perform this process after upgrading from a previous release to the current version of ContactManager. Or you might perform this process as part of an initial implementation after migrating contacts from your legacy system to the ContactManager database.

The vendor services you associate with your contacts are described at “Vendor services” on page 163. Vendor services enable you to specify the services provided by vendor contacts in greater detail than is supported by contact subtypes. In ContactManager, after you create your set of vendor services, you can add services to each contact by editing the contact. However, if you have a large number of contacts that need services added to them, adding services to each contact, one at a time, might not be practical.

You can use vendor services onboarding to process all your vendor contacts and add services to them.

IMPORTANT: Before you can use this feature, you must:

- Create and load your vendor services tree. See “Vendor services” on page 163.
- Ensure that all your contacts are in a current ContactManager database.
  - If you are upgrading from a previous release, you must complete the upgrade to this release before starting the process described in this topic.
  - If you are implementing a new installation of ContactManager, you must have transferred all your legacy contacts to the ContactManager database, preferably as appropriate `ABContact` subtypes. In the base configuration, vendor services onboarding uses the contact subtype to determine appropriate vendor services to apply. You can configure this process to use other criteria.
- Ensure that all contacts to which you want to add services have a Vendor tag. The standard ContactManager upgrade process can tag all contacts that are vendor subtypes with the Vendor tag. If you have previously worked with vendor contacts in ClaimCenter 7.0, 8.0, or 9.0, ClaimCenter has tagged your contacts for you. See “Contact tags” on page 159.

- Set up your vendor service mappings in the `ServiceMappings` class. See "Adding service map settings to the ServiceMappings class" on page 282.
- Optionally define settings for the Vendor Services Load Status column, which you can use to track your work on each contact. See "Configure vendor services load status" on page 284.

Initial configuration for vendor services onboarding

Before you can use the **Vendor Services Onboarding** screen, you must set up your vendor service mappings in a Gosu class. Additionally, you might want to configure an optional tracking field supplied in the base configuration, which is saved as the column Vendor Services Load Status in the exported CSV files.

Adding service map settings to the ServiceMappings class

You must edit the `ServiceMappings` class to define mappings between sets of contact field values and services. ContactManager uses these mappings to apply services to contacts.

To avoid having to stop and restart ContactManager, you can set this class up before starting the vendor services onboarding process in the user interface. Alternatively, you can generate the initial CSV files first, Step 1 in the user interface, to see what the values of the `Key` column are for each contact. You can then enter the key values and their associated vendor service codes in this class.

Note: If you edit the `ServiceMappings` class after generating the initial CSV files, you must restart ContactManager.

The `ServiceMappings` class defines pairs of mappings between contact keys defined in the `ExportImportVendorServicesUtil` class and vendor service codes. These mapping pairs determine which vendor services are assigned to a contact in the mapping stage.

You can edit the `ServiceMappings` class in Guidewire Studio™ for ContactManager. In this class, you enter your actual key value and service code pairs, as described later in this topic. When you are done, restart ContactManager.

In the base configuration, the `ServiceMappings` class comes with four sample `HashMap String` pairs. Each pair is a `Key` value from the CSV file associated with one or more vendor service codes:

```
"Auto Repair Shop United States" -> "autoothercourtesycar",
"Adjudicator United States" -> "autoappraise, autoadjudicate, propinspectadjudicate",
"Doctor United States" -> "medicalcare",
"Vendor (Company) United States" -> "contrepairgarden, contrepairkitchen"
```

With these default mappings, performing the mapping step in the user interface assigns the following services to the following contacts:

- Vendor contacts of subtype `AutoRepairShop` with a primary address in the United States are assigned the service Provide Courtesy Car.
- Vendor contacts of subtype `Adjudicator` with a primary address in the United States are assigned the services Auto - Appraisal, Adjudicate Claim, and Property - Inspection - Adjudicate Claim.
- Vendor contacts of subtype `Doctor` with a primary address in the United States are assigned the service Medical - Medical care.
- Vendor contacts of subtype `CompanyVendor` with a primary address in the United States are assigned the services Contents - Repair - Garden and Contents - Repair - Kitchen Appliances.

Key value service map setting

The key value is the first value of the service mapping pair in the `ServiceMappings` class. It is generated for each contact when you initially generate the CSV files. Each CSV file has a `Key` column. The values stored in the `Key` column are defined in the class `ExportImportVendorServicesUtil` in its `keys` property. In the base configuration, the default key value is a concatenation of the `ABContact` properties `Subtype` and `PrimaryAddress.Country`:

```
private static var keys = {"Subtype", "PrimaryAddress.Country"}
```

In the base configuration, for each contact added to the CSV file, ContactManager reads the values for the contact's `Subtype` and `PrimaryAddress.Country` fields. It then concatenates these values into a single string and populates the `Key` column for that contact. For example, for sample contact European Autoworks located in San Francisco, California, USA, the `Key` column value would be:

```
Auto Repair Shop United States
```

You do not have to use these two `ABContact` fields to generate the `Key` column. In `ExportImportVendorServicesUtil.keys`, you can specify any `ABContact` field or fields that you want to use to map contacts to services. You would then need to set up the `ServiceMappings` definitions to use the actual values in the `Key` columns generated for your CSV files. See the description of the `key` property in "Properties of ExportImportVendorServicesUtil" on page 293.

See also

- "Adding service map settings to the ServiceMappings class" on page 282

Vendor service codes service map setting

Vendor service codes are the second value of the service mapping pair in the `ServiceMappings` class. This value consists of one or more vendor service codes that you want ContactManager to assign to a contact with that specific key value.

To see your vendor service codes, you can export the file `vendorservicetree.xml` in the **Administration** tab by navigating to **Utilities** > **Export Data**. For the **Data to Export** column, select Vendor Service Tree and click **Export**.

The base configuration mapping definitions in `ServiceMappings` use vendor service codes from the sample `vendorservicetree.xml` file in `ContactManager/modules/configuration/config/sampledata`. For example, the default code defined for `"Auto Repair Shop United States"`, `"autoothercourtesycar"`, is the `<Code>` value for the `Provide courtesy car` service:

```
<SpecialistService public-id="svc:aut_oth_cou">
  <Active>true</Active>
  <Code>autoothercourtesycar</Code>
  <Description displayKey="true">Services.Auto.ProvideCourtesyCar.Description</Description>
  <Name displayKey="true">Services.Auto.ProvideCourtesyCar.Name</Name>
  <Parent public-id="svc:aut_oth"/>
</SpecialistService>
```

See also

- "Adding service map settings to the ServiceMappings class" on page 282

Generated contact data columns in the vendor services CSV File

In the base configuration, there are five contact data columns defined in `ExportImportVendorServicesUtil` that become the first five columns of the CSV file generated from the **Vendor Services Onboarding** screen. These five columns are `Name`, `LinkID`, `Address`, `Key`, and `Vendor Services Load Status`. You can configure different contact data columns as needed, as described in the table of "Properties of ExportImportVendorServicesUtil" on page 293.

The first three columns correspond to standard fields on `ABContact`: `Name` or `FirstName` + `LastName`, `LinkId`, and `PrimaryAddress`.

The fourth column, `Key`, is defined in the property `ExportImportVendorServicesUtil.keys`, and is used as part of the vendor service mapping process.

The fifth column, `Vendor Services Load Status`, might require configuration before you can make use of it.

See also

- "Key value service map setting" on page 282
- "Configure vendor services load status" on page 284

Configure vendor services load status

About this task

The generated CSV column, `Vendor Services Load Status`, might require configuration before you can make use of it. You can use this column to track your progress in assigning vendor services to a contact.

Procedure

1. Check `ABContact.etx` to see if it has the typekey `VendorServicesLoadStatus`.
    a) In Guidewire Studio™ for ContactManager, press `Ctrl+Shift+N` and enter `ABContact`.
    b) In the search results, double-click `ABContact.etx` to open it in the editor.
    c) Look for the typekey `VendorServicesLoadStatus`. If it is on this entity, it is likely to be listed at the end.
2. If the typekey does not exist, add it.
    a) At the top of the tree on the left, right-click **entity (extension)** `ABContact` and choose **Add new** > **typekey**.
    b) Enter the following values:
        - **name** – `VendorServicesLoadStatus`
        - **typelist** – `VendorServicesLoadStatus`
        - **nullok** – `true`
3. If needed for your tracking purposes, extend the `VendorServicesLoadStatus` typelist to have more values than `Final` and `Draft`.
    a) Navigate to **configuration** > **config** > **Metadata** > **Typelist** and right-click `VendorServicesLoadStatus.tti`.
    b) Choose **New** > **Typelist extension**, and then click **OK** in the **Typelist Extension** dialog.
    c) Studio opens a new `VendorServicesLoadStatus.ttx` file in the directory at **configuration** > **config** > **Extensions** > **Typelist**.
    d) To add each typecode in the Typecode editor, right-click a row in the panel on the left and choose **New** > **typecode**. Then enter a code, name, and description for the new typecode.
4. Restart ContactManager to pick up these changes.

What to do next

See also

- “Remove the vendor services load status field” on page 285
- “Generated contact data columns in the vendor services CSV File” on page 283

Generate and use vendor services load status

Before you begin

You must export a vendor services CSV file before using this procedure. Additionally, you might need to configure the vendor services load status.

Procedure

1. Open an exported CSV file in Microsoft Excel and select the row for a contact that you want to track.

    The value of the `Vendor Services Load Status` field for the contact by default is initially `null`. It is an empty string, a blank, in the spreadsheet cell.
2. In the `Vendor Services Load Status` cell for that contact, enter a valid tracking value for the contact, such as `Draft`.

Valid values are typecodes from the `VendorServicesLoadStatus` typelist, which by default are `null`, `Draft`, and `Final`.

3. Save the file.
4. The next time you import the CSV file, ContactManager copies the value in the `Vendor Services Load Status` cell and saves it the contact's `VendorServicesLoadStatus` typekey field.

What to do next

See also

- "Configure vendor services load status" on page 284
- "Export the initial set of vendor services onboarding CSV files" on page 286.

Remove the vendor services load status field

About this task

You might not want to use Vendor Services Load Status to track your work on vendor service contacts. If not, you can remove the field. For more information, see "Generate and use vendor services load status" on page 284.

Procedure

1. In Guidewire Studio™ for ContactManager, press `Ctrl+N` and enter `ExportImportVendorServicesUtil`.
2. In the search results, double-click `ExportImportVendorServicesUtil` to open this class in the editor.
3. Find the property `initialHeaders` and delete the following code from the property definition:

```
, ABContact#VendorServicesLoadStatus.PropertyInfo.DisplayName
```

4. Comment out the following line of code:

```
private static var vendorServicesLoadStatusIndex = 4
```

5. Find the property `numberOfNonServiceColumns` and change its value from `5` to `4`.

   The setting for `numberOfNonServiceColumns` matches the number of fields generated in the base configuration. If you have added or removed fields, your setting for `numberOfNonServiceColumns` will differ.

6. In the method `updateVendorServicesInDB`, comment out the following lines of code:

```
if (cells.length > vendorServicesLoadStatusIndex) {
  var loadStatus = cells[vendorServicesLoadStatusIndex]
  if (VendorServicesLoadStatus.get(loadStatus) == null && loadStatus != "") {
    var cellList = cells.toList() as ArrayList<String>
    while (cellList.size() < header2.size()) {      cellList.add("")
    }
    cellList.add(cellList.size(), "Bad VendorServiceLoadStatus value")
    recordErrorMessage("Row with name '" + cells[nameColumnIndex] +
      "' and id '" + cells[idColumnIndex] +
      "' has bad VendorServiceLoadStatus value")
    errors.add(cellList as String[])
  } else {
    vendor.VendorServicesLoadStatus =
      VendorServicesLoadStatus.get(cells[vendorServicesLoadStatusIndex])
  }
}
```

7. Remove the typekey `VendorServicesLoadStatus` from `ABContact.etx`.
8. Restart ContactManager to pick up these changes.

Working with vendor services onboarding

After completing the configuration steps described in "Initial configuration for vendor services onboarding" on page 282, you use the **Vendor Services Onboarding** screen to apply vendor services to contacts.

Adding vendor services to your vendor contacts is a multi-step process that you perform in the ContactManager **Vendor Services Onboarding** screen on the **Administration** tab.

To start, navigate to **Utilities** > **Vendor Services Onboarding** and then perform the following multistep procedures:

1. "Export the initial set of vendor services onboarding CSV files" on page 286
2. "Map services to vendor contacts" on page 287
3. "Edit the generated vendor services onboarding CSV files" on page 288
4. "Import mapped contacts in vendor services onboarding CSV files" on page 289
5. "Troubleshooting vendor service importing, exporting, and mapping errors" on page 289

Export the initial set of vendor services onboarding CSV files

Before you begin

Before you perform this step, open the **Vendor Services Onboarding** screen as described at "Working with vendor services onboarding" on page 285.

About this task

The first user interface step in vendor services onboarding is to generate a set of CSV files. Each file has one contact subtype in it, with one row for each vendor contact and columns representing each possible vendor service they might provide. These files are used later to map contacts to vendor services, and then to import those mappings into ContactManager.

If you encounter errors during this process, see "Troubleshooting vendor service importing, exporting, and mapping errors" on page 289.

Procedure

1. On the **Administration** tab, navigate to **Utilities** > **Vendor Services Onboarding.**
2. In the **1. Export Vendors to CSV** section, click **Export**.
   ContactManager generates a set of CSV files, each of which contains one subtype of `ABContact`.

Results

You can see various messages in the **Process Log** section of the screen:

- If the export is successful, you see the message, **Export complete. Files located at C:\outputfiles\**. The directory listed in this message depends on your setting in `ExportImportVendorServicesUtil.outputFolder`. In the base configuration, this top level directory is `C:\outputfiles\`.
- If you have changed a service type to a category, you see a warning message stating that the service type will not import. For example: **Branch service Adjudicate Claim exported but will not import**.

What to do next

The next step is "Map services to vendor contacts" on page 287.

See also

- "Files generated by exporting vendors to CSV" on page 287
- "Change an existing vendor service into a category" on page 166
- "Customizing vendor services onboarding" on page 293

Files generated by exporting vendors to CSV

Exporting files in the **Export Vendors to CSV** screen generates a set of CSV files, each of which contains one subtype of `ABContact`. In the base configuration, this action produces a directory structure and set of files similar to the following:

```
c:\ouputfiles\
  VendorServicesLoad-Mon Mar 03 16.15.50 PST 2017\
    Adjudicator-0.csv
    Attorney-0.csv
    Auto Repair Shop-0.csv
    Doctor-0.csv
    Law Firm-0.csv
    Medical Care Organization-0.csv
    Vendor (Company)-0.csv
```

Each CSV file is specific to one subtype of contact, as its name indicates. ContactManager creates a set of these files in a directory that you can specify. Part of the immediate directory name is a timestamp, enabling you to create separate sets of files as you test your output and refine the process.

The top level directory is defined in `ExportImportVendorServicesUtil.outputFolder`.

The time-stamped directory is defined in the `timeStampedFolderName` property of the `ExportImportVendorServicesUtil.exportVendors` method.

In the base configuration, each CSV file contains up to 500 contacts of the subtype specified. The contacts are the data rows of the spreadsheet. The number of contacts per CSV file is defined in `ExportImportVendorServicesUtil.maxRowsPerSpreadsheet`.

- If there are more than 500 contacts of that subtype, ContactManager creates additional CSV files with that subtype name. ContactManager increments the number in the file name by one for each additional file it creates.
- For example, one additional file is added for `Doctor-0.csv`. This file is named `Doctor-1.csv`.

In the base configuration, for each contact, there are five columns of general data, followed by columns for all your services. The default contact data columns are Name, LinkID, Address, Key, and Vendor Services Load Status.

- These data column names are defined in `ExportImportVendorServicesUtil.initialHeaders`.
- The number of data columns is defined in `ExportImportVendorServicesUtil.numberOfNonServiceColumns`.
- Four of the data columns have database index properties defined in `ExportImportVendorServicesUtil` to improve database access performance. For example, the LinkID column has the index property `idColumnIndex`.
- The value of the `Key` column is defined in `ExportImportVendorServicesUtil.keys` and is used by the `ServiceMappings` class. See “Adding service map settings to the ServiceMappings class” on page 282.

See also

- The Vendor Service Load Status column is described in “Configure vendor services load status” on page 284.

Map services to vendor contacts

Before you begin

Before you perform this step:

- Complete the step “Export the initial set of vendor services onboarding CSV files” on page 286.
- Ensure that you have set up the `ServiceMappings` class with your own key and vendor service code pairs. See “Adding service map settings to the ServiceMappings class” on page 282.

About this task

The second step on the **Vendor Services Onboarding** screen is mapping services to contacts. If you encounter errors during this process, see “Troubleshooting vendor service importing, exporting, and mapping errors” on page 289.

Procedure

1. If you previously opened any of the CSV files in an editor like Excel, be sure to close them before you perform this step.
2. On the **Administration** tab, navigate to **Utilities** > **Vendor Services Onboarding.**
3. In the **2. Map Services to Vendors in CSV** section, enter the path to your time-stamped directory and click **Map**.

   In this screen, you can also enter the path to a specific file. Additionally, you can enter a directory path that has subdirectories and use the **Include Subdirectories** check box to process all subdirectories.
4. If the mapping is successful, you see the message **Mapping Complete** in the **Process Log** section of the screen. This area is also where you see error messages.

Results

After the mapping is complete:

- The CSV files have `_mapped` added to their names. Any subsequent runs of this step adds another `_mapped` to the file names.
- If a file's name does not change when you perform this step, that file did not get mapped. Check to see if the file is open in an editor and close it if it is. You can run the process again without causing any problems in the files that have already been mapped. For example, after you run the process for the second time, the files have the suffix `_mapped_mapped`.

What to do next

The next step is "Edit the generated vendor services onboarding CSV files" on page 288.

Edit the generated vendor services onboarding CSV files

Before you perform this step, complete the step "Map services to vendor contacts" on page 287.

After generating the CSV files from the **Vendor Services Onboarding** screen, you can open the mapped CSV files in a spreadsheet program like Microsoft Excel and edit them. For example, you might want to manually edit vendor service assignments for some contacts.

> **Note:** When you save the file in Microsoft Excel, you are likely to see a message window asking if you want to save the file in the current format: comma-delimited CSV. That is the correct format. Click **Yes** to save the file in that format.

There are a number of things you need to be careful about when editing these files:

- In the base configuration, the default value for a vendor service that is mapped to a contact is the `String` value `"On"`. For unmapped vendor services, the default value is the empty string, `""`. These values are defined in the `onValue` and `offValue` properties of `ExportImportVendorServicesUtil`. Be sure to use the values you have defined when manually assigning vendor services to and removing them from a contact.
- In general, avoid editing the data columns for contacts, especially `LinkID` and `Key`. The `Key` field is necessary for vendor service mapping to work properly. The `LinkID` field is necessary for importing and saving the contact data with the correct contact.
- In general, do not manually add or remove vendor service columns in a generated CSV file.

  If you remove a service column that is not used in the mapping, you see an error message, but the file still gets processed.

  - If you are still refining your service tree, wait until the tree is complete before starting the vendor service onboarding process.
  - If you change your service tree, you can perform the entire vendor service onboarding process again. For example, you can set up the `ServiceMappings` class to process only your new services. Any vendor services that you have already imported and applied to contacts are preserved for those contacts when ContactManager generates the CSV files.

- Do not change any of the header rows, especially the second header row for vendor service columns, which contains vendor service codes.
- While you can delete a contact from a CSV file, which prevents that contact from being processed, do not manually add a contact. Instead, set things up so the file export in Step 1 can add the contact. For example, some contacts do not get added to the CSV file because they do not have vendor tags. You can edit the contacts in ContactManager and add Vendor tags to them, and then regenerate the CSV files.
- Do not delete the generated files, especially if you have edited them, until you have performed the import part of the process.

The next step is “Import mapped contacts in vendor services onboarding CSV files” on page 289.

Import mapped contacts in vendor services onboarding CSV files

Before you begin

Before you perform this step, complete the step “Edit the generated vendor services onboarding CSV files” on page 288.

About this task

When you are satisfied that the mappings are correct for your contacts, you perform the third step in the user interface, importing the contacts with their associated services.

If there are any errors during the import, you see error messages in the **Process Log**. ContactManager continues to process the file and saves any unsuccessful imports in a set of CSV error files that you correct and re-import. See “Errors with importing vendor service contacts” on page 291.

Procedure

1. On the **Administration** tab, navigate to **Utilities** > **Vendor Services Onboarding**.
2. In the **3. Import Vendors with Services from CSV** section, enter the path to your time-stamped directory and click **Import**.

   Note: In this section, you can also enter the path to a specific file. Additionally, you can enter a directory path that has subdirectories and use the **Include Subdirectories** check box to process all subdirectories.
3. If the import is successful, you see the message **Import Complete** in the **Process Log** section of the screen. This area is also where you see error messages.

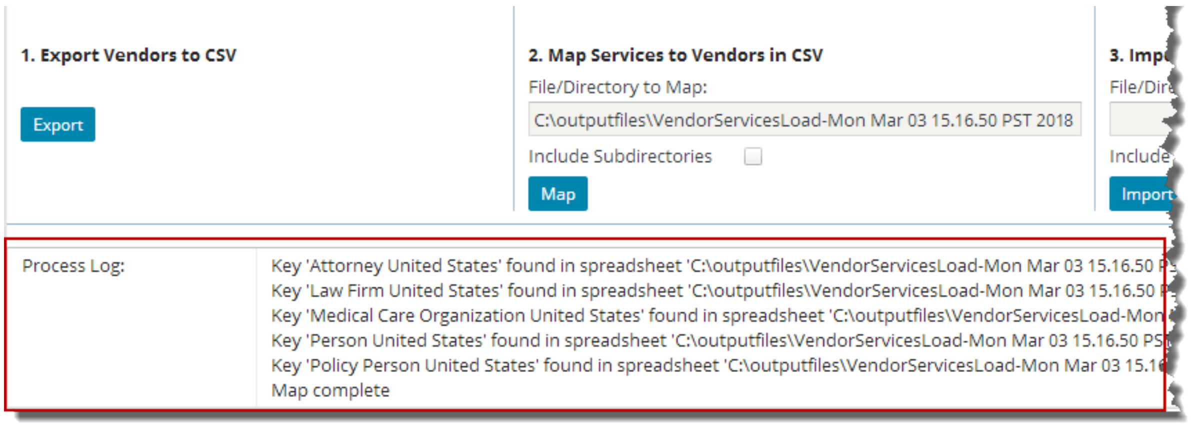
Troubleshooting vendor service importing, exporting, and mapping errors

ContactManager can encounter errors when exporting vendor service files, mapping files, and importing vendor service files.

Note: Removing a contact row does not cause an error. The contact is just not processed.

Because you can edit the CSV files, errors most often occur during mapping and import. You see these errors in the **Process Log** section of the **Vendor Service Onboarding** screen, below the sections with the steps.

For example, in **2. Map Services to Vendors in CSV**, you get a series of key value errors.



```
Process Log:   Key 'Attorney United States' found in spreadsheet 'C:\outputfiles\VendorServicesLoad-Mon Mar 03 15.16.50 P
               Key 'Law Firm United States' found in spreadsheet 'C:\outputfiles\VendorServicesLoad-Mon Mar 03 15.16.50 P
               Key 'Medical Care Organization United States' found in spreadsheet 'C:\outputfiles\VendorServicesLoad-Mon
               Key 'Person United States' found in spreadsheet 'C:\outputfiles\VendorServicesLoad-Mon Mar 03 15.16.50 PS
               Key 'Policy Person United States' found in spreadsheet 'C:\outputfiles\VendorServicesLoad-Mon Mar 03 15.1
               Map complete
```

Note: These errors occurred because ContactManager exported some contact subtypes that were not defined in the `ServiceMappings` class. See "Adding service map settings to the ServiceMappings class" on page 282. See also "Row errors with importing vendor service contacts" on page 291.

Importing and exporting vendor services: errors with files and directories

Because mapping and importing require that files exist, you can get file or directory errors when you do Step 2 or Step 3 of vendor services onboarding. Some typical errors are:

`Filename` **`is not a valid csv file`**

The named file either does not exist (a mapping error) or it has a file extension other than `.csv`. You might get this error if you renamed a file with the wrong extension, or if you saved a file in the directory that was not a CSV file. ContactManager cannot process the file unless it is a comma-delimited CSV file.

`Directory` **`is not a valid csv file`**

You probably entered the wrong directory name, and ContactManager cannot find it.

`IllegalArgumentException`

One cause of this exception is that ContactManager attempted to read a file with a `.csv` extensions that was not a CSV file. For example, you opened one of the generated CSV files in a text editor and attempted to update it manually. Instead, open these files in a spreadsheet editor and save them as comma-delimited CSV files.

Errors with mapping vendor service contacts

The following errors can occur when you map service to vendors in the **2. Map Services to Vendors in CSV** section:

- `Key` *`KeyValue`* `found in spreadsheet` *`CSVFileName`* `but not in map` – This contact has a value in its `Keys` cell that is not defined in the `ServiceMappings` class.

  If you want to map the contact, you must:

  1. Add a matching entry to `ServiceMappings`. See "Adding service map settings to the ServiceMappings class" on page 282.
  2. After adding one or more additional mapping pairs to the `ServiceMappings` class, you must restart ContactManager and then map your files again. See "Map services to vendor contacts" on page 287.

     This condition might not be an error. For example, during initial generation of your CSV files, you generate `Key` values for `Attorney` subtypes, but there are no services you want to map to attorneys. In this case, the message is similar to the following:

     ```
     Key 'Attorney United States' found in spreadsheet 'C:\outputfiles
     \VendorServicesLoad-Mon Mar 03 15.16.50 PST 2014\Attorney-0.csv' but not in map.
     ```

- `Column` *`VendorServiceID`* `was improperly added to spreadsheet headers in file:` *`Filename`* – Each generated vendor service column has two headers, the service name and the service code. There is an invalid service code in the header for one of your vendor service columns. You might either have edited the code value of an existing vendor service or added a vendor service column manually. In general, avoid adding vendor service columns manually and do not edit the header rows. If you did change a code value to an invalid value, you can correct it in the file and remap the file.

- `A service column was deleted from` *`FileOrDirectoryName`* – If you delete a service column from a CSV file, that service cannot be mapped. If the service was not going to be mapped anyway, no harm is done in deleting the column, and you can ignore this message. If you want the service to be mapped, you must add the column back in or regenerate the file and then map it.
- `IndexOutOfBoundsException` – One cause of this exception is that you removed one of the contact data columns from a CSV file, such as the `LinkID` or the `Key` column. The CSV files are not usable without these contact data columns. You must export your files again to recover from this error.

Errors with importing vendor service contacts

If ContactManager encounters any errors during the import process, it displays error messages in the **Process Log** section of the **Vendor Service Onboarding** screen. ContactManager processes all the data it can. If an error prevents a row from being processed, ContactManager copies that row and saves it in a CSV file in the `Errors` directory. It continues processing the rest of the file. If you see errors during import, especially the row errors listed later, check the `Errors` directory for files that have an `error-` prefix added to the original file name. See “Row errors with importing vendor service contacts” on page 291.

Column errors might not prevent a contact from being processed, but they might prevent a vendor service from being added to a contact. See “Column errors with importing vendor service contacts” on page 292.

ContactManager does not alter any rows in the original exported CSV files, so those files will continue to have erroneous data in them unless you change it.

Row errors with importing vendor service contacts

The following row errors can prevent a contact from being imported. ContactManager imports all contacts it can from the file and saves the ones that have errors in an error CSV file. You can edit an error file and re-import it.

`Row with name:` ***`Name`*** **`and id:`** ***`ID`*** **`has bad id value`**

The `LinkID` field is the unique identifier for a contact.

- The most likely cause of this error is that someone edited this field in the CSV file.
- Another cause could be that someone manually added a contact to the file with an invalid `LinkID`. You can correct the `LinkID` value in the error file and then re-import the file.
- The most severe cause of this error is that the entire `LinkID` column was removed, in which case you see an error message for every contact in the file. In this last case, the file is virtually unusable and must be exported and mapped again.

`Row with name:` ***`Name`*** **`and id:`** ***`ID`*** **`has bad on/off value`**

At least one of the `String` values indicating if a vendor service applies or does not apply to a contact has been entered incorrectly. These values are defined in the `onValue` and `offValue` properties of `ExportImportVendorServicesUtil`. Be sure to use the defined values when manually assigning vendor services to and removing them from a contact. You can correct the values in the error file and then re-import the file.

Correcting row errors

Edit and correct any row error files you see. After you fix the data in these error files and save the files, you can re-import the error files. After you re-import an error file, ContactManager removes any contacts that were successfully imported from the file. If there are still contacts with errors in the file, ContactManager adds another `error-` prefix to the file name, and saves the file.

If you get more errors after re-importing error files with fixes in them, continue fixing data in the error files and re-importing them until you get no more errors. When there are no more contacts with errors in an error file, ContactManager deletes the file.

See also

- "Example: correct row errors after importing vendor services" on page 292

Example: correct row errors after importing vendor services

Before you begin

This example of vendor service import errors illustrates how to correct row errors, described generally at "Row errors with importing vendor service contacts" on page 291.

About this task

In this example, ContactManager is unable to process a contact in the file `Auto Repair Shop-0_mapped_mapped.csv`. You see the error in the **Process Log** section while importing the file.

Procedure

1. The contact is saved in a CSV file in the `Errors` directory. This directory is at the same level as the time-stamped directory you are importing from. The name of the CSV file is `errors-Auto Repair Shop-0_mapped_mapped.csv`. The following directory structure is an example:

```
c:\ouputfiles\
  Errors\
    errors-Auto Repair Shop-0_mapped_mapped.csv
  VendorServicesLoad-Mon Mar 03 16.15.50 PST 2018\
    Adjudicator-0_mapped_mapped.csv
    Attorney-0_mapped_mapped.csv
    Auto Repair Shop-0_mapped_mapped.csv
    Doctor-0_mapped_mapped.csv
    Law Firm-0_mapped_mapped.csv
    Medical Care Organization-0_mapped_mapped.csv
    Vendor (Company)-0_mapped_mapped.csv
```

2. Open `errors-Auto Repair Shop-0_mapped_mapped.csv` in Microsoft Excel, make corrections, and save the file.
3. In the **Administration** tab, navigate to **Utilities** > **Vendor Services Onboarding**.
4. In the **3. Import Vendors with Services from CSV** section, enter the path to the error file, and then click **Import**.
5. If there are more errors reported during this re-import, check the `Errors` directory for files with an additional `errors-` prefix, such as `errors-errors-Auto Repair Shop-0_mapped_mapped.csv`. Open these files and fix the errors in the remaining contacts in the files.
6. Continue fixing errors in the error files and re-importing them until there are no more errors reported.
   If there are no more errors in an error file, ContactManager deletes the file after you re-import it.

Column errors with importing vendor service contacts

The following column errors can indicate that a single vendor service was not processed in the CSV file. However, all the other vendor services in the file are processed for all the contacts in the file. These errors do not prevent a contact from being processed, and no error files are saved.

To correct column errors, after the import finishes, you can export the files again. All the data you imported is added to the newly exported files, and the columns are restored. You can then re-import these newly exported files to pick up any column data that was not originally processed.

`A service column was deleted from` *`FileOrDirectoryName`*

If you delete a service column from a CSV file, that service cannot be associated with the contacts in this file. This message can also indicate that you deleted a nonessential contact data column, such as `Address`. This error does not prevent ContactManager from processing the file.

`Column with ID:` *`VendorServiceID`* `cannot be added`

Each generated vendor service column has two headers, the service name and the service code. ContactManager was unable to process a vendor service column because it has an invalid code value in its service code header. This error does not prevent ContactManager from processing the file.

`Column with ID:` *`VendorServiceID`* `cannot be added. File` *`Filename`* `needs to be fixed to complete import`

The most likely cause of this error is that the column represents a service type that was changed to a category. The import will fail unless the column is deleted. See "Change an existing vendor service into a category" on page 166.

Customizing vendor services onboarding

The Gosu class `gw.exportimport.ExportImportVendorServicesUtil` has optional settings and methods that determine how the Vendor Services Onboarding feature works. In this class, you specify values of properties that determine settings like:

- The directory where your CSV files are stored.
- The number of contacts saved in each CSV file.
- The number and name of the contact data columns that precede the service columns in your spreadsheet.
- The number of query results returned for each database query.
- The Vendor Services Load Status column. See "Configure vendor services load status" on page 284.
- The `ABContact` fields that are used to map a vendor service to a contact. See:
  - The `keys` property in "Properties of ExportImportVendorServicesUtil" on page 293
  - "Key value service map setting" on page 282

Additionally, you must configure the mappings between contact properties and vendor service codes to enable mapping for your contacts and services. You configure these mappings in the class `ServiceMappings`. See "Adding service map settings to the ServiceMappings class" on page 282

Properties of ExportImportVendorServicesUtil

The following table lists the properties that you typically customize in `ExportImportVendorServicesUtil`.

Property	Description	Import/Export Step
`outputFolder`	The main folder that ContactManager uses to store the CSV files it exports and imports. Default value is `"C:\\outputfiles\\"`.	All three steps
`queryPageSize`	The number of results returned from a query to the database, by default 100. Affects performance. You can adjust this number if performance is too slow.	All three steps
`maxRowsPerSpreadsheet`	The number of contacts that can be added to a CSV file, by default, 500. When this limit is reached, ContactManager creates a new file and increments the number in the file name.	• "Export the initial set of vendor services onboarding CSV files" on page 286
`exportVendors.timeStampedFolderName`	The time-stamped subfolder that ContactManager uses to store the CSV files it exports and imports. Default value is: `outputFolder + "VendorServicesLoad-" + DateUtil.currentDate().toString().replace(":", ".")` **Note:** The time stamp is an important part of the folder name that supports iterative passes through the contacts.	• "Export the initial set of vendor services onboarding CSV files" on page 286
`initialHeaders`	Defines the names of the first set of columns in the spreadsheet, the contact data columns that show information other than vendor service associations. For example, these columns indicate the `Name`, `LinkId`, and `Key` of the contact. These columns precede the columns	• "Export the initial set of vendor services onboarding CSV files" on page 286

Property	Description	Import/Export Step
	of services, which are defined by the `arraylist` in `columnHeader1`. The actual values of this first set of columns are populated for each contact by the method `createAndAddNextRow`. **Notes:** • The `Key` column is used to map the contact to one or more services. Its value is defined in the `keys` property. The specific value for a contact is used during the mapping stage if it is part of a `HashMap` pair defined in the `ServicesMapping` class. See “Adding service map settings to the ServiceMappings class” on page 282. • `Vendor Services Load Status` is a column you can use to indicate if you have previously loaded services for a contact. It can be used for tracking purposes when you are doing multiple reloads. Some configuration is required to use this column. See “Configure vendor services load status” on page 284. • If you add a new contact data column, also add an index property for it.	
`numberOfNonServiceColumns`	The number of columns specified in `columnHeader2`. If you change the number of contact data columns, also change this value. See also “Remove the vendor services load status field” on page 285.	• “Export the initial set of vendor services onboarding CSV files” on page 286
`keys`	Properties of the contact, saved in the `Key` column of the CSV file. Default key values are `subtype` and `address.country`. These values are used in the vendor mapping step to associate the contact with a vendor, as defined by `HashMap` pairs in the `ServicesMapping` class. See “Adding service map settings to the ServiceMappings class” on page 282.	• “Export the initial set of vendor services onboarding CSV files” on page 286 • “Map services to vendor contacts” on page 287
`onValue`	The value stored in the spreadsheet for a service that is associated with a specific contact. By default, this value is `"On"`.	• “Map services to vendor contacts” on page 287
`offValue`	The value stored in the spreadsheet for a service that is not associated with a specific contact. By default, this value is the empty string, `""`.	• “Map services to vendor contacts” on page 287
`nameColumnIndex`	An index used for mapping and importing the `Name` column defined in `columnHeader2`. The index numbers for the `columnHeader2` columns are zero-relative. This index is index 0.	• “Map services to vendor contacts” on page 287 • “Import mapped contacts in vendor services onboarding CSV files” on page 289
`idColumnIndex`	An index used for mapping and importing the `LinkID` column defined in `columnHeader2`. The index numbers for the `columnHeader2` columns are zero-relative. This index is index 1.	• “Map services to vendor contacts” on page 287

Property	Description	Import/Export Step
		• "Import mapped contacts in vendor services onboarding CSV files" on page 289
`keyColumnIndex`	An index used for mapping and importing the Key column defined in `columnHeader2`. The index numbers for the `columnHeader2` columns are zero-relative. This index is index 3.	• "Map services to vendor contacts" on page 287 • "Import mapped contacts in vendor services onboarding CSV files" on page 289
`vendorServicesLoadStatusIndex`	An index used for mapping and importing the `Vendor Services Load Status` column defined in `columnHeader2`. The index numbers for the `columnHeader2` columns are zero-relative. This index is index 4.	• "Map services to vendor contacts" on page 287 • "Import mapped contacts in vendor services onboarding CSV files" on page 289

Methods of ExportImportVendorServicesUtil

The following table lists the methods that you typically customize in `ExportImportVendorServicesUtil`.

Method	Description	Import/Export Step
`exportVendors`	Does the initial export of vendor contacts and their data and service columns to the CSV files. Creates the directories and files as needed. This method calls `writeCurrentGrid` to create new files with the current contact subtype in the name. If you want to modify this file naming convention, create a new method similar to `writeCurrentGrid` that uses your file naming convention. Then change the code in `exportVendors` that calls `witeCurrentGrid` and have it call your new method.	"Export the initial set of vendor services onboarding CSV files" on page 286
`mapServices`	Maps services to contacts as specified. Uses the settings in the `ServiceMappings` class to do this mapping.	"Map services to vendor contacts" on page 287
`importVendors`	Reads the CSV file or files and saves mapped vendor service data with each contact in the file.	"Import mapped contacts in vendor services onboarding CSV files" on page 289
`createAndAddNextRow`	Adds a contact row to the CSV file. You must edit this method if you want to change any of the contact data columns defined in `columnHeader2` and `numberOfNonServiceColumns`.	"Export the initial set of vendor services onboarding CSV files" on page 286

chapter 15

ContactManager integration reference

Like the core Guidewire applications, ContactManager provides a set of web services, plugins, high level entities, mapping classes, and messaging events. ContactManager is intended to be integrated with the core applications, so it is important to understand the classes and files that enable ContactManager to communicate with the Guidewire core applications.

Overview of ContactManager integration

ContactManager, like the Guidewire core applications, is a complete application built on the Guidewire platform. ContactManager has its own versions of:

- Entities
- Rules
- Plugin interfaces
- Web service (WS-I SOAP) APIs published from the ContactManager application
- Messaging events
- Destination plugins
- Gosu API documentation (Gosudoc)

Integrating ContactManager with core applications from the InsuranceSuite—ClaimCenter, PolicyCenter, and BillingCenter—enables the core applications to use centralized contact management.

- In the ClaimCenter application, you can click the **Address Book** tab to search for and view contacts stored in ContactManager. You can add and edit contacts on a claim's **Contacts** screen and in the New Claim wizard.
- In PolicyCenter, you can click the **Contact** tab and create new contacts, search for existing contacts, select a recently viewed contact, and change contact information.
- In BillingCenter, you can click **Search** > **Contacts** to find contacts You can also add, edit, and delete centrally managed contacts on **Contacts** screens of the **Account** and **Policy** tabs.

The Guidewire core applications send information to ContactManager across the network by using plugins to call ContactManager's web services. ContactManager itself has plugins that it uses to call the core applications' web services and to perform internal tasks.

Additionally, each core application implements a `ContactAPI` web service that ContactManager can use to send contact changes to the application.

To change or add properties to contact-related entities, you can extend the data model in both the Guidewire core applications and ContactManager. Alternatively, you can add tags for any contact, or you can specify services to use with vendor contacts. Whichever approach you take, you must ensure that information flows correctly between the applications.

The following areas of Guidewire core application functionality are not present in ContactManager:

- Notes
- Administrative groups
- Activities
- Assignment

There are no ContactManager entities, plugins, or APIs associated with these objects and features.

ContactManager entities

ContactManager has entities that are similar to entities in ClaimCenter, PolicyCenter, and BillingCenter. For example, the ClaimCenter, PolicyCenter, and BillingCenter `Contact` entity has a corresponding ContactManager `ABContact` entity.

`ABContact` is an important entity for ContactManager. It is the primary entity that ContactManager uses to manage contacts. The `ABContact` entity has the subtypes `ABCompany`, `ABPerson`, and `ABPlace`, and each of those three entities has specialized versions. For example, `ABCompany` has a vendor subtype called `ABCompanyVendor`.

This subtype hierarchy parallels the `Contact` subtype hierarchy in Guidewire core applications.

Note: ContactManager also has entities like `User` and `Group` to enable you to add employees who can access ContactManager. These entities are not involved in integrating with core applications.

Some important ContactManager entities are listed in the following table:

Entity or class	Description
`ABContact`	The ContactManager version of the `Contact` entity that stores name, phone number, address, and so on for the contact.
`ABContactContact`	The ContactManager version of the `ContactContact` entity that connects contacts that have a relationship.
`ABContactTag`	An entity connecting an `ABContact` entity to a tag, a typecode in the `ContactTagType` typelist. `ABContact` has an array of `ABContactTag` entities.
`ABContactSpecialistService`	An entity connecting an `ABContact` to a service. The `ABContact` entity has an array reference to this entity.
`Address`	An address associated with a contact.
`Document`	A document associated with a contact.
`History`	An entity that captures the history of actions performed on the contact, such as when it was created and what changes were made to the contact data. Additionally, this object can record which user and application performed the actions.
`ReviewSummary`	An entity that captures a summary of a review's information, passed from ClaimCenter.

See also

- "ABContact data model" on page 109
- "Contact data model" on page 111
- "Overview of vendor documents" on page 177
- "ContactManager service provider performance review data model" on page 246

ContactManager link IDs and comparison to other IDs

ContactManager entities that implement the `ABLinkable` delegate, such as `ABContactTag`, `ABContactAddress`, and `ABContact` and its subentities, have a `LinkID` property. This property uniquely identifies an `ABLinkable` entity instance for integration use, such as with Guidewire core applications. It is similar to the `PublicID` property in Guidewire core applications, which those applications can use as a primary key value for entities in external systems. However, because ContactManager is the system of record for contacts across the core applications, ContactManager must ensure that a contact or address be uniquely identifiable across all Guidewire applications. Therefore, unlike a `PublicID`, a `LinkID` cannot be changed.

If the core application does not specify an `External_UniqueID` when it calls ContactManager to create a new contact, ContactManager creates the `LinkID` for a new entity. If the core application does specify the unique ID when it sends a create request, ContactManager populates the `LinkID` with the `External_UniqueID` specified in the `XmlBackedInstance` data.

ContactManager passes the `LinkID` back to the core applications. A core application can use the return value either to identify the local version of the contact already created or to populate the equivalent `AddressBookUID` property.

> **Note:** The unique ID passed to ContactManager for any `ABLinkable` entity in the `ABContact` graph might already exist on an entity of that type. If ContactManager detects that one of these entities, including retired entities, already use this ID, ContactManager throws a `DuplicateKeyException` for the call to `createContact`. In this case, the contact is not created in ContactManager. It is up to the calling application to recover from this state, either by providing a new unique ID or allowing ContactManager to create a unique ID.

The `AddressBookUID` property is the core application version of the ContactManager `LinkID` property. `AddressBookUID` properties and `LinkID` properties are mapped between core applications and ContactManager in the `ContactMapper` classes.

The following table compares the various types of IDs related to contacts:

Type of ID	Description
`LinkID`	The name for the internal ID that ContactManager uses for each `ABContact` entity and subentity
`PublicID`	The standard Guidewire public record ID that can be changed as needed.
`AddressBookUID`	In Guidewire core applications, this property of the `Contact` entity is the internal ID for a contact entity that is stored in ContactManager. If you are using a Guidewire core application and ContactManager together, an address book UID and a link ID have the same value. If you integrate with a different external contact management system, the address book UID has the same value as the internal ID of an object in that external application.

See also

- "ContactMapper class" on page 313
- For specifics on how `External_UniqueID` and `LinkID` are handled, see the `createContact` method in the table at "ABContactAPI methods" on page 303.
- For ContactManager examples of mapping between `AddressBookUID` and `LinkID`, see "Mapping fields of a ContactManager contact" on page 315.
- For examples of mapping between `AddressBookUID` and `LinkID`, see "Mapping fields of a core application contact" on page 318.
- For general information on `PublicID`, see the *Integration Guide*.

ContactManager web services

Web services provide a language-neutral, platform-neutral mechanism for invoking actions or requesting data from other applications across a network. Guidewire applications provide web services that are intended to be used both by other Guidewire applications and by external applications. You can also write your own web services.

See also

- For general, overview information on Guidewire web services, see the *Integration Guide*.

Web services provided by ContactManager

ContactManager provides the following web services. The first one, `ABContactAPI`, is the primary web service used by Guidewire core applications to communicate with ContactManager. The second one, `ABVendorEvaluationAPI`, supports vendor evaluations. The remaining web services are standard services available in all Guidewire applications.

For a complete list of base configuration web services, see the *Integration Guide*

Web Service	Description	More information
`ABContactAPI`	The primary web service used by Guidewire core applications to search for, create, update, and delete contacts. This web service's WSDL is retrieved by every Guidewire core application to make it available to the application plugin that communicates with ContactManager. The class package is `gw.webservice.ab.ab1000.abcontactapi`.	"ABContactAPI web service" on page 303
`ABVendorEvaluationAPI`	A web service that enables ClaimCenter to send and receive review summary information. The class package is `gw.webservice.ab.ab1000.abvendorevaluationapi`.	For more information on vendor contact performance reviews, see "ClaimCenter service provider performance reviews" on page 237.
`ImportToolsAPI`	Imports administrative data from an XML file. Use this web service only with administrative database tables, such as entities of type `User`. The system does not perform complete data validation tests on any other type of imported data.	See the *Integration Guide* for information on importing administrative data.
`LoginAPI`	WS-I authentication happens with each API call. However, if you want to explicitly test specific authentication credentials in your web service client code, ContactManager publishes the built-in `LoginAPI` 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. You can also use `LoginAPI` to purposely leave a user session open for logging purposes.	See the *Integration Guide* for more information on login authentication.
`MaintenanceToolsAPI`	Provides a set of tools that start and manage various background processes. The methods of this web service are available only when the server run level is `maintenance` or higher.	See the *Integration Guide* for more information on the maintenance tools web service.
`MessagingToolsAPI`	Provides messaging methods for managing the messaging system remotely for recovery from message acknowledgment errors. The methods of this web service are available only when the server run level is `multiuser`.	See the *Integration Guide* for more information on the messaging tools web service.

Web Service	Description	More information
`PersonalDataDestructionAPI`	Enables an external application to request: • Destruction of a contact's data by `AddressBookUID` or by `PublicID`. • Destruction of a user by user name.	"Data destruction web service" on page 217
`ProfilerAPI`	Sends information to the built-in system profiler.	See the *Integration Guide* for more information on the profiling web service.
`SystemToolsAPI`	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. Some uses of this web service include: • Getting the version of the server, including application version and schema version • Checking the consistency of the underlying physical database • Getting and setting the run level	See the *Integration Guide* for more information on the system tools web service.
`TableImportAPI`	Provides a table-based import interface for high volume data import. This web service is typically used for large-scale data conversions, particularly for migrating records from a legacy system into ContactManager prior to bringing ContactManager into production.	See the *Integration Guide* for more information on the table import web service.
`TypelistToolsAPI`	Provides tools for getting the list of valid typecodes for a typelist and for mapping between ContactManager internal codes and codes in external systems.	See the *Integration Guide* for more information on mapping typecodes and external system codes.
`WorkflowAPI`	Performs various actions on a workflow, such as suspending and resuming workflows and invoking workflow triggers.	See the *Integration Guide* for more information on the workflow web service.
`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 the *Integration Guide* for more information on the zone data import web service.

Support classes for ContactManager web services

ContactManager provides a number of Gosu classes that are used by its web services. The following classes are the most common ones that you might edit to support your extensions to ContactManager, such as support for new search criteria. The class path for most of these classes is `gw.webservice.ab.ab1000.abcontactapi`.

Gosu Class	Description
`ABContactAPIDocumentSearchCriteria`	A Gosu class that specifies the contact document search criteria that the Guidewire core applications can use. The class package is `gw.webservice.ab.ab1000.abcontactapi`. This class is used by the web service `ABContactAPI`.

Gosu Class	Description
	Do not edit this class to add new search criteria for contact documents. ContactManager cannot make use of extensions to the contact document search classes to support custom document search criteria. See also "ContactManager web service for retrieving contact documents" on page 190.
`ABContactAPIFindDuplicatesResult`	A Gosu class used to define duplicate `ABContact` entities found by the `ABContactAPI.findDuplicates` method. You can edit this class to add your contact extensions so they can be returned by the `findDuplicates` method.
`ABContactAPIPendingContactChange`	An instance of this Gosu class is the second parameter to the method `ABContactAPI.createContactPendingApproval`. This class contains the metadata for the change that ContactManager will send back to the calling application if the change is rejected. The calling application can then notify the user of the rejection. See also "Review pending changes to contacts" on page 278.
`ABContactAPIProximitySearchParameters`	A Gosu class that defines the search parameters for a proximity search. You can modify this class. See also "Geocoding and proximity search for vendor contacts" on page 73.
`ABContactAPISearchCriteria`	A Gosu class that specifies the contact search criteria that the Guidewire core applications can use. The class package is `gw.webservice.ab.ab1000.abcontactapi`. This class is used by the web service `ABContactAPI`. You can edit this class to add new search criteria for contacts, as described in "Add search support in ContactManager for Guidewire core applications" on page 60.
`ABContactAPISearchResult`	A Gosu class that specifies the fields included in the contact search results that ContactManager sends back to the Guidewire core application. The class package is `gw.webservice.ab.ab1000.abcontactapi`. This class is used by the web service `ABContactAPI`. See "Add search support in ContactManager for Guidewire core applications" on page 60.
`ABContactAPISpecialistService`	A Gosu class that defines a `SpecialistSevice` entity to be returned by the method `ABContactAPI.getSpecialistService`. You can edit this class if you have made extensions to how `Services` work.
`ABVendorEvaluationAPIReviewSummary`	A Gosu class that enables ClaimCenter to update the review summary of a vendor evaluation with additional, new information. The class package is `gw.webservice.ab.ab1000.abvendorevaluationapi`. This class supports the web service `ABVendorEvaluationAPI`.
`AddressInfo`	A Gosu class that defines `Address` objects to be passed in `ABContactAPI` method calls. If you have extended the `Address` entity, you can edit this class to add your extensions, such as new address fields.

See also

- "ABContactAPI methods" on page 303

ABContactAPI web service

The `ABContactAPI` web service is the primary web service used by Guidewire core applications to search for, create, update, and delete contacts. This web service also supports specifying services in search criteria used in searches for contacts. This web service's WSDL is retrieved by every Guidewire core application to make it available to the core application plugin that communicates with ContactManager.

- In ContactManager, `ABContactAPI` is in the class path `gw.webservice.ab.ab1000.abcontactapi`.
- In ClaimCenter, PolicyCenter, and BillingCenter, the WSDL for `ABContactAPI` is in the same web service collection for all three applications. Open Guidewire Studio™ and then open the **Project** window. Navigate to **configuration** > **gsrc** and then to `wsi/remote/gw/webservice/ab/ab1000.wsc`. Double-click `ab1000.wsc` to open the editor.

You can use this web service to load data from other data sources, to query contacts, to query services, or to update contacts. For example, you could write a command-prompt tool that enables an external system to add new contacts based on new business data from another part of the company.

ABContactAPI and typelists

The `ABContactAPI` web service exposes the typecodes in contact-related typelists as strings, which simplifies the code needed to access these typecodes from external or Guidewire core applications. For example, states are defined in the typelist `State`. The following annotation in `ABContactAPI` exposes this typelist as a string:

```
@WsiExposeEnumAsString(typekey.State)
```

Exposing this typelist as a `string` makes it possible for a Guidewire core application to do a simple assignment like the following in the ClaimCenter `ContactSearchMapper` class:

```
searchCriteriaInfo.ServiceState = searchCriteria.ServiceState.Code
```

ABContactAPI methods

The ContactManager `ABContactAPI` web service provides the following methods.

Method	Parameters	Description
`createContact`	`abContactXML` – Contact information in `XmlBackedInstance` format.	Creates a new contact and returns an `AddressBookUIDContainer` containing IDs for the `Contact` and child objects. Contact information is expected to be in `XmlBackedInstance` format. If the `abContactXML` parameter includes an `External_UniqueID` field and that field has a value, ContactManager uses this value to populate the `LinkID` field. In this manner, the calling application specifies the ultimate value of the `LinkID`. If the `External_UniqueID` field is missing or is `null`, ContactManager generates its own `LinkID` and passes it back to the calling application. ContactManager determines if the unique ID passed to ContactManager for any `ABLinkable` entity in the `ABContact` graph already exists on an entity of that type, including retired entities. If so, ContactManager throws a `DuplicateKeyException` for the call to `createContact`. In this case, the contact is not created in ContactManager. It is up to the calling application to recover from this state, either by providing a new unique ID or allowing ContactManager to create a unique ID.

Method	Parameters	Description
		Calls ValidateABContactCreationPlugin to ensure that there is enough data to create the contact. If not, the method returns RequiredFieldException to the calling application. If there is enough data, the method creates a new ABContact of the subtype specified by abContactXML. The method then populates the new entity with data it retrieves by calling ContactIntegrationMapper.populateABContactFromXML.
createContactPendingApproval	abContactXML – Contact information in XmlBackedInstance format. updateContext – User, entity, and application information sent by core application. An instance of ABContactAPIPendingContactChange.	Creates a new contact of the type specified and sets its status to PENDING_APPROVAL. Returns an AddressBookUIDContainer containing IDs for the Contact and child objects and the update context and transaction ID. This method is called by the core application because the core application user creating the contact does not have permission to create a contact. Contact information is expected to be in XmlBackedInstance format. Calls ValidateABContactCreationPlugin to ensure that there is enough data to create the contact. If not, the method returns RequiredFieldException to the calling application. If there is enough data and no other exceptions are thrown, the method creates a new ABContact of the subtype specified by abContactXML. The method then populates the new entity with data it retrieves by calling ContactIntegrationMapper.populateABContactFromXML and sets its status to PENDING_APPROVAL.
findDuplicates	abContactXML – XmlBackedInstance that contains the contact data for which duplicates are being found. abContactAPISearchSpec – Specifies how the search is to be returned.	Finds contacts that match the specified contact. Returns an ABContactAPIFindDuplicatesResultContainer object containing summary information about each match.
getReplacementAddress	addressLinkID – linkID of an address that has been replaced because of a merge	Gets the address that has replaced the address passed in the parameter. An address can be replaced as a part of a merge. Returns the linkID of the address that replaces the address denoted by the argument.
getReplacementContact	contactLinkID – linkID of a contact that has been replaced because of a merge	Gets the contact that has replaced the contact passed in the parameter. A contact can be replaced as a result of a merge. Returns the linkID of the contact that replaces the contact denoted by the argument, or null if the contact has not been replaced by another contact.
getSpecialistServices	contactLinkID – linkID of the contact that has specialist services	Gets the specialist services associated with the contact passed in the parameter. Returns an array of ABContactAPISpecialistService objects, or null if the contact has no specialist services.

Method	Parameters	Description
`removeContact`	`abContactXML` – `XmlBackedInstance` that contains the `linkID` of the contact to be removed and miscellaneous information.	Removes the contact with the matching `linkID` if the contact is not being used by any remote application. If a remote application is not running when this method runs, the contact is assumed to be in use and is not removed. Returns a `Boolean` indicating whether the contact was successfully removed.
`retrieveContact`	`linkID` – ID uniquely associated with this contact. In a core application, this value is the `AddressBookUID`.	Retrieves information about the contact uniquely specified by the `linkID`. Returns contact information in `XmlBackedInstance` format.
`retrieveDocumentsforContact`	`contactLinkID` – ID uniquely associated with this contact. In a core application, this value is the `AddressBookUID`. `abContactAPIDocumentSearchCriteria` – The criteria for the search. Required and cannot be `null`. `abContactAPIDocumentSearchSpec` – Specifications for how the results are to be returned.	Retrieves information about the contact's documents. Returns document information in `ABContactAPIDocumentSearchResultContainer` format.
`retrieveRelatedContacts`	`linkID` – ID uniquely associated with this contact. This parameter must not be null. `relationshipTypes` – Array of `ContactBidiRel` relationship types of the related contacts to return information on. This parameter must not be null.	Retrieves information about the contact's related contacts. Returns a `RelatedContactInfoContainer` containing information about the contact's related contacts.
`searchContact`	`abContactAPISearchCriteria` – Criteria for the search. This parameter must not be null. `abContactAPISearchSpec` – Specifies how the	Searches for all contacts that match the given search criteria. Return an `ABContactAPISearchResultContainer` object containing the search results.

Method	Parameters	Description
	results are to be returned.	
`updateContact`	`abContactXML` – Contact information in `XmlBackedInstance` format.	Updates an existing contact and returns an `AddressBookUIDContainer` object containing IDs for the `Contact` and child objects. Contact information is expected to be in `XmlBackedInstance` format. An existing `ABContact` is selected based on the `abContactXML.LinkID`. If none is found, the method throws `BadIdentifierException`. Then, `origValue` attributes for all fields and foreign keys being updated are checked against those same values in the selected `ABContact`. If discrepancies are found, the method throws `EntityStateException`. The purpose is to manage versioning of contact-related changes across multiple client applications. If no exceptions have yet been thrown, the method calls `ContactIntegrationMapper.populateABContactFromXML` to update the data for local entities from `abContactXML`.
`updateContactPendingApproval`	`abContactXML` – Contact information in `XmlBackedInstance` format. `updateContext` – User, entity, and application information sent by core application.	Submits for approval an update to an existing contact that is pending until approved. The core application calling this method has determined that the user updating the contact does not have permission to do so. Contact information is expected to be in `XmlBackedInstance` format. If no existing `ABContact` can be found based on the `abContactXML.LinkID`, the method throws `BadIdentifierException`. If an `ABContact` entity is found with this `LinkID`, this method creates a `PendingUpdate` entity for the contact.
`validateCreateContact`	`abContactXML` – Contact information in `XmlBackedInstance` format.	Determines if the specified contact can be created. Calls `ValidateABContactCreationPlugin.validateCanCreate` to see if `abContactXML` has the minimum data required to create an `ABContact` entity. Returns an `ABContactAPIValidateCreateContactResult` object indicating whether validation passed and, if validation failed, an error message. For more information, see "ValidateABContactCreationPlugin plugin interface" on page 337.

Retrieving contacts and their relationships

If you want to integrate ContactManager with an application other than a Guidewire core application, you can write custom web services to do so. For example, to retrieve properties from contact records, you can write a custom web service (SOAP API) that extracts the subset of entity contact information that you want. Using an `AddressBookUID`, the API could extract properties from an `ABContact` record and return it from ContactManager to the SOAP client.

For typical cases, return only certain properties or subobjects, such as the properties needed to display or edit the record. Design your integration point accordingly to return only the necessary data, which reduces bandwidth and server resources.

In some cases, you might want to create new entities or classes to encapsulate your custom data.

If you are retrieving contact records, you can optionally retrieve related contacts, as you can do with ClaimCenter. Related contacts are referred to generally in ClaimCenter and ContactManager as *relationships*. For example, to retrieve a contact's parent or guardian or employer contact information, specify one of these types of relationships. Finally, design web services for each integration point to return that information if it exists.

To use relationship retrieval, you must understand the difference between *source* and *target* relationships. These terms are ways of using the relationship identification codes for relationships such as *parent/guardian* and *employer*, but also specify the directionality of the relationship.

Relationship codes in Gosu are defined in the `ContactRel` typelist in Guidewire Studio™. For example, this Gosu enumeration includes relationship code `TC_employer` as a reference to the employer typecode in the typelist. Suppose you want the record for someone named John Smith. If you want to return John Smith's employers, specify `TC_employer` as a *target relationship*. If you want to return John Smith's employees, specify `TC_employer` as a *source relationship*.

A *target relationship* describes the relationship if the `ContactRel` typecode description fits it into the sentence with the following structure:

"I want to retrieve a contact and the ______ of the contact, if any is found."

For example, for the employee relationship (`employer`), the sentence would be:

"I want to retrieve a contact and the **employer** of the contact, if any is found."

If you want your request to specify the opposite of that relationship, such as employee instead of employer, specify the relationship as a source relationship.

While retrieving a contact record, you might want both directions of relationships. For example, you might want to simultaneously retrieve all employees of a company and the company's primary contact. Use the `ContactBidiRel.EMPLOYER` source relationship and the `ContactBidiRel.PRIMARYCONTACT` target relationship.

IMPORTANT: If extracting contact information by using custom SOAP APIs, write your API to return only the required properties and the required relationships. If you return an entire `ABContact` or unneeded related contacts, large amounts of data created by SOAP serialization can trigger server or client memory problems.

ABVendorEvaluationAPI web service

This web service provides methods used by ClaimCenter to communicate with ContactManager about service provider performance review information.

Service provider performance reviews are described at "ClaimCenter service provider performance reviews" on page 237.

This web service provides the following methods:

Method	Parameters	Description
`addNewReviewSummary`	`reviewInfo` – The new review summary, an `ABVendorEvaluationAPIReviewSummary` object. The review summary sent to ContactManager must not have a `LinkID` set.	Adds a new review summary based on the `ABVendorEvaluationAPIReviewSummary` object passed in. Returns the created review summary, an `ABVendorEvaluationAPIReviewSummary` object, complete with `LinkID`.
`deleteReviewSummary`	`linkID` – The `LinkID` of the review summary as a `String`.	Deletes the review summary that has the passed-in `LinkID`. Returns a `Boolean` indicating if the review summary was successfully deleted.
`updateReviewScoresForContact`	`linkID` – The `LinkID` of the associated vendor contact, as a `String`.	Updates the scores for all the reviews for the contact whose `LinkID` is passed in.

Method	Parameters	Description
		Returns an `int` indicating the current score.

ContactManager messaging events

ContactManager sends messages to integrated Guidewire core applications after changes to certain entities, such as adding or deleting a contact or changing data for a contact. These changes trigger events, which trigger code that sends messages. For example, a user of a core application changes the address of a contact stored in ContactManager. ContactManager receives that change and updates the contact, triggering an event, `ABContactChanged`. This event triggers event message rules that cause ContactManager to send the contact change to the integrated core applications.

Additionally, if documents are being handled asynchronously, there are event messages that handle communication with the document management server. These messages are triggered by `Document` events for adding, changing, removing, and storing a document.

You can see the messaging events that are generated for each entity in the *Data Dictionary*. To build the data dictionary for your system, open a command prompt and navigate to the ContactManager installation folder, and then enter the command `gwb genDataDictionary`. The command builds the security and data dictionaries. It saves the *Data Dictionary* in `ContactManager/build/dictionary/data/index.html`.

ContactManager messaging events by entity

The following table lists the messaging events for each ContactManager entity for which ContactManager sends messages.

Entity	Events	Description

`ABContact`	`ABContactAdded`
	`ABContactChanged`
	`ABContactRemoved`
		Contact entities exist only as subtypes of `ABContact`, such as `ABPerson`. Those subtypes generate standard A/C/R events for root entity `ABContact`.

	`ABContactPendingChangeRejected`	An `ABContact` pending change request has been rejected.
	`ABContactResync`	An `ABContact` has been resynchronized. Re-send all related messages to external systems as appropriate. See also • For information on message ordering and multi-threaded sending, see the *Integration Guide*. • For information on resynchronizing messages for a primary object", see the *Integration Guide*

`ABAdjudicator`	`ABContactAdded`
	`ABContactChanged`
	`ABContactRemoved`
	`ABContactPendingChangeRejected`
	`ABContactResync`
		Standard A/C/R events for root entity `ABContact`.

`ABAttorney`	`ABContactAdded`
	`ABContactChanged`
	`ABContactRemoved`
	`ABContactPendingChangeRejected`
	`ABContactResync`
		Standard A/C/R events for root entity `ABContact`.

`ABAutoRepairShop`	`ABContactAdded` `ABContactChanged`	Standard A/C/R events for root entity `ABContact`.

Entity	Events	Description
	ABContactRemoved ABContactPendingChangeRejected ABContactResync	
ABAutoTowingAgcy	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABCompany	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABCompanyVendor	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABDoctor	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABLawFirm	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABLegalVenue	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABMedicalCareOrg	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABPerson	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABPersonVendor	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABPlace	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.
ABPolicyCompany	ABContactAdded ABContactChanged ABContactRemoved ABContactPendingChangeRejected ABContactResync	Standard A/C/R events for root entity ABContact.

Entity	Events	Description
`ABPolicyPerson`	`ABContactAdded` `ABContactChanged` `ABContactRemoved` `ABContactPendingChangeRejected` `ABContactResync`	Standard A/C/R events for root entity `ABContact`.
`ABUserContact`	`ABContactAdded` `ABContactChanged` `ABContactRemoved` `ABContactPendingChangeRejected` `ABContactResync`	Standard A/C/R events for root entity `ABContact`.
`Document`	`DocumentAdded` `DocumentChanged` `DocumentRemoved` `DocumentStore`	Standard A/C/R events for entity `Document`. **Note:** The EventFired rules available in the base configuration for documents support only asynchronous operation with a document management system. The default mode of document storage is synchronous.

See also

- For information on ContactManager event messaging rules, see “EventMessage rule set category” on page 207.
- “Enabling asynchronous event fired document messaging” on page 189

ABContact message safe ordering

ContactManager supports safe ordering of `Message` entity instances associated with `ABContact` entity instances. Messages associated with an entity instance are sent ordered by `ABContact` instance for each messaging destination. Safe ordering allows only one message per contact-destination pair to be *in flight*—sent but not acknowledged at any given time. Messages of this type are referred to as *safe-ordered messages*.

If an event generates two safe-ordered messages related to one entity instance, ContactManager sends the first message immediately but does not send the second message. After ContactManager receives an acknowledgment from the destination about the first message, it sends the second message.

Because ContactManager manages events for many entity instances, a Guidewire core application can have many messages in flight at a time, even multiple messages to one destination. However, there can be only one message in flight for each contact-destination pair.

See also

- For more information about safe ordering as it relates to claims, see “Sending safe-ordered messages” in the *Integration Guide*.

ABContact message resynchronization

ContactManager supports resynchronizing a contact—the `ABContact` entity—with an external system. If an `ABContact` entity resynchronizes, a destination could listen for the `ABContactResync` event. If that event triggers, the Event Fired rules that process it could re-send important messages to external systems for this entity to synchronize with the external system.

See also

For more information about message resynchronization as it relates to claims, see the *Integration Guide*.

ABClientAPI interface

ContactManager requires that each Guidewire core application implement the `ABClientAPI` interface and expose it as a web service.

- ClaimCenter implements this interface in `gw.webservice.cc.cc1000.contact.ContactAPI`.
- PolicyCenter implements this interface in `gw.webservice.pc.pc1000.contact.ContactAPI`.
- BillingCenter implements this interface in `gw.webservice.bc.bc1000.contact.ContactAPI`.

ContactManager then calls the web service methods in each application when it needs to broadcast changes in contacts to the applications. Each Guidewire core application can determine what to do in response to the call from ContactManager.

The current version of this ContactManager interface is in the package `gw.webservice.contactapi.ab1000.ABClientAPI`.

See also

- For information on core application implementations of `ContactAPI`, see the contact web service topics in the *Integration Guide*.

Method signatures of the ABClientAPI interface

The `ABClientAPI` interface provides the following method signatures.

Method	Parameters	Description
`isContactDeletable`	`addressBookUID` – The address book unique ID of the contact. For ContactManager, this is the `linkID`.	Return `true` either if the contact associated with the `AddressBookUID` can be deleted or if no contact is associated with `AddressBookUID`. Return `false` if the contact cannot be deleted.
`mergeContacts`	`keptContactABUID` – The `addressBookUID` of the contact to be kept. `deletedContactABUID` – The `addressBookUID` of the contact to be deleted.	Merge two contacts that were merged in ContactManager. ContactManager does not send contact data with this method call. It is up to the Guidewire core application to retrieve the data for the kept contact.
`pendingCreateApproved`	`context` – An `ABClientAPIPendingChangeContext` object providing information on the user requesting this change.	Notifies the client system that a pending contact creation it submitted has been approved by ContactManager. The client application can use the information in the context parameter to inform the user who submitted the request that the contact was created. Additionally, the core application can update the sync status of the contact and post an appropriate message.
`pendingUpdateApproved`	`context` – An `ABClientAPIPendingChangeContext` object providing information on the user requesting this change.	Notifies the client system that a pending update it submitted has been approved by ContactManager. The client application can use the information in the context parameter to inform the user who submitted the change that the change was approved. Additionally, the core application can update the sync status of the contact and post an appropriate message.
`pendingCreateRejected`	`context` – An `ABClientAPIPendingChangeContext` object providing information on the user requesting this change.	Notifies the client system that a pending contact creation it submitted has been rejected by ContactManager. The client application can use the information in the context parameter to inform the user who submitted the contact to be created that the creation was rejected. Additionally, the core application can update the sync status of the contact and post an appropriate message.
`pendingUpdateRejected`	`context` – An `ABClientAPIPendingChangeContext` object	Notifies the client system that a pending update it submitted has been rejected by ContactManager.

Method	Parameters	Description
	providing information on the user requesting this change.	The client application can use the information in the context parameter to inform the user who submitted the change that the change was rejected. Additionally, the core application can update the sync status of the contact and post an appropriate message.
`removeContact`	`addressBookUID` – Unique ID of a contact that has been removed in ContactManager.	The contact with this `AddressBookUID` has been removed in ContactManager. This call does not require that the contact be removed by the Guidewire core application. For example, if the contact is in use by an account, PolicyCenter does not retire it.
`updateContact`	`contactXML` – `XmlBackedInstance` that contains the `addressBookUID` of the contact and the updates to be made.	Data for this contact has changed in ContactManager. Update the contact by using the data in `contactXML`.

Deleting contacts in ClaimCenter

ClaimCenter, unlike the other core applications, can have multiple local instances of any contact, an instance for each claim. Therefore, ClaimCenter might have a lot of work to do in deleting a contact. ClaimCenter has to check every local instance of the contact. For each local instance, ClaimCenter must ensure that there are no entities that reference the instance that would prevent it from being deleted. For example, if there are any claims that have a contact as an involved party, the contact cannot be deleted.

See also

- "Synchronizing ClaimCenter and ContactManager contacts" on page 201
- For information on `Contact` web service APIs, see the *Integration Guide*.

ClaimCenter ContactAPI web service methods for removing contacts

The ClaimCenter implementation of `ABClientAPI` is the web service `gw.webservice.cc.cc1000.contact.ContactAPI`. This web service provides implementations of all the methods, two of which are used by ContactManager in requesting that ClaimCenter delete a contact:

`isContactDeletable(addressBookUID : String)`

If there are fewer than ten local instances of the contact, this method finds all the contacts linked to the `addressBookUID`. It then calls the `ContactRetireHelper.retireContact(Contact)` method on each one and returns `true` if they can all be successfully retired.

If any contact cannot be successfully retired, the method returns `false` to indicate that the contact cannot be deleted. If there are more than ten contacts, the method returns `false` and then generates work items to check all the local contacts as part of the work queue. The work queue calls `ContactRetireHelper.retireContact` to retire each contact instance.

`removeContact(addressBookUID : String)`

Calls `ContactRetireHelper.retireContact` for each ClaimCenter contact found with the `addressBookUID`.

ClaimCenter classes for removing contacts

ClaimCenter provides two classes you can use in configuring how ClaimCenter handles any data model extensions you have added that might affect the removal, or *retiring*, of contacts.:

`IRetireContactPlugin`

This plugin interface has a method to return a set of safe properties. These properties represent foreign key links to a contact that can be retired without any further checking when a contact is retired. You can add to the list any extension foreign keys that you deem safe. You can use the implementation of this interface, `gw.plugin.contact.RetireContactPlugin`, to add properties to the safe list.

`ContactRetireBean`

This interface can be implemented by entities that have an unsafe foreign key link to a contact. The implementation can determine if an entity instance will prevent a contact instance from being retired. If not, the contact can be retired, and then the implementation can also determine if any other contacts or entities can be retired along with the contact. See the class `gw.api.contact.ContactContactRetireBeanImpl` for an example of checking to see if a `ContactContact`, and the associated `Contact`, can be retired.

Implementation details for retiring contacts in ClaimCenter

ClaimCenter provides the `gw.api.contact.ContactRetireHelper` class to determine if a contact can be retired. This class provides two methods used in retiring contacts:

`public static boolean retireContact(Contact contact)`

Attempts to retire the passed in Contact, returning `true` if it was successful in doing so. This method is called by the `ContactRetire` work queue.

`public static boolean computeCanRetireContact(Contact contact, ContactRetireContext retireContext)`

Available to be used when implementing the `ContactRetireBean` interface to see if a contact can be retired. You might call this method if, while using `ContactRetireBean.computeCanRetireBeanForContactProperty`, you encounter another contact and need to retire it as well.

The `retireContact` method splits the foreign key references for the `Contact` into two components, those properties deemed safe from the `IRetireContactPlugin` implementation and the other properties. The safe properties for the contact do not block the retirement of the contact. Additionally, if there is a bean connected to the contact through this property, it is retired along with the contact.

In the base implementation, the safe references are:

Entity	Array field
`ContactAddress`	`Contact`
`ContactCategoryScore`	`Contact`
`ContactTag`	`Contact`
`EFTData`	`Contact`
`OfficialID`	`Contact`
`Review`	`Contact`

The other properties by default will block the retirement of the contact unless they implement the `ContactRetireBean` interface. If an entity implements that interface, the `computeCanRetireBeanForContactProperty` method of the interface can be called to see if the entity can be retired.

In the base configuration, the only entity requiring this interface to be implemented is `ContactContact`. This entity has the dual role of being an array on `Contact` and also having a foreign key to a contact. `ContactContact` represents a join table to two contacts, so the other contact is a foreign key reference. The class that implements the `ContactRetireBean` interface is `gw.api.contact.ContactContactRetireBeanImpl`.

ContactMapper class

Guidewire core applications send contact information to ContactManager by using an XML SOAP object. They use the `ContactMapper` class both to generate the SOAP object and to interpret it when they receive an XML SOAP object from ContactManager.

ContactManager and the Guidewire core applications each have their own version of the `ContactMapper` class. This class enables the applications to convert `Contact` entities and subentities to XML and send them to ContactManager. The class also enables Guidewire core applications to receive XML representations of the entities from ContactManager and convert them to `Contact` entities.

ContactManager also uses its version of this class to generate XML to send to the applications and to interpret the XML it receives from the applications. The class file in each core application and in ContactManager explicitly specifies every field of the entities to send or receive and how to handle them in that application.

In the base configuration of each Guidewire core application, the `ContactMapper` class is configured to handle all the `Contact` entities and subentities. The class handles the entities and fields of those entities that are to be sent to and received from ContactManager.

In the base configuration, there are differences in entity names, typecodes, and contact entity field names between the core applications and ContactManager. Each Guidewire core application has its own domain namespace, but the ContactManager web services require the ContactManager domain namespace. The core applications use ContactManager web services to communicate with ContactManager. Therefore, when there are differences in names of fields, data types, and typecodes between the core application and ContactManager, each core application is responsible for doing the name mapping.

- To map differing entity field names, a core application uses a method on the `ContactMapper` class.
- To map names of data types and typecodes, a core application uses the `ContactMapper` class in conjunction with a mapping class that specifies these name mappings. For example, a `Contact` in the core application maps to an `ABContact` in ContactManager.

The ContactManager `ContactMapper` class handles only `ABContact` entities and subentities. Because ContactManager must be able to communicate with all the Guidewire core applications, its `ContactMapper` class must contain all fields of each type of contact that each core application needs. In a Guidewire core application, you can map just the contact fields that you want to send to ContactManager and receive from it.

The `ContactMapper` class creates a SOAP object called `XMLBackedInstance` to pass the data between a Guidewire core application and ContactManager. This object contains the fields and data of contacts and their related arrays and foreign keys that are defined in the `ContactMapper` class. `XmlBackedInstance` has a representation both as an object and as an XML string.

If you are working only with the entities supplied by Guidewire in the base configuration, you do not need to make any changes to the `ContactMapper` class.

You can extend your Guidewire core application's `Contact` data model. You make the extension in ContactManager as well, and then map the entities to each other in `ContactMapper` in both the Guidewire core application and in ContactManager. Additionally, you use the application's mapping class to map differing entity names and any different typecodes to and from ContactManager.

See also

- "Core application mapping" on page 117.
- For information on the contact data model in the base configuration, see "Overview of contact entities" on page 109.
- "Extending the contact data model" on page 109.

ContactManager ContactMapper class

In ContactManager, you use the class `gw.contactmapper.ab1000.ContactMapper` to map `ABContact` entities and subentities. The class maps them to an XML object to send to Guidewire core applications and maps them from the XML objects received from Guidewire core applications. The class in the base ContactManager configuration works only with `ABContact` entities and subentities. ContactManager leaves it to the applications to map `ABContact` entities and subentities to and from `Contact` entities and subentities.

You can access this class in Guidewire Studio™ for ContactManager in the **Project** window. Navigate to **configuration** > **gsrc** and then to `gw.contactmapper.ab1000.ContactMapper`.

The following methods map the fields, foreign keys, and arrays of the ContactManager `ABContact` entities and subentities:

`fieldMapping`
Maps fields of an entity

`fkMapping`
Maps foreign keys of an entity

`arrayMapping`
Maps array references of an entity

Mapping fields of a ContactManager contact

The method `fieldMapping(`*`Entity#Field`*`)` by default maps `ABContact` entity and subentity fields in both directions, both to and from a Guidewire core application.

Note: Parameters to methods in `ContactMapper` use Gosu *feature literals* syntax to statically refer to an entity's fields. See the *Gosu Reference Guide*.

The mapping directions are:

To a Guidewire core application
From a ContactManager entity to an XML object to be sent to a Guidewire core application

From a Guidewire core application
From an XML object sent by a core application to a ContactManager entity

You also use this method to map fields of an entity that the contact references with either an array reference or a foreign key. To map a foreign key or array reference itself, you use different methods.

Each complete method call must be followed by a comma unless it is the last method call in the block.

Specifying a Single Direction

You specify a single direction by using `.withMappingDirection`, as follows:

`.withMappingDirection(TO_XML)`
Maps the field from ContactManager to the core application.

For example, use this method call to map a `LinkID` field to a core application:

```
fieldMapping(ABContact#LinkID)
  .withMappingDirection(TO_XML),
```

`.withMappingDirection(TO_BEAN)`
Maps the field from the core application to ContactManager.

Mapping externally specified unique IDs of a ContactManager contact

ContactManager supports creation of a contact with an external unique ID specified by the core application.

- If an external unique ID is specified in the `XmlBackedInstance` data sent in the `createContact` method call, ContactManager populates the `LinkID` of the new contact with that value. The field that ContactManager checks for in the `createContact` call is `External_UniqueID`.
- If the `createContact` method call does not specify an external unique ID, ContactManager generates its own unique ID and stores that value in the contact's `LinkID`.

For example, the following lines of mapping code from various parts of the `construct` method in the `ContactManager` class `ContactMapper` support this mechanism:

```
fieldMapping(ABContact#External_UniqueID),
fieldMapping(ABContactAddress#External_UniqueID),
fieldMapping(Address#External_UniqueID),
fieldMapping(ABContactTag#External_UniqueID),
fieldMapping(EFTData#External_UniqueID),
fieldMapping(ABContactCategoryScore#External_UniqueID),
```

See also

- “PolicyCenter mapping of externally specified unique IDs” on page 320
- “Creating and linking a contact” on page 193
- “ContactManager link IDs and comparison to other IDs” on page 299
- “ABContactAPI methods” on page 303

Mapping foreign keys of a ContactManager contact

The method `fkMapping(`*`Entity#ForeignKey`*`)` maps foreign keys for `ABContact` entities, subentities, and join tables. You can use the following additional qualifying methods as well:

`withMappingDirection(TO_XML)`

Specifies that the foreign key is mapped to the core application.

`withMappingDirection(TO_BEAN)`

Specifies that the foreign key is mapped from the core application.

Use the `fieldMapping` method to map all the fields of the entity to which a foreign key refers.

For example, the foreign key on `ABContactAddress` that points to an `Address` object is defined as follows:

```
fkMapping(ABContactAddress#Address),
```

Additionally, the fields for the `Address` object must also be defined by using `fieldMapping` method calls. The code for the `Address` entity referenced by the foreign key is:

```
fieldMapping(Address#LinkID)
  .withMappingDirection(TO_XML),
fieldMapping(Address#External_PublicID),
fieldMapping(Address#AddressLine1),
fieldMapping(Address#AddressLine1Kanji),
fieldMapping(Address#AddressLine2),
fieldMapping(Address#AddressLine2Kanji),
fieldMapping(Address#AddressLine3),
fieldMapping(Address#AddressType),
fieldMapping(Address#City),
fieldMapping(Address#CityKanji),
fieldMapping(Address#Country),
fieldMapping(Address#County),
fieldMapping(Address#Description),
fieldMapping(Address#GeocodeStatus),
fieldMapping(Address#PostalCode),
fieldMapping(Address#State),
fieldMapping(Address#ValidUntil),
fieldMapping(Address#CEDEX),
fieldMapping(Address#CEDEXBureau),
```

Mapping array references of a ContactManager contact

The method `arrayMapping(`*`Entity#ArrayKey`*`)` maps array references for `ABContact` entities and subentities. You can use the following additional qualifying methods as well:

`withMappingDirection(TO_XML)`

Specifies that the array reference is mapped to the core application.

`withMappingDirection(TO_BEAN)`

Specifies that the array reference is mapped from the core application.

Use the `fieldMapping` method to map all the fields of the entity to which an array reference refers.

For example, the array extension example “Extending contacts with an array” on page 138 adds a new entity named `ContactServiceState`. This entity represents a state in which a vendor contact operates. Each vendor can operate in more than one state, so the example adds an array reference to `ABContact` named `ContactServiceArea` that references an array of `ContactServiceState` entities.

> **Note:** The example also creates a ClaimCenter `ContactServiceState` entity and a `ContactServiceArea` array reference on the ClaimCenter `Contact` entity.

To transfer this data into and out of ContactManager, you define the array reference and map the fields of the entity to which it refers in `ContactMapper`, as follows:

```
arrayMapping(ABContact#ContactServiceArea),
fieldMapping(ContactServiceState#LinkID)
  .withMappingDirection(TO_XML),
fieldMapping(ContactServiceState#External_PublicID),
fieldMapping(ContactServiceState#External_UniqueID),
fieldMapping(ContactServiceState#ServiceState)
```

Special handling in ContactManager for addresses

Addresses sent from core applications require special handling in ContactManager to handle scenarios like swapping a primary address for a secondary address. Turning address data into XML does not require any special handling, so addresses being sent to core applications use the normal `ContactMapper` code. For these reasons, the following foreign key and array references are defined as one-way, from ContactManager to the core application:

```
fkMapping(ABContact#PrimaryAddress)
  .withMappingDirection(TO_XML),
arrayMapping(ABContact#ContactAddresses)
  .withMappingDirection(TO_XML),
```

When a change to a primary or secondary address for a contact comes in from a core application, ContactManager calls `ABUpdateBeanPopulator.populateAddresses` instead of the regular mapping code.

While the address foreign key and array reference require special handling, the fields of an address do not. You can add or delete address fields as needed by using the `fieldMapping` method.

Core application ContactMapper class

In the Guidewire core applications, you use the class `gw.contactmapper.ab1000.ContactMapper` to map entities between the core application and ContactManager. The class maps entities to an XML object to send to ContactManager and maps entities from the XML objects received from ContactManager.

You can access this class in Guidewire Studio from the **Project** window. Navigate to **configuration** > **gsrc** and then to `gw.contactmapper.ab1000.ContactMapper`.

> **Note:** ContactManager also has a `ContactMapper` class. If you want ContactManager to handle changes you make in the ClaimCenter class, you must update the ContactManager class as well. For example, you might add a new field to a `Contact` subtype and want ContactManager to use that new field with the `ABContact` subtype.

Each core application uses `ContactMapper` to map differing field names used by the core application and ContactManager. To map differing contact entity names and typecodes used by ClaimCenter and ContactManager, each core application uses a separate *`xx`*`NameMapper` Gosu class. You can access this class for each core application in Guidewire Studio from the **Project** window. Navigate to **configuration** > **gsrc** and then to `gw.contactmapper.ab1000`. Double-click the following class for your application to open it an editor:

- ClaimCenter – `CCNameMapper`
- PolicyCenter – `PCNameMapper`
- BillingCenter – `BCNameMapper`

There is no equivalent ContactManager class for mapping differing entity and typecode names. All the mapping of differing names is done on the core application side.

See also

- "ContactManager ContactMapper class" on page 314.

- "ContactMapper class" on page 313.
- For an example that shows how to map entity names from ClaimCenter to ContactManager, see "Map the new subtype names in ClaimCenter" on page 127.

Mapping fields of a core application contact

The method `fieldMapping(`*`Entity#Field`*`)` by default maps `Contact` entity and subentity fields in both directions:

To ContactManager

From a core application entity to an XML object to be sent to ContactManager

From ContactManager

From an XML object sent by ContactManager to a a core application

You also use this method to map fields of an entity that the contact references with either an array reference or a foreign key. To map a foreign key or array reference itself, you use different methods, as described later.

Each method call with qualifying methods, if any, is an entry in the set of return values. If you add a method to the set, add a comma after it if your method is not the last method in the set.

Specifying a single mapping direction for a field

You specify a single direction for the field mapping by using the qualifying method `withMappingDirection`, as follows:

`.withMappingDirection(TO_XML)`

Maps the field from the core application to ContactManager.

For example, use the following method call to map a `PublicID` field to the ContactManager `External_PublicID` field. This mapping is one-way because the core application sets and maintains this value.

```
fieldMapping(Contact#PublicID)
  .withMappingDirection(TO_XML)
  .withABName(MappingConstants.EXTERNAL_PUBLIC_ID),
```

`.withMappingDirection(TO_BEAN)`

Maps the field from ContactManager to the core application.

Mapping fields with differing names

If a field for a contact has a different name in the core application from the name in ContactManager, you map it by using the `withABName` method. For example, the `AddressBookUID` property on a ClaimCenter `Contact` entity is called `LinkID` on an `ABContact` in ContactManager. The following code maps these different field names for a `Contact` entity:

```
fieldMapping(Contact#AddressBookUID)
  .withABName(MappingConstants.LINK_ID),
```

Mapping foreign keys of a core application contact

The method `fkMapping(`*`Entity#ForeignKey`*`)` maps foreign keys for `Contact` entities, subentities, and join tables. The method by default operates in both directions. You can use the following additional qualifying methods as well:

`withMappingDirection(TO_XML)`

Specifies that the foreign key is mapped to ContactManager.

`withMappingDirection(TO_BEAN)`

Specifies that the foreign key is mapped from ContactManager.

`withABName(`*`name`*`:String)`

Specifies a different name for the foreign key on the entity in ContactManager

Use the `fieldMapping` method to map all the fields of the entity to which a foreign key refers.

For example, the foreign key on `ContactAddress` that points to an `Address` object is defined as follows:

```
fkMapping(ContactAddress#Address),
```

Additionally, the fields for the `Address` object must all be defined by using `fieldMapping` method calls. The code for the `Address` entity referenced by the foreign key is:

```
fieldMapping(Address#AddressBookUID)
  .withABName(MappingConstants.LINK_ID),
fieldMapping(Address#PublicID)
  .withMappingDirection(TO_XML)
  .withABName(MappingConstants.EXTERNAL_PUBLIC_ID),
fieldMapping(Address#AddressLine1),
fieldMapping(Address#AddressLine2),
fieldMapping(Address#AddressLine3),
fieldMapping(Address#AddressType),
fieldMapping(Address#City),
fieldMapping(Address#County),
fieldMapping(Address#Country),
fieldMapping(Address#Description),
fieldMapping(Address#GeocodeStatus),
fieldMapping(Address#PostalCode),
fieldMapping(Address#State),
fieldMapping(Address#ValidUntil),
fieldMapping(Address#AddressLine1Kanji),
fieldMapping(Address#AddressLine2Kanji),
fieldMapping(Address#CityKanji),
fieldMapping(Address#CEDEX),
fieldMapping(Address#CEDEXBureau),
```

Mapping array references of a core application contact

The method `arrayMapping(`*`Entity#ArrayKey`*`)` maps array references for `Contact` entities and subentities. The method by default operates in both directions. You can use the following additional qualifying methods as well:

`withMappingDirection(TO_XML)`

Specifies that the array reference is mapped to ContactManager.

`withMappingDirection(TO_BEAN)`

Specifies that the array reference is mapped from ContactManager.

For example, in ClaimCenter, the array of category scores from a vendor performance review is maintained by ContactManager. Therefore, the direction of the array reference is one-way, from ContactManager:

```
arrayMapping(Contact#CategoryScores)
  .withMappingDirection(TO_BEAN),
```

`withABName(`*`name`*`:String)`

Specifies a different name for the array reference on the entity in ContactManager.

Use the `fieldMapping` method to map all the fields of the entity to which an array reference refers.

For example, the array extension example “Extending contacts with an array” on page 138 adds a new entity named `ContactServiceState`. This entity represents a state in which a vendor contact operates. Each vendor can operate in more than one state, so the example adds an array reference to `Contact` named `ContactServiceArea` that references an array of `ContactServiceState` entities.

Note: The example also creates a ContactManager `ContactServiceState` entity and a `ContactServiceArea` array reference on the ContactManager `ABContact` entity.

To transfer this data to and from a core application, you define the array reference and map the fields of the entity to which it refers in `ContactMapper`, as follows:

```
arrayMapping(Contact#ContactServiceArea),
fieldMapping(ContactServiceState#AddressBookUID)
  .withABName(MappingConstants.LINK_ID),
fieldMapping(ContactServiceState#PublicID)
  .withMappingDirection(TO_XML)
  .withABName(MappingConstants.EXTERNAL_PUBLIC_ID),
fieldMapping(ContactServiceState#ServiceState)
```

PolicyCenter mapping of externally specified unique IDs

PolicyCenter specifies unique IDs for the contacts it creates and sends to ContactManager. ContactManager uses these unique IDs for the new contact's `LinkID` value rather than creating its own unique ID.

The PolicyCenter `ContactMapping` class defines the following mapping for the unique ID of a contact that is to be created in ContactManager:

```
fieldMapping(Contact#ExternalID)
  .withMappingDirection(TO_XML)
  .withABName(EXTERNAL_UNIQUE_ID),
```

See also

- "PolicyCenter contact creation with external unique IDs" on page 194
- "Mapping externally specified unique IDs of a ContactManager contact" on page 315
- "Creating and linking a contact" on page 193
- "ContactManager link IDs and comparison to other IDs" on page 299
- "ABContactAPI methods" on page 303

Special handling in ContactMapper for core applications

In some cases, it is not possible to use `ContactMapper` code to handle the transfer of contact data. In those cases, the core application uses methods that handle more complex cases, like related contacts and addresses.

Special handling in one direction

Address handling is an example of special handling of data sent from ContactManager. Addresses sent from ContactManager require special handling in core applications to support scenarios like swapping a primary address for a secondary address. Turning address data into XML does not require any special handling, so addresses being sent to ContactManager use the normal `ContactMapper` code. For these reasons, the following foreign key and array references are defined as one-way, from the core application to ContactManager:

```
fkMapping(Contact#PrimaryAddress)
  .withMappingDirection(TO_XML),
arrayMapping(Contact#ContactAddresses)
  .withMappingDirection(TO_XML),
```

When a change to a primary or secondary address for a contact comes in from a core application, the core application calls `populateAddresses` instead of the regular mapping code. The method is defined in `ContactIntegrationXMLMapperAppBase`.

This method call is explicit in ClaimCenter:

```
arrayMapping(Contact#ContactAddresses)
  .withMappingDirection(TO_BEAN)
  .withArrayBeanBlock( \ am, bp ->
    populateAddresses(bp.Bean as Contact, bp.XmlBackedInstance)),
```

PolicyCenter and BillingCenter also call `populateAddresses` for incoming addresses, but the method call is not explicit in `ContactMapper`.

While the address foreign key and address array reference require special handling, the fields of an address do not. You can add or delete address fields as needed by using the `fieldMapping` method.

Special handling in both directions

In ClaimCenter, related contacts use a parameter of the `withMappingDirection` method not discussed so far:

```
.withMappingDirection(BOTH)
```

You can use this method to specify one behavior for a field when it is sent to ContactManager and another behavior when the field is received from ContactManager. For example:

```
fkMapping(ContactContact#RelatedContact)
  .withMappingDirection(BOTH)
  .withABName("RelABContact")
  .withEntityXMLBlock( \ lm, xp -> populateContactXmlForRelatedContact(lm, xp))
  .withEntityBeanBlock( \ lm, bp -> populateBeanFromXml(bp)),
```

Excluding contact fields from ClaimCenter contact synchronization

You can use the `ContactMapper` method `withAffectsSync` to configure fields to be excluded from the set of fields that ClaimCenter uses to determine if a contact is synchronized with ContactManager. By default, all fields that you add to `ContactMapper` are included in the fields that are checked for synchronization status.

If there is a field in `ContactMapper` that you want excluded from the synchronization check, use the `withAffectsSync` method and set its parameter to `false`. For example, the following code excludes the field `CategoryScores`:

```
arrayMapping(Contact#CategoryScores)
  .withMappingDirection(TO_BEAN)
  .withAffectsSync(false),
```

Note: You do not use `ContactMapper` for fields that determine contact relationships, such as `contactBidiRelCode="employer"`. The inclusion or exclusion of contact relationship fields is defined in the `RelationshipSyncConfig` class.

See also

- "Synchronizing ClaimCenter contact fields" on page 205
- "Synchronizing ClaimCenter and ContactManager contacts" on page 201

Excluding contact fields from being saved in the ClaimCenter database

You can use the `ContactMapper` method `withPersist` to specify that a contact field not be saved in the ClaimCenter database. This feature enables ClaimCenter to receive the field from ContactManager and display it the ClaimCenter user interface, but not save the field. By default, all contact fields in `ContactMapper` are saved, or *persisted*.

IMPORTANT: Setting a field to not persist means that you want the ClaimCenter user to be able to see the value of the field but not change it. Do not enable editing for the field in screens like `ContactDetails.pcf`, and do not send the field back to ContactManager.

To set a field to not persist, use the `withPersist` method and set its parameter to `false`. You must also use `withMappingDirection(TO_BEAN)` to map the field only from ContactManager to ClaimCenter. Additionally, use `withAffectsSync(false)` to exclude the field from being included in the synchronization check. Otherwise, the contact would always be out of sync. For example:

```
arrayMapping(Contact#ContactManagerOnlyField)
  .withMappingDirection(TO_BEAN)
  .withAffectsSync(false)
  .withPersist(false),
```

Excluding properties from contact synchronization

In general, to determine which properties to consider in determining if a contact is synchronized, in most cases you use the `ContactMapper` class. With the exception of contact relationships, `ContactMapper` must list all contact fields to be transferred between ClaimCenter and ContactManager. If you omit a property from this class, it is not transferred or considered for synchronization.

By default, all properties listed in `ContactMapper` are considered in determining synchronization status.

If there is a property in `ContactMapper` that you want excluded from the synchronization check, use the `withAffectsSync` method and set its parameter to `false`. For example, the following code excludes the property `CategoryScores`:

```
arrayMapping(Contact#CategoryScores)
  .withMappingDirection(TO_BEAN)
  .withAffectsSync(false),
```

See also

- “Excluding contact fields from ClaimCenter contact synchronization” on page 321

Overview of ContactManager plugins

Plugins are software modules that ContactManager uses to perform actions or calculate results. Strictly speaking, the term *plugin* refers to an interface, and the term *plugin implementation* is a class that implements a plugin interface. However, in practice, *plugin implementation* is often shortened to just *plugin*.

As described in the Integration Guide, you can write your own plugin implementations of Guidewire plugin interfaces in Gosu.

Additionally, there are *plugin registry files*, which you access in ContactManager studio. You use a registry to identify which plugin implementation ContactManager is to use.

See also

- For a general overview of plugins, see the *Integration Guide*.

Register a plugin implementation in ContactManager

About this task

An example of a plugin registry in ContactManager is `ClaimSystemPlugin.gwp`, which ContactManager can use to send changes in contacts to ClaimCenter.

Procedure

1. Start Guidewire Studio™ for ContactManager.

At a command prompt, navigate to the ContactManager installation folder and enter the following command:

```
gwb studio
```

2. In the **Project** window, navigate to **configuration** > **config** > **Plugins** > **registry** and then to `ClaimSystemPlugin.gwp`, which is a *plugin registry*. Double-click this file to open it in the editor.

3. In the Plugin Registry editor on the right, in the **Class** field, the default plugin that is registered in the base application is `gw.plugin.integration.StandAloneClientSystemPlugin`.

This class is a *plugin implementation* that implements the `ClientSystemPlugin` class. ContactManager does not use this plugin implementation to communicate with a core application. Its purpose is to provide a way to demonstrate ContactManager without having a core application integrated.

Note: You might have set up ContactManager to integrate with ClaimCenter, as described in “Integrating ContactManager with ClaimCenter in QuickStart” on page 21. In that case, the class that is registered is `gw.plugin.claim.cc1000.CCClaimSystemPlugin`, which ContactManager actually does use to communicate changes to ClaimCenter.

4. If you navigate to **configuration** > **gsrc** and then to `gw.plugin.claim.cc1000.CCClaimSystemPlugin`, you can see that this plugin implementation extends the `ClientSystemPlugin1000` class. The `ClientSystemPlugin1000` class extends the `AbstractClientSystemPlugin` class, which implements `ClientSystemPlugin`.

ContactManager plugins

The following is a list of most of the ContactManager plugins, except for the messaging plugins.

IMPORTANT: If you want to change how ContactManager communicates with a core application to send contact updates with `BillingSystemPlugin`, `ClaimSystemPlugin`, or `PolicySystemPlugin`, extend `gw.plugin.AbstractClientSystemPlugin`. Do not implement the plugin interface `gw.plugin.ClientSystemPlugin`. Implementing the `ClientSystemPlugin` interface will prevent ContactManager from starting.

AuthenticationSourceCreatorPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **AuthenticationSourceCreatorPlugin.gwp**
- Plugin Interface – `gw.plugin.security.AuthenticationSourceCreatorPlugin`
- Default Registered Plugin Implementation – `com.guidewire.pl.system.security.impl.DefaultAuthenticationServicePlugin`

Authenticates the user logging in to ContactManager.

BillingSystemPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **BillingSystemPlugin.gwp**
- Plugin Interface – `gw.plugin.ClientSystemPlugin`

  IMPORTANT: Do not implement this interface. Instead, extend `gw.plugin.AbstractClientSystemPlugin`.

- Default registered sample plugin implementation – `gw.plugin.integration.StandAloneClientSystemPlugin`
- Working integration plugin class implementation – `gw.plugin.billing.cc1000.BCBillingSystemPlugin`
- Parent class of working integration plugin – `gw.plugin.integration.ClientSystemPlugin1000`

Sends changes in contacts to BillingCenter. See “Integrating ContactManager with Guidewire core applications” on page 21.

ClaimSystemPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **ClaimSystemPlugin.gwp**
- Plugin Interface – `gw.plugin.ClientSystemPlugin`

  IMPORTANT: Do not implement this interface. Instead, extend `gw.plugin.AbstractClientSystemPlugin`.

- Default registered sample plugin implementation – `gw.plugin.integration.StandAloneClientSystemPlugin`
- Working integration plugin class implementation – `gw.plugin.claim.cc1000.CCClaimSystemPlugin`
- Parent class of working integration plugin – `gw.plugin.integration.ClientSystemPlugin1000`

Sends changes in contacts to ClaimCenter. See “Integrating ContactManager with Guidewire core applications” on page 21.

GeocodePlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **GeocodePlugin.gwp**
- Parent Class – `gw.api.geocode.AbstractGeocodePlugin`
- Plugin Class Implementation – `gw.plugin.geocode.impl.BingMapsPluginRest`

Connects with a geocoding service to provide geocoding information for addresses. For information on setting up geocoding with this plugin, see “Geocoding and proximity search for vendor contacts” on page 73.

IABContactScoringPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **IABContactScoringPlugin.gwp**
- Plugin Interface – `gw.plugin.contact.IABContactScoringPlugin`
- Plugin Class Implementation – `gw.plugin.spm.impl.ABContactScoringPlugin`

Scores provider reviews sent from ClaimCenter. See “ClaimCenter service provider performance reviews” on page 237.

IAddressAutocompletePlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **IAddressAutocompletePlugin.gwp**
- Plugin Interface – `gw.plugin.addressautocomplete.IAddressAutocompletePlugin`
- Plugin Class Implementation – `gw.api.address.DefaultAddressAutocompletePlugin`

Use this plugin to configure how automatic address completion and fill-in operate. For details on address autocompletion and autofill plugin, see the *Globalization Guide*.

IDocumentContentSource

Note: This plugin is available only in ClaimCenter.

- Registry – **configuration** > **config** > **Plugins** > **registry** > **IDocumentContentSource.gwp**
- Plugin Interface – `gw.plugin.document.IDocumentContentSource`
- Parent Class – `gw.plugin.document.impl.BaseLocalDocumentContentSource`
- Plugin Class Implementation – `gw.plugin.document.impl.AsyncDocumentContentSource`

See “Vendor document management plugins” on page 188.

IDocumentMetadataSource

Note: This plugin is available only in ClaimCenter.

- Registry – **configuration** > **config** > **Plugins** > **registry** > **IDocumentMetadataSource.gwp**
- Plugin Interface – `gw.plugin.document.IDocumentMetadataSource`
- Parent Class – `gw.plugin.document.impl.ServletBackedDocumentBaseSource`
- Plugin Class Implementation – `gw.plugin.document.impl.ServletBackedDocumentMetadataSource`

The plugin is disabled in the base configuration.

See “Vendor document management plugins” on page 188.

IEncryption

- Registry – **configuration** > **config** > **Plugins** > **registry** > **IEncryption.gwp**
- Plugin Interface – `gw.plugin.util.IEncryption`

- Plugin Class Implementation – `gw.plugin.encryption.EncryptionByReversePlugin`

Encodes or decodes a `String` based on an algorithm you provide to hide important data, such as bank account numbers or private personal data. ContactManager does not provide any encryption algorithm in the product. ContactManager only calls the `EncryptionByReversePlugin` implementation, which does nothing. See the *Integration Guide* for information on encryption integration.

IFindDuplicatesPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **IFindDuplicatesPlugin.gwp**
- Plugin Interface – `gw.plugin.contact.IFindDuplicatesPlugin`
- Plugin Class Implementation – `gw.plugin.contact.findduplicates.FindDuplicatesPlugin`

Finds duplicate contacts. See “IFindDuplicatesPlugin plugin interface” on page 328.

IGroupExceptionPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **IGroupExceptionPlugin.gwp**
- Plugin Interface – `gw.plugin.exception.IGroupExceptionPlugin`
- Plugin Class Implementation – `com.guidewire.pl.domain.escalation.RulesBasedGroupExceptionPlugin`

Calls the group exception rule set. See the *Integration Guide* for information on exception and escalation plugins.

IPhoneNormalizerPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **IPhoneNormalizerPlugin.gwp**
- Plugin Interface – `gw.plugin.phone.IPhoneNormalizerPlugin`
- Parent Class of Registered Plugin – `gw.api.phone.DefaultPhoneNormalizerPlugin`
- Default Registered Plugin Class – `gw.plugin.phone.DefaultABPhoneNormalizerPlugin`

The default registered plugin class extends the `DefaultPhoneNormalizerPlugin` class, which extends `AbstractPhoneNormalizerPlugin`. `DefaultABPhoneNormalizerPlugin` provides support for phone numbers to `ABContact` and its subtypes. See the *Administration Guide* for more information on the phone number normalizer plugin.

ITestingClock

- Registry – **configuration** > **config** > **Plugins** > **registry** > **ITestingClock.gwp**
- Plugin Interface – `gw.plugin.system.ITestingClock`
- Parent Class of Registered Plugin – `com.guidewire.pl.plugin.system.internal.LongBasedTestingClock`
- Plugin Class Implementation – `com.guidewire.pl.plugin.system.internal.OffsetTestingClock`

The default registered plugin class extends the `LongBasedTestingClock` class, which extends `com.guidewire.pl.plugin.system.internal.ClusterWideTestingClockBase<java.lang.Long>`. Used for testing complex behavior over a long span of time, such as timeouts that are multiple days or weeks later. This plugin is for development (non-production) use only. It programmatically changes the system time to simulate passage of time in ContactManager.

IMPORTANT: You must never use the testing clock plugin on a production server. See “Testing clock plugin interface” on page 339.

IUserExceptionPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **IUserExceptionPlugin.gwp**
- Plugin Interface – `gw.plugin.exception.IUserExceptionPlugin`

- Plugin Class Implementation – `com.guidewire.pl.domain.escalation.RulesBasedUserExceptionPlugin`

Calls the user exception rule set. See the *Integration Guide* for information on exception and escalation plugins.

OfficialIdToTaxIdMappingPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **OfficialIdToTaxIdMappingPlugin.gwp**
- Plugin Interface – `gw.plugin.contact.OfficialIdToTaxIdMappingPlugin`
- Plugin Class Implementation in ContactManager, ClaimCenter, and BillingCenter – `com.guidewire.pl.plugin.contact.internal.AlwaysFalseOfficialIdToTaxIdMappingPlugin`
- Plugin Class Implementation in PolicyCenter – `gw.plugin.contact.impl.PCOfficialIdToTaxIdMappingPlugin`

In the base configuration, this plugin is used only by PolicyCenter. However, it is available to all core applications. In the base configurations of ContactManager, ClaimCenter, and BillingCenter, the registered plugin is `AlwaysFalseOfficialIDToTaxIDMappingPlugin`, which always returns `false`. ContactManager, ClaimCenter, and BillingCenter support only a single `TaxID` field and do not need to do this mapping.

In the base configuration of PolicyCenter, the registered plugin is `gw.plugin.contact.impl.PCOfficialIdToTaxIdMappingPlugin`. PolicyCenter supports an array of official IDs for each contact. For the integration with ContactManager to work, PolicyCenter needs to be able to map one of the typecodes from its `OfficialIDType` typelist to the contact's `TaxID` field. In PolicyCenter, this field is the contact's primary ID for identification purposes.

In the base configuration of PolicyCenter, the plugin implementation overrides the method `isTaxId`. This method override determines if the tax ID is in the `OfficialIDType` typelist as a Social Security Number (`TC_SSN`) or a Federal Employer Identification Number (`TC_FEIN`). You can extend PolicyCenter with your own official IDs and map the ones that are appropriate for your locale or country.

PendingContactChangeConfigurationPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **PendingContactChangeConfigurationPlugin.gwp**
- Plugin Interface – `gw.plugin.contact.PendingContactChangeConfigurationPlugin`
- Plugin Class Implementation – `gw.plugin.contact.pendingchange.PendingContactChangeConfigurationPluginImpl`

Controls the matching used in generating the contact difference view in the Pending Updates screen. To configure how pending updates work, write a plugin that implements the interface.

PolicySystemPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **PolicySystemPlugin.gwp**
- Plugin Interface – `gw.plugin.ClientSystemPlugin`

IMPORTANT: Do not implement this interface. Instead, extend `gw.plugin.AbstractClientSystemPlugin`.

- Default registered plugin implementation – `gw.plugin.integration.StandAloneClientSystemPlugin`
- Working integration plugin class implementation – `gw.plugin.policy.cc1000.PCPolicySystemPlugin`
- Parent class of working integration plugin – `gw.plugin.integration.ClientSystemPlugin1000`

Sends changes in contacts to PolicyCenter. See "Integrating ContactManager with Guidewire core applications" on page 21.

ValidateABContactCreationPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **ValidateABContactCreationPlugin.gwp**

- Plugin Interface – `gw.plugin.contact.ValidateABContactCreationPlugin`
- Parent Class of Registered Plugin Class – `gw.plugin.contact.ValidateABContactCreationPluginBase`
- Default Registered Plugin Class – `gw.plugin.contact.ValidateABContactCreationPluginImpl`

Validates that creation criteria have been met before creating a contact. See “ValidateABContactCreationPlugin plugin interface” on page 337.

ValidateABContactSearchCriteriaPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **ValidateABContactSearchCriteriaPlugin.gwp**
- Plugin Interface – `gw.plugin.contact.ValidateABContactSearchCriteriaPlugin`
- Parent Class of Registered Plugin Class – `gw.plugin.contact.ValidateABContactSearchCriteriaPluginBase`
- Default Registered Plugin Class – `gw.plugin.contact.ValidateABContactSearchCriteriaPluginImpl`

Validates that search criteria have been met for finding a contact.

To specify that a Contact entity field is required in a search, add it to `gw.plugin.contact.ValidateABContactSearchCriteriaPluginImpl`.

The following code snippet from `ValidateABContactSearchCriteriaPluginBase` shows the default search criteria defined for an `ABPerson` entity:

```
protected function abPersonCanSearch(
  searchCriteria : ABContactSearchCriteria) : boolean {
  if (searchCriteria.FirstName != null or
      searchCriteria.FirstNameKanji != null) {
    if (searchCriteria.Keyword == null and
        searchCriteria.KeywordKanji == null) {
      return false
    }
  }
  if (searchCriteria.Keyword == null
      and searchCriteria.KeywordKanji == null
      and searchCriteria.FirstName == null
      and searchCriteria.FirstNameKanji == null
      and searchCriteria.TaxID == null
      and satisfiesNoLocaleSpecificCriteriaRequirements(
        searchCriteria)
      and searchCriteria.Address.PostalCode == null
      and not searchCriteria.isValidProximitySearch()){
    return false
  }
  return true
```

WebservicesAuthenticationPlugin

- Registry – **configuration** > **config** > **Plugins** > **registry** > **WebservicesAuthenticationPlugin.gwp**
- Plugin Interface – `gw.plugin.security.WebservicesAuthenticationPlugin`
- Plugin Class Implementation – `gw.plugin.security.DefaultWebservicesAuthenticationPlugin`

For WS-I web services only, configures custom authentication logic. See the *Integration Guide* for information on the web services authentication plugin.

ContactManager messaging plugins

Note:

See the *Integration Guide* for information on messaging and events.

BCBillingSystemTransport

- Registry – **configuration** > **config** > **Plugins** > **registry** > **BCBillingSystemTransport.gwp**
- Interfaces –

- `gw.plugin.messaging.MessageTransport`
- `gw.plugin.InitializablePlugin`

- Plugin Class Implementation – `gw.plugin.integration.InitializableClientSystemTransport`

CCClaimSystemTransport

- Registry – **configuration** > **config** > **Plugins** > **registry** > **CCClaimSystemTransport.gwp**
- Interfaces –
  - `gw.plugin.messaging.MessageTransport`
  - `gw.plugin.InitializablePlugin`
- Plugin Class Implementation – `gw.plugin.integration.InitializableClientSystemTransport`

PCPolicySystemTransport

- Registry – **configuration** > **config** > **Plugins** > **registry** > **PCPolicySystemTransport.gwp**
- Interfaces –
  - `gw.plugin.messaging.MessageTransport`
  - `gw.plugin.InitializablePlugin`
- Plugin Class Implementation – `gw.plugin.integration.InitializableClientSystemTransport`

documentStoreTransport

- Registry – **configuration** > **config** > **Plugins** > **registry** > **documentStoreTransport.gwp**
- Interfaces –
  - `gw.plugin.messaging.MessageTransport`
  - `gw.plugin.InitializablePlugin`
- Plugin Class Implementation – `gw.plugin.document.impl.DocumentStoreTransport`

emailMessageTransport

- Registry – **configuration** > **config** > **Plugins** > **registry** > **emailMessageTransport.gwp**
- Parent Class – `gw.api.email.AbstractEmailMessageTransport`
- Plugin Class Implementation – `gw.plugin.email.impl.EmailMessageTransport`

IFindDuplicatesPlugin plugin interface

The `FindDuplicatesPlugin` plugin implementation implements the `IFindDuplicatesPlugin` plugin interface in Gosu and is available in Guidewire Studio™ for ContactManager. To view or edit the plugin implementation, start ContactManager Studio. Then, in the **Project** window, navigate to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.FindDuplicatesPlugin`.

> **Note:** The `FindDuplicatesPlugin` plugin implementation and its helper classes use *Gosu generics*. For more information, see the *Gosu Reference Guide*.

The plugin attempts to find exact and potential matches for the subtype of `ABContact` that is passed to it. If no duplicate finder exists for the subtype, the plugin traverses the `ABContact` tree to find the closest parent type it can use in the search.

The plugin uses a set of duplicate finder classes, which define potential and exact match logic for a set of `ABContact` subtypes. The plugin sets up the duplicate finder classes in two `Map` definitions:

```
static final var WIDE_MAP : Map<typekey.ABContact,
        Type<DuplicateFinderBase>> = {
  typekey.ABContact.TC_ABCOMPANY ->
```

```
      CompanyDuplicateFinder<ABCompany>,
    typekey.ABContact.TC_ABPERSON ->
      PersonDuplicateFinder<ABPerson>,
    typekey.ABContact.TC_ABPLACE ->
      PlaceDuplicateFinder
  }
  static final var DEFAULT_MAP : Map<typekey.ABContact,
         Type<DuplicateFinderBase>> = {
    typekey.ABContact.TC_ABCOMPANYVENDOR ->
      CompanyDuplicateFinder<ABCompanyVendor>,
    typekey.ABContact.TC_ABCOMPANY ->
      CompanyDuplicateFinder<ABCompany>,
    typekey.ABContact.TC_ABPERSONVENDOR ->
      PersonVendorDuplicateFinder,
    typekey.ABContact.TC_ABUSERCONTACT ->
      UserDuplicateFinder,
    typekey.ABContact.TC_ABPERSON ->
      PersonDuplicateFinder<ABPerson>,
    typekey.ABContact.TC_ABPLACE ->
      PlaceDuplicateFinder
  }
```

These two `Map` definitions are likely to be the only area of the plugin itself that you need to edit. For example, if you add a duplicate finder class for the `ABAttorney` subtype, you could add a typekey definition for that class to the `DEFAULT_MAP` definition. You do not have to add a map definition for any subtype, even a subtype extension that you create. Because the plugin traverses the `ABContact` tree to find the nearest parent type, any subtype of `ABContact` already has matching logic defined for it. Add a duplicate finder class only if you need special matching logic for a subtype.

Note: In the base configuration, `WIDE_MAP` is configured with fewer subtypes than `DEFAULT_MAP`, resulting in looser matching. This behavior is the intended use of these two variables. However, there is no semantic difference between `WIDE_MAP` and `DEFAULT_MAP`. For example, if you added more subtypes to `WIDE_MAP`, you could make search results using `WIDE_MAP` more specific than `DEFAULT_MAP`.

The `WIDE_MAP` definition is used by the ContactManager batch process Duplicate Contact Finder in the configuration parameter `DuplicateContactsWideSearch`. Additionally, it is used in contact linking calls from ClaimCenter.

See also

- "Changing match results and search scope settings" on page 274
- "Detecting and merging duplicate contacts" on page 271
- "Linking in ClaimCenter" on page 195

Duplicate finder classes

`FindDuplicatesPlugin` uses a set of duplicate finder classes, each of which defines:

- The minimum set of fields required to attempt a match.
- The fields required for exact matches for the `ABContact` subtype.
- The fields required for potential matches for the `ABContact` subtype.

Each class checks for the existence of a minimum set of fields for performing a match with this subtype. If there is enough data for a match, the class can perform a search for potential matches and check the potential matches to see if any are an exact match.

If there is not enough data for a match, the `validateMandatoryFields` method throws a `TooLooseContactDuplicateMatchCriteriaException`. These exceptions use display keys, which you can access in Guidewire Studio™ for ContactManager. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor. In the editor, press `Ctrl +F` and search for `TooLooseContactDuplicateMatchCriteriaException`.

To view or edit the duplicate finder classes, start ContactManager Studio. Then, in the **Project** window, navigate to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates`. Under this node, you see all the duplicate finder classes.

In these classes, you can change the logic required for matching a subtype. For example, you want to add a new field to the potential match logic in one of the duplicate finder classes. You must first add appropriate matching code for the field to the query builder class that supports that duplicate finder class. You then add the query finder method call to the duplicate finder class. For an example of a class that adds field matching logic, see “PersonDuplicateFinder class” on page 331.

The query builder classes are located one node down in the **Project** window. Navigate to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.querybuilder`. For more information on these classes, see “Query builder classes” on page 334.

You also might want to define new matching for an `ABContact` subtype that does not have a duplicate finder class. In this case, you would create a new duplicate finder class and a query builder class for the subtype.

> **Note:** You do not have to add a duplicate finder class for any subtype, even a subtype extension that you create. Because the plugin traverses the `ABContact` tree to find the nearest parent type, any subtype of `ABContact` already has matching logic defined for it. Add a duplicate finder class only if you need special matching logic for a subtype.

The duplicate finder class descriptions that follow describe the functionality available in the base product. The `CompanyDuplicateFinder` description includes code for required fields, potential match, and exact match. The code in the other duplicate finders is similar.

CompanyDuplicateFinder class

This Gosu class defines duplicate contact matching for the `ABCompany` subtype. If the subtype is `ABCompanyVendor`, `ABAutoRepairShop`, `ABAutoTowingAgcy`, `ABLawFirm`, or `ABMedicalCareOrg`, matching is performed for `ABCompany`. The class uses the query builder `CompanyQueryBuilder`, described at “CompanyQueryBuilder class” on page 335.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.CompanyDuplicateFinder`.

> **Note:** The match criteria descriptions that follow includes the code for required fields, potential match, and exact match. The code in the other duplicate finders is similar.

`CompanyDuplicateFinder` defines the following match criteria.

Fields to match

`Name`

Match this field as `starts with` or `equals`, depending on context.

`PrimaryAddress` (`AddressLine1`, `City`, `State`, `PostalCode`)

Match these `PrimaryAddress` fields as `equals`.

`TaxID`

Match this field as `equals`.

`WorkPhone` or `FaxPhone`

Match any single phone field as `equals`.

Required fields

`Name` and at least one of `PrimaryAddress`, `TaxID`, or any phone field: `WorkPhone` or `FaxPhone`.

The following code defines the required fields for an `ABCompany`:

```
override function validateMandatoryFields() {
  if (_searchContact.Name == null or
      (hasNoPrimaryAddress() and
      hasNoPhoneNumber() and
      _searchContact.TaxID == null))
    throwException(_searchContact)
}
```

Potential match types

- Starts with `Name` and equals either `PrimaryAddress` or a phone field.

- Equals `TaxID`.

The following code defines the potential match fields for an `ABCompany`:

```
override function makeQueries() : List<Query<C>> {
  var queries = new ArrayList<Query<C>>()
  //Query: TaxID
  new CompanyQueryBuilder<C>(_searchContact)
    .hasEqualTaxId()//AND
    .buildAndAdd(queries)
  //Query: Name and PhoneNumber
  if (not hasNoPhoneNumber()) {
    new CompanyQueryBuilder<C>(_searchContact)
      .startsWithName()//AND
      .hasEqualPhoneNumbers()
      .buildAndAdd(queries)
  }
  //Query: Name and Address
  if (not hasNoPrimaryAddress()) {
    new CompanyQueryBuilder<C>(_searchContact)
      .startsWithName()//AND
      .hasEqualAddress()
      .buildAndAdd(queries)
  }
  return queries
}
```

Exact Match

Equals both `TaxID` and `Name`.

The following code defines the exact match fields for an `ABCompany`:

```
override function isExactMatch(
       searchContact : C, resultABContact : C) : boolean {
  return equalsAndNotNull<String>(
           searchContact.TaxID, resultABContact.TaxID) &&
         equalsAndNotNull<String>(
            searchContact.Name, resultABContact.Name)
}
```

PersonDuplicateFinder class

This Gosu class defines matching for the `ABPerson` subtype. If the subtype is `ABAdjudicator`, matching is performed for `ABPerson`.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.PersonDuplicateFinder`.

The class uses the query builder `PersonQueryBuilder`, described at "PersonQueryBuilder class" on page 337. That query builder extends `PersonQueryBuilderBase`, which adds logic for matching the first name, last name, date of birth, and phone numbers of the contacts. You can use that class and this one as an example of how to add matching logic for a field to a query builder and a Duplicate Finder.

`PersonDuplicateFinder` defines the following match criteria:

Fields to match

`FirstName`

Match this field as `starts with` or `equals`, depending on context.

`LastName`

Match this field as `equals`.

`PrimaryAddress` (`AddressLine1`, `City`, `State`, `PostalCode`)

Match these `PrimaryAddress` fields as `equals`.

`LicenseNumber` and `LicenseState`

Match both Driver's License fields as `equals`.

`DateOfBirth`

Match this field as `equals`.

`TaxID`

Match this field as `equals`.

`HomePhone`, `WorkPhone`, `FaxPhone`, or `CellPhone`

Match any single phone field as `equals`.

Required fields

`FirstName` and `LastName` and at least one of `PrimaryAddress`, `TaxID`, any phone field, or `LicenseNumber` and `LicenseState`.

Match types

Potential

Starts with `FirstName` and equals `LastName` and equals one of `PrimaryAddress`, `DateOfBirth`, any phone field, or `LicenseNumber` and `LicenseState`.

Potential

Equals `LastName` and `TaxID`.

Exact

Equals `FirstName` and `LastName` and `DateOfBirth`, and equals one of `PrimaryAddress` or any phone field.

Exact

Equals `FirstName` and `LastName` and `TaxID`.

PersonVendorDuplicateFinder class

This Gosu class defines matching for the `ABPersonVendor` subtype. If the subtype is `ABAttorney` or `ABDoctor`, matching is performed for `ABPersonVendor`.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.PersonVendorDuplicateFinder`.

The class uses the query builder `PersonQueryBuilder`, described at “PersonQueryBuilder class” on page 337.

`PersonDuplicateFinder` defines the following match criteria:

Fields to match

`FirstName`

Match this field as `starts with` or `equals`, depending on context.

`LastName`

Match this field as `equals`.

`PrimaryAddress (AddressLine1, City, State, PostalCode)`

Match these `PrimaryAddress` fields as `equals`.

`TaxID`

Match this field as `equals`.

`HomePhone`, `WorkPhone`, `FaxPhone`, or `CellPhone`

Match any single phone field as `equals`.

Required fields

`FirstName` and `LastName` and at least one of `PrimaryAddress`, `TaxID`, or any phone field.

Match types

Potential

Starts with `FirstName` and equals `LastName` and equals one of `PrimaryAddress` or any phone field.

Potential

Equals `TaxID`.

Exact

Equals `FirstName` and `LastName` and `TaxID`.

PlaceDuplicateFinder class

This Gosu class defines matching for the `ABPlace` subtype. If the subtype is `ABLegalVenue`, matching is performed for `ABPlace`.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.PlaceDuplicateFinder`.

The class uses the query builder `PlaceQueryBuilder`, described at "PlaceQueryBuilder class" on page 335.

`PlaceDuplicateFinder` defines the following match criteria:

Fields to match

`Name`

Match this field as `starts with` or `equals`, depending on context.

`PrimaryAddress` (`AddressLine1`, `City`, `State`, `PostalCode`)

Match these `PrimaryAddress` fields as `equals`.

Required fields

`Name` and `PrimaryAddress`.

Match types

Potential

Starts with `Name`.

Potential

Equals `PrimaryAddress`.

Exact

Equals `Name` and `AddressLine1` and `City` and `State` and `PostalCode`.

UserDuplicateFinder class

This class defines matching for the `ABUserContact` subtype.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.UserDuplicateFinder`.

The class uses the query builder `UserContactQueryBuilder`, described at "UserContactQueryBuilder class" on page 337.

`UserContactDuplicateFinder` defines the following match criteria:

Fields to match

`FirstName`

Match this field as `starts with` or `equals`, depending on context.

`LastName`

Match this field as `equals`.

`EmployeeNumber`

Match this field as `equals`.

Required fields

`FirstName` and `LastName`, or `EmployeeNumber`.

Match types

Potential

Starts with `FirstName` and equals `LastName`.

Exact

Equals `EmployeeNumber`.

Query builder classes

The query builder Gosu classes build queries that the duplicate finder classes use to search the database for specific subtypes of `ABContact`. They define queries that:

- Compare fields to see if they are equal. See `hasEqualTaxID` under "ContactQueryBuilder class" on page 334.
- Compare fields to see if they start with the same characters. See `startsWithName` under "ContactQueryBuilder class" on page 334.
- Compare a set of fields to see if they are all equal. See `hasEqualAddress` under "ContactQueryBuilder class" on page 334.
- Compare a set of fields to see if one of them is equal. See `hasEqualPhoneNumbers` under "ContactQueryBuilder class" on page 334.

For information on the duplicate finder classes, see "Duplicate finder classes" on page 329.

ContactQueryBuilder class

This Gosu class provides a basic set of `ABContact` queries for the query builder classes. The classes that extend this one produce a set of queries for specific `ABContact` subtypes, which in turn are used by the duplicate finder classes for those subtypes.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.querybuilder.ContactQueryBuilder`.

The class builds the following queries:

`hasEqualTaxId`

Adds an expression to the query. The expression determines if the `TaxID` field of the `ABContact` being checked for duplicates is equal to the `TaxID` field of an `ABContact` in the database.

```
function hasEqualTaxId() : U {
  addExpression(new EqualFieldExpression<T>(
       "TaxID", _contact.TaxID))
  return this as U
}
```

`hasEqualPhoneNumbers`

Adds an expression to the query. Determines if the `HomePhone`, `WorkPhone`, or `FaxPhone` field of the `ABContact` being checked for duplicates is equal to the equivalent field of an `ABContact` in the database. If one of the fields matches, the contacts have equal phone numbers.

```
function hasEqualPhoneNumbers() : U {
  var numbers = PhoneNumbers
  var phoneOperators : List<FieldExpression<T>> = {
    new InFieldExpression<T>("HomePhone", numbers),
    new InFieldExpression<T>("WorkPhone", numbers),
    new InFieldExpression<T>("FaxPhone", numbers)
  }
  addExpression(new OrCompositeFieldExpression<T>(
    phoneOperators.toTypedArray() ))
  return this as U
}
```

`hasEqualAddress`

Adds an expression to the query. Compares fields of the `PrimaryAddress` of the `ABContact` being checked for duplicates to the equivalent fields of the `PrimaryAddress` of an `ABContact` in the database. If the `AddressLine1` , `State`, `City`, and `PostalCode` fields are equal for both contacts, the addresses are considered equal.

```
function hasEqualAddress() : U {
  addExpression(new AndCompositeFieldExpression<T>({
    new EqualFieldExpression<T>(
          "AddressLine1", "PrimaryAddress",
          _contact.PrimaryAddress.AddressLine1,false),
    new EqualFieldExpression<T>(
          "State", "PrimaryAddress",
          _contact.PrimaryAddress.State,false),
    new EqualFieldExpression<T>(
           "City", "PrimaryAddress",
           _contact.PrimaryAddress.City.flase),
    new EqualFieldExpression<T>(
           "PostalCode", "PrimaryAddress",
           _contact.PrimaryAddress.PostalCode,false)
    }))
  return this as U
}
```

`startsWithName`

Adds an expression to the query. Uses the entire string in the `Name` field of the `ABContact` being checked for duplicates. Compares this string to a substring at the start of the `Name` field of an `ABContact` in the database. If the string and the substring are equal, the contacts are a starts with match. For example, "Asim" would be a starts with match with "Asimov".

```
function startsWithName() : U {
  addExpression(new StartsWithFieldExpression<T>("Name", _contact.Name))
  return this as U
}
```

CompanyQueryBuilder class

This Gosu class builds queries for an `ABCompany` entity. It extends `ContactQueryBuilder`, making the query logic defined in that class available to the class `CompanyDuplicateFinder`.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.querybuilder.CompanyQueryBuilder`.

See also

- "ContactQueryBuilder class" on page 334
- "CompanyDuplicateFinder class" on page 330

PlaceQueryBuilder class

This Gosu class builds queries for an `ABPlace` entity. It extends `ContactQueryBuilder`, making the query logic defined in that class available to the class `PlaceDuplicateFinder`.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.querybuilder.PlaceQueryBuilder`.

See also

- "ContactQueryBuilder class" on page 334
- "PlaceDuplicateFinder class" on page 333

PersonQueryBuilderBase class

This Gosu class builds queries for querying an `ABPerson` entity. It extends `ContactQueryBuilder` and adds new query logic used by `PersonQueryBuilder` and `UserContactQueryBuilder`. You can compare `PersonQueryBuilderBase` to "ContactQueryBuilder class" on page 334 to see how to add new query logic for a field, such as `LastName` or `LicenseNumber`.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.querybuilder.PersonQueryBuilderBase`.

The class builds the following queries in addition to, or in place of, those in `ContactQueryBuilder`:

`startsWithFirstName`

Adds an expression to the query. Uses the entire string in the `FirstName` field of the `ABPerson` being checked for duplicates. Compares this string to a substring at the start of the `FirstName` field of an `ABPerson` in the database. If the string and the substring are equal, the contacts are a *starts with* match. For example, "Asim" would be a starts with match with "Asimov".

```
function startsWithFirstName() : U {
  addExpression(new StartsWithFieldExpression<T>(
      "FirstName", Contact.FirstName))
  return this as U
}
```

`hasEqualLastName`

Adds an expression to the query. The expression determines if the `LastName` field of the `ABPerson` being checked for duplicates is equal to the `LastName` field of an `ABPerson` in the database.

```
function hasEqualLastName() : U {
  addExpression(new EqualFieldExpression<T>(
      "LastNameDenorm", Contact.LastName, false))
  return this as U
}
```

`hasEqualBirthDate`

Adds an expression to the query. The expression determines if the `DateOfBirth` field of the `ABPerson` being checked for duplicates is equal to the `DateOfBirth` field of an `ABPerson` in the database.

```
function hasEqualBirthDate() : U {
  addExpression(new EqualFieldExpression<T>(
      "DateOfBirth", Contact.DateOfBirth))
  return this as U
}
```

`hasEqualLicenseNumber`

Adds an expression to the query. The expression determines if the `LicenseNumber` and `LicenseState` fields of the `ABPerson` being checked for duplicates are equal to the same fields of an `ABPerson` in the database.

```
function hasEqualLicenseNumber() : U {
  addExpression(new AndCompositeFieldExpression<T>({
    new EqualFieldExpression<T>("LicenseNumber", Contact.LicenseNumber),
    new EqualFieldExpression<T>("LicenseState", Contact.LicenseState)
  }))
  return this as U
}
```

`hasEqualPhoneNumbers`

Adds an expression to the query. The method overrides `ContactQueryBuilder.hasEqualPhoneNumbers` and adds the `CellPhone` field as one of the phone number fields to check. The method determines if the `HomePhone`, `WorkPhone`, `FaxPhone`, or `CellPhone` field of the `ABPerson` being checked for duplicates is equal to the equivalent field of an `ABPerson` in the database. If one of the fields matches, the contacts have equal phone numbers.

```
override function hasEqualPhoneNumbers() : U {
  var numbers = PhoneNumbers
  var phoneOperators : List<FieldExpression<T>> = {
    new InFieldExpression<T>("HomePhone", numbers),
    new InFieldExpression<T>("WorkPhone", numbers),
    new InFieldExpression<T>("FaxPhone", numbers)
    new InFieldExpression<T>("CellPhone", numbers)
  }
  addExpression(new OrCompositeFieldExpression<T>(
    phoneOperators.toTypedArray() ))
  return this as U
}
```

PersonQueryBuilder class

This Gosu class builds queries for an `ABPerson` entity. It extends `PersonQueryBuilderBase`, making the query logic defined in that class available to the classes `PersonDuplicateFinder` and `PersonVendorDuplicateFinder`.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.querybuilder.PersonQueryBuilder`.

See also

- "PersonQueryBuilderBase class" on page 335
- "PersonDuplicateFinder class" on page 331
- "PersonVendorDuplicateFinder class" on page 332

UserContactQueryBuilder class

This Gosu class builds queries for an `ABUserContact` entity. It extends `PersonQueryBuilderBase`, making the query logic defined in that class available to the class `UserDuplicateFinder`. In addition, it adds a method, `hasEqualEmployeeNumber` for comparing equality of the `EmployeeNumber` fields of the two contacts.

To open this class in Guidewire Studio™, navigate in the **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.findduplicates.querybuilder.UserContactQueryBuilder`.

See also

- "PersonQueryBuilderBase class" on page 335
- "UserDuplicateFinder class" on page 333

ValidateABContactCreationPlugin plugin interface

The registry file for this plugin is `ValidateABContactCreationPlugin.gwp`. This plugin interface is implemented in Gosu in the abstract class `ValidateABContactCreationPluginBase`. The class `ValidateABContactCreationPluginImpl` extends this base class and is the default plugin class registered in the base configuration.

The class `ValidateABContactCreationPluginBase` implements most of the logic that validates new `ABContact` instances and instances of `ABContact` subentities.

To extend the contact creation logic, such as for a new subentity, open Guidewire Studio™. Then navigate in the Studio **Project** window to **configuration** > **gsrc** and then to `gw.plugin.contact.ValidateABContactCreationPluginImpl`.

Contact minimum creation requirements

The `ValidateABContactCreationPluginImpl` class enables you to provide your own definition of the minimum criteria required for creating new contacts. In the base configuration, all contacts are required to have a contact tag.

In the base configuration, `ValidateABContactCreationPluginBase` implements the following minimum creation requirements:

- `ABPerson` – Last name, contact tag, and one of the following:
  - Primary address – Address line 1, city, state, ZIP code
  - Phone number – Any of the phone numbers, such as home, cell, fax, or work phone
  - Tax ID
  - Date of birth
  - Driver's license and driver's license state
- `ABCompany` – Name, contact tag, and one of the following:
  - Primary address – Address line 1, city, state, ZIP code

- Phone number – Any of the phone numbers, such as fax or work phone
- Tax ID

- `ABCompanyVendor` – Name, contact tag, and tax ID
- `ABPersonVendor` – Last name, contact tag, and tax ID
- `ABPlace` – Name, contact tag, and primary address, consisting of address line 1, city, state, and ZIP code
- `ABUser` – Last name, contact tag, and employee number

Verify Minimum Criteria work queue

The Verify Minimum Criteria work queue enables you to iterate through a set of contacts that you have imported from staging tables to verify if they meet minimum creation criteria. The work queue uses the `ValidateABContactCreationPlugin` plugin interface.

The work queue iterates over all instances of `ABContact` and its subtypes in ContactManager for which verification has not yet passed, checking if `ABContact.MinimumCriteriaVerified` is `false`. For each unverified contact, the work queue calls the plugin implementation class registered with `ValidateABContactCreationPlugin.gwp`.

Note: In the base configuration, this class is `gw.plugin.contact.ValidateABContactCreationPluginImpl`.

If the plugin class indicates that the contact is valid, the work queue sets the contact's `MinimumCriteriaVerified` field to `true`.

The `MinimumCriteriaVerified` field is translated to XML when contacts are sent over web services. This field is exposed when the contact is obtained by calling `ABContactAPI.retrieveContact`, enabling a core application to determine if a contact has passed minimum validation criteria.

See also

- “Contact minimum creation requirements” on page 337

Configure the TaxID contact creation requirement

About this task

You can set whether or not `TaxID` is required for creating contacts. In the base configuration, this setting affects only `PersonVendor` and `CompanyVendor` subtypes. You set the variable `RequiresTaxID` in the registry file `ValidateABContactCreationPlugin.gwp`.

`ValidateABContactCreationPluginImpl` uses this value when it calls the following method that it inherits from `ValidateABContactCreationPluginBase`:

```
override function setParameters(p0: Map<Object, Object>) {
  _requiresTaxID = Boolean.valueOf(p0.get("RequiresTaxID") as String)
}
```

Procedure

1. If necessary, start Guidewire Studio™ for ContactManager.
2. In the **Project** window, navigate to **configuration** > **config** > **Plugins** > **registry** and double-click `ValidateABContactCreationPlugin.gwp`.
3. The default registered plugin implementation is in the **Gosu Class** field:

   `gw.plugin.contact.ValidateABContactCreationPluginImpl`
4. Under the **Gosu Class** field, `RequiresTaxID` is defined in the **Parameters** table. The default value is `true`. If you set it to `false`, `TaxID` is not required for creating any contact subtype.

Code example for creating ABContact and ABCompany contacts

The following code sample from `ValidateABContactCreationPluginBase` shows code defining contact creation requirements for `ABContact` and `ABCompany`:

```
protected function abContactIsInvalid(contact : ABContact) : boolean {
  return contact.Tags == null or contact.Tags.IsEmpty
}
 protected function abCompanyIsInvalid(contact : ABCompany) : boolean {
  if (abContactIsInvalid(contact))
    return true
   if (RequiresTaxID) {
    return contact.Name == null
      or (isLackingCompleteAddress(contact.PrimaryAddress)
          and isLackingAnyPhoneNumber(contact)
          and contact.TaxID == null)
  } else {
    return contact.Name == null
      or (isLackingCompleteAddress(contact.PrimaryAddress)
          and isLackingAnyPhoneNumber(contact))
}
```

Configuring validation error messages for contact creation

If an `ABContact` instance fails validation, `ValidateABContactCreationPluginImpl` throws `TooLooseContactCreateCriteriaException`, which supports a set of display keys based on locale and `ABContact` subtype.

You can use Guidewire Studio™ for ContactManager to access the set of display keys provided in the base configuration. For example, you can edit the U.S English display key properties file. Navigate in the **Project** window to **configuration** > **config** > **Localizations** > **Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor. In the editor, search for `TooLooseContactCreateCriteriaException`.

There is a set of `TooLooseContactCreateCriteriaException` display keys for `ABContact` and some of its subtypes. These display keys are accessible in code. For example:

```
Java.TooLooseContactCreateCriteriaException.ABCompany
Java.TooLooseContactCreateCriteriaException.ABCompany.ja_JP
Java.TooLooseContactCreateCriteriaException.ABCompanyVendor
Java.TooLooseContactCreateCriteriaException.ABCompanyVendor.ja_JP
```

For example, an `ABContact` that is an `ABCompanyVendor` with locale `en_US` fails validation. It throws an exception, using for its message the display key `Java.TooLooseContactCreateCriteriaException.ABCompanyVendor`. If the Japanese locale is specified, the exception uses the display key `Java.TooLooseContactCreateCriteriaException.ABCompanyVendor.ja_JP`.

Similar behavior applies to `ABContact` entities that are of class `ABCompany` but not `ABCompanyVendor`. They use the `Java.TooLooseContactCreateCriteriaException.ABCompany` display keys.

See also

- For general information on working with display keys, see the *Configuration Guide*.

Testing clock plugin interface

IMPORTANT: The `ITestingClock` plugin interface is supported only for testing on non-production development servers. Do not register an implementation of this plugin on production servers.

To test ContactManager behavior over a simulated long span of time, you can implement the `ITestingClock` plugin interface and programmatically change the system time to simulate the passing of time. For example, you can define a plugin implementation 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.

Time must always increase, not go back in time. Going back in time is likely to cause unpredictable behavior in ContactManager.

See also

- For full information on using `ITestingClock`, see the *Integration Guide*.