Guidewire PolicyCenter™ for Guidewire Cloud

Cloud API Developer Guide

Release: 2025.03

GUIDEWIRE

Product Name: Guidewire PolicyCenter for Guidewire Cloud
Product Release: 2025.03
Document Name: Cloud API Developer Guide
Document Revision: 29-January-2026

Contents

Support

For assistance, visit the Guidewire Community.

Guidewire customers

https://community.guidewire.com

Guidewire partners

https://partner.guidewire.com

part 1

Endpoint architecture

Cloud API provides CRUD (Create Read Update Delete) endpoints that allow integrating systems to interact with PolicyCenter. In order to integrate effectively, it's helpful to understand the architecture of those endpoints. The topics in this section explain the overall architecture behind accessing Cloud API endpoints and the various configuration files that define the architecture.

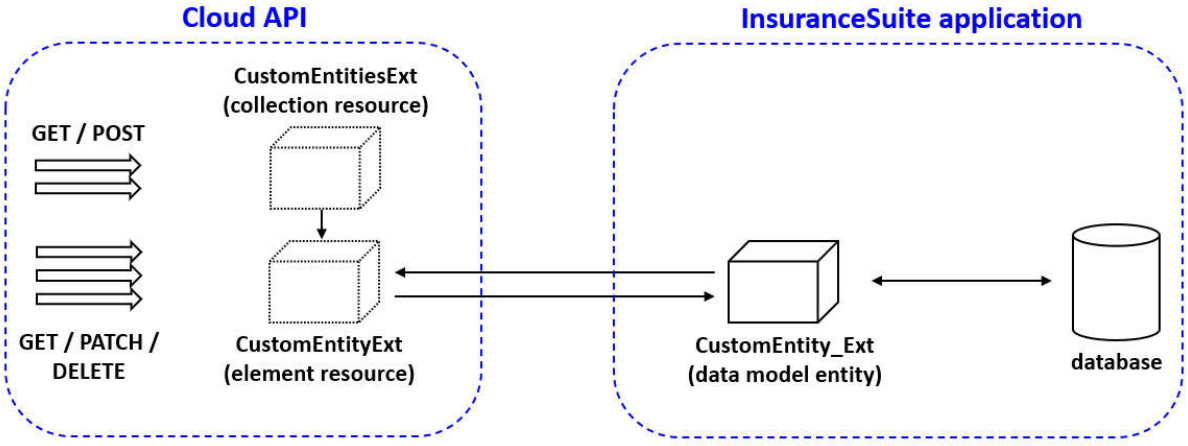
chapter 1

CRUD endpoint architecture

CRUD endpoints is a term that refers to the endpoints that let you GET a collection of a given resource type and that let you GET, POST, PATCH, or DELETE an element of that resource type. The term "CRUD" is an acronym for Create Read Update Delete.

High-level architecture

The following diagram depicts the high-level architecture of a set of CRUD endpoints within Cloud API. These endpoints are for a custom entity whose name is `CustomEntity_Ext`. (Most of the time, endpoints are generated for custom entities. This is why the examples below feature `CustomEntity_Ext`. However, it is also possible to generate endpoints for base configuration entities that do not yet have endpoints.)



There are five CRUD endpoints, but they do not all interact with the same type of resource.

- The first two CRUD endpoints are for GET (for a collection) and POST. These endpoints interact with a collection resource whose name is `customEntitiesExt`. (Note that the resource name uses a plural term.)
- The final three CRUD endpoints are for GET (for a specific element), PATCH, and DELETE. These endpoints interact with an element resource whose name is `customEntityExt`. (Note that the resource name uses a singular term.

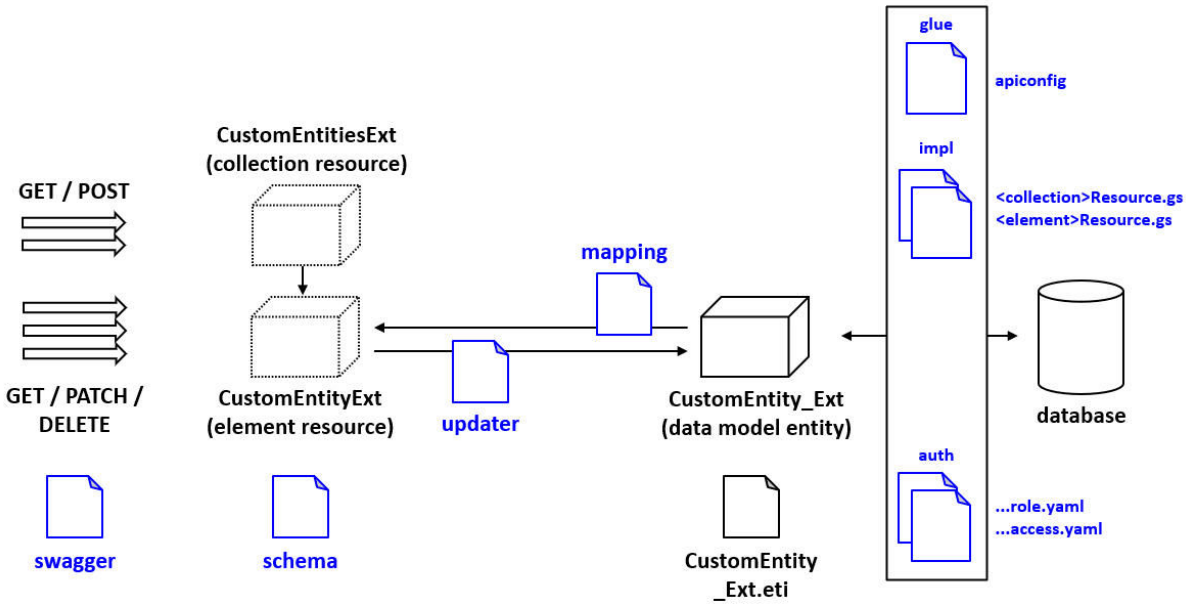
The components of the architecture map to each other in the following ways:

- The collection resource (`customEntitiesExt`) makes use of the element resource.

- The element resource (`customEntityExt`) maps to the data model entity.
- The data model entity (`CustomEntity_Ext`) is used to retrieve data from and send data to the database.

Files that define the architecture

The CRUD endpoint architecture is defined in a series of files, as depicted in the following diagram.



The **swagger file** defines the endpoints themselves (the paths, operations, and associated resources).

The **schema file** defines the schema used by the element and collection resources.

The **mapping file** defines how information is mapped from the data model entity to the element resource. This information is used for GETs, and for the responses of POSTs and PATCHes.

The **updater file** defines how information is mapped from the element resource to the data model entity. This information is used for POSTs and PATCHes.

The **eti file** defines the data model entity.

There are also a series of files that define logic for how the endpoints interact with the application.

- The **apiconfig file** is a "glue" file. One of its purposes is to map both the element resource and collection resource to corresponding "Resource" Gosu files. For collection resources, this file can also specify a default sort order.
- There are two "impl" files that define implementation details.
  - The **<collection>Resource.gs file** is a Gosu file that defines required behaviors for working with collections. This includes behaviors such as how to retrieve the collection from the database.
  - The **<element>Resource.gs file** is a Gosu file that defines required behaviors for working with elements. This includes behaviors such as how to initialize a new element.
- There are two sets of "auth" files that define authorization.
  - There are one or more **role.yaml files** that define endpoint access for callers of the endpoints.
  - There are a set of **access.yaml files** that define resource access for callers of the endpoints.

chapter 2

Reasons to modify configuration files

There are several reasons you might need to modify configuration fields.

Generating endpoints for custom entities

The *REST endpoint generator* is a tool that generates CRUD endpoints for custom data model entities. The REST endpoint generator also creates schema definition files for the resources used by the generated endpoints. However, these files are incomplete. Developers must always add to the generated configuration in the resource definition files.

For more information on the REST endpoint generator, see “The REST endpoint generator” on page 121.

Extending base configuration resources

Cloud API has endpoints that perform CRUD operations on several data model entities in the base configuration. For example, the GET `/common/v1/activities` endpoint queries for instances of the `Activity` data model entity. Insurers can extend these data model entities by adding custom fields. Insurers may also want to expose these fields to the REST endpoints. Insurers can configure schema definition files and resource files to add custom resource properties that map to custom data model fields.

To extend a base configuration resource:

1. If necessary, extend the underlying data model entity with new fields. For more information, see *Configuration Guide*.
2. Configure the appropriate schema, mapping, and updater extension files. For more information, see “Adding scalars” on page 33, “Adding compound datatypes” on page 39, “Adding foreign keys” on page 43, and “Adding one-to-ones” on page 57.
3. Add attributes to define additional property behaviors, if necessary. For more information, see “Modifying endpoint behaviors” on page 73.

Exposing base configuration fields to Cloud API

For the data model entities that are exposed in Cloud API, the base configuration does not expose every field in a given entity to Cloud API. There may be situations where you need to expose one or more of these fields to Cloud API.

For example, the `User` data model entity is exposed to Cloud API. The `VacationStatus` field is also exposed. But, the `ExperienceLevel` field is not. An insurer may want to expose the `ExperienceLevel` field to Cloud API.

The approach for doing this follows the same process as extending a base configuration resource. There are only two differences:

- There is no need to extend the data model as the field in question already exists.

- Guidewire recommends appending "_Ext" to the schema property name to avoid potential conflicts in future releases.

Integration graphs

Cloud API schemas are used to define the integration graphs used by Guidewire App Events . An *integration graph* is a data model graph that defines a set of business information to be sent to an external application as part of outbound integrations. For example, the Claim graph defines what information to send about a claim. Insurers may want to extend Cloud API schemas to ensure that certain information is included in the integration graph.

For more information on integration graphs, see “Integration graphs” on page 163 and the *App Events Guide*.

chapter 3

Syntax for schema configuration files

This section describes the syntax for the different types of schema configuration files.

Schema file syntax

The file attributes

The initial part of a schema extension file contains attributes about the file itself. Do not modify these attributes, as doing so could cause Cloud API to behave in unexpected ways.

- `$schema`: References the JSON Schema namespace declaration
- `x-gw-combine`: Lists an array of files that contain schema definitions that are either referenced by schemas in this file or extended by schemas in this file.

The definitions section

The `definitions` section lists definitions for one or more schemas. For each schema, the following attributes are specified:

- A schema `title` and `description`, which is used for API definition documentation.
- The resource `type`, which, for Cloud API, is typically set to object.
- A list of `properties` for the schema.

Every schema property includes:

- The property name (such as `activityPattern` in the example below)
- The property's `title` and `description`, which is used for API definition documentation.
- A "datatype" for the property.
  - For scalar value properties, `type` is specified. This defines the JSON datatype of the property.
  - For compound value properties (such as typekeys), `ref` is specified. This defines the URI reference for the property.
- Optional attributes as needed for the business functionality of the property.

For example, the following is a portion of the base configuration `Activity` schema:

```
"definitions": {
  "Activity": {
    "title": "Activity",
```

```
    "description": "An `Activity` is an assignable item...",
    "type": "object",
    "properties": {
      "activityPattern": {
        "title": "Activity pattern",
        "description": "The code of the `ActivityPattern` used...
        "type": "string"
        ...
      },
      "activityType": {
        "title": "Activity type",
        "description": "The type of this activity, such as `general`...",
        "$ref": "#/definitions/TypeKeyReference",
        ...
      },
      ...
    }
  }
}
```

Mapping file syntax

The file attributes

The initial part of a mapping extension file contains attributes about the file itself. Do not modify these attributes, as doing so could cause Cloud API to behave in unexpected ways.

- `$schema`: References the JSON Schema namespace declaration
- `x-gw-combine`: Lists an array of files that contain schema definitions that are either referenced by schemas in this file or extended by schemas in this file.

The mapping section

A mapping file contains a `mappers` section, which lists one or more API resources and, for each readable property in the resource, the way data is mapped to the property.

For each resource, the following attributes are specified:

- The `schemaDefinition` that defines the structure of the resource
  - This references a schema declared in a `schema.json` file.
- The data model entity that serves as the `root` for mapping information for this resource.
- A list of `properties`

Every mapper property includes:

- The property name (such as `closeDate`, `description`, and `mandatory` in the example below)
  - A `path` attribute. This is a Gosu expression that is used to populate the value of the property. Typically, this expression returns a value from the entity defined in the root attribute.
  - Depending on the nature of the property, there may be additional attributes.

For example, the following is a portion of the mapper for the base configuration `Activity` resource:

```
"mappers": {
  "Activity": {
    "schemaDefinition": "Activity",
    "root": "entity.Activity",
    "properties": {
      "closeDate": {
        "path": "Activity.CloseDate"
      },
      "description": {
        "path": "Activity.Description"
      },
      "mandatory": {
        "path": "Activity.Mandatory"
      },
      ...
    }
  }
}
```

Note the following:

- This resource is defined in the schema who name is `Activity` (This schema is defined in some other `schema.json` file.)
- The root for the resource mapping is `entity.Activity`.
- For each instance of the resource:
  - The `closeDate` property is set to the `Activity` entity's `CloseDate` field.
  - The `description` property is set to the `Activity` entity's `Description` field.
  - The `mandatory` property is set to the `Activity` entity's `Mandatory` field.

Updater file syntax

The file attributes

The initial part of an updater extension file contains attributes about the file itself. Do not modify these attributes, as doing so could cause Cloud API to behave in unexpected ways.

- `$schema`: References the JSON Schema namespace declaration
- `x-gw-combine`: Lists an array of files that contain schema definitions that are either referenced by schemas in this file or extended by schemas in this file.

The updaters section

An updater file contains an `updaters` section, which lists one or more API resources and, for each writeable property in the resource, the way data is mapped to the data model.

For each resource, the following attributes are specified:

- The `schemaDefinition` that defines the structure of the resource
  - This references a schema declared in a `schema.json` file.
- The data model entity that serves as the `root` for mapping information for this resource.
- A list of `properties`

Every updater property includes:

- The property name (such as `description` and `mandatory` in the example below)
- A `path` attribute. This defines, for each resource property, the data model entity property into which the resource property is to be written.
- Depending on the nature of the property, there may be additional attributes.

For example, the following is a portion of the updater for the base configuration `Activity` resource:

```
"updaters": {
  "Activity": {
    "schemaDefinition": "Activity",
    "root": "entity.Activity",
    "properties": {
      "description": {
        "path": "Activity.Description"
      },
      "mandatory": {
        "path": "Activity.Mandatory"
      },
      ...
    }
  }
}
```

Note the following:

- This resource is defined in the schema who name is `Activity` (This schema is defined in some other `schema.json` file.)

- The root for the resource mapping is `entity.Activity`.
- For each instance of the resource:
  - The value of the `description` property is written to the `Activity` entity's `Description` field.
  - The value of the `mandatory` property is written to the `Activity` entity's `Mandatory` field.

Note that there may be properties that appear in the mapping file but not the updater file. This typically occurs with properties that are read-only. For example, the `Activity` entity has a `CloseDate` property, which the application sets when the activity is closed. This property appears in the mapping file, as it can be read. But it does not appear it the updater file because it cannot be written to.

Extension files

Whenever you configure a schema, you modify an extension file. Within the context of Cloud API, an *extension file* is a file that insurers can modify that adds or overrides the base configuration. All extension files have "ext" in their name and are located in an `ext` subdirectory.

For schema configuration, Cloud API includes the following types of extension files:

- *schema extension files*, where insurers add new properties to a schema
- *mapping extension files*, where insurers define how data is mapped from the data model to the schema
- *updater extension files*, where insurers define how data is mapped from the schema to the data model (for properties that are writeable)

Every API has its own set of extension files. The names of the files, and the nodes in Guidewire Studio that you can use to access them, are:

- `<API>_ext-1.0.schema.json`
  - **integration** > **schemas** > **ext** > **<API>.v1**
- `<API>_ext-1.0.mapping.json`
  - **integration** > **mappers** > **ext** > **<API>.v1**
- `<API>_ext-1.0.updater.json`
  - **integration** > **updaters** > **ext** > **<API>.v1**

Extension files have the same syntax as base configuration files. But extension files often omit some information that is not necessary for extending a schema. For example, when a resource is declared in a base configuration file, the schema file defines the resource's `title`, `description`, and `type`. When that same resource is extended in an extension file with additional properties, there is no need to repeat the `title`, `description`, or `type`. These values are inherited from the base configuration file.

Every `ext` subdirectory has a sibling `gw` subdirectory. The `gw` subdirectory holds the base configuration files. Insurers are not permitted to modify these files. However, you may find it helpful to review those files for example of how base configuration resources are defined.

chapter 4

Swagger and apiconfig files

Swagger files and apiconfig files do not directly define schema behavior. But they do contain information related to schemas. For the sake of completeness, they are defined here.

Swagger files

Within the context of Cloud API, a *swagger file* defines the endpoints and operations for a given API. Typically, there is no need for an insurers to configure a Cloud API swagger file.

Swagger files are stored in the `/apis` subdirectory.

Swagger file syntax

A swagger file can contain a section that defines an API. For example, the following is a portion of the `common_pl-1.0.swagger.yaml` file, which defines the Common API.

```
swagger: "2.0"
info:
  title: "Common API"
  description: "APIs for common InsuranceSuite platform objects like activities and notes"
  version: "1.4.0"
basePath: /common/v1
consumes:
- application/json
produces:
- application/json
```

A swagger file can also contain a `paths` section, which defines a set of endpoint paths and the associated operations and resources for that path. For example, the following is a portion of the `common_pl-1.0.swagger.yaml` file, which defines the first path in the Common API, `/activities`.

```
paths:
  /activities:
    get:
      summary: "Retrieve the `Activity` elements that are assigned to the caller"
      description: "Retrieves the `Activity` elements that are assigned to the caller"
      operationId: getActivities
      x-gw-extensions:
        childResourceType: Activity
        operationType: get-collection
        resourceType: Activities
      x-gw-parameter-sets:
      - get-collection
      responses:
        "200":
          description: "The paginated list of `Activity` elements"
          schema:
            $ref: "#/definitions/ActivityList"
```

Apiconfig files

Within the context of Cloud API, an *apiconfig file* is a glue file that maps both resources to Gosu files. The apiconfig file is also where default sort orders for collections are defined.

The only time you need to configure an apiconfig file to connect resources to Gosu files is when you are completing a configuration for CRUD endpoints created by the REST endpoint generator. For more information, see “Configuring glue and impl classes for generated endpoints” on page 147.

For more information on defining default sort orders, see “Making properties sortable” on page 83.

chapter 5

Integration graphs

An *integration graph* is a special base configuration schema and mapper that certain base configuration outbound integrations use to send information about a parent object and its children objects. Integration graphs are not used by Cloud API, but the files that define them are a part of Cloud API.

As of this release, Cloud API has the following integration graphs:

- The ClaimCenter `claim_graph`, whose parent is Claim
- The PolicyCenter `policyperiod_graph`, whose parent is PolicyPeriod
- The ContactManager `contact_graph`, whose parent is Contact

There are several outbound integrations that make use of integration graphs. For example, integration graphs reflect the structure of the event payload that App Events generates for downstream systems.

Guidewire recommends against insurers configuring integration graphs directly. However, if you create an extension entity that is a descendent of one of the parent entities listed above, you can add that entity to the appropriate integration graph. This is done through the REST endpoint generator. For more information, see “Integration graphs” on page 163.

WARNING: Guidewire recommends against insurers using the integration graph schema/mapping files for custom *outbound* integration points. This is because, if not used properly, the integration point could attempt to generate a payload so large that it compromises server performance or prevents the server from running.

chapter 6

Configuration troubleshooting

If you have an issue with your Cloud API configuration, your primary source of information for troubleshooting is the log files.

Reviewing log file information

Be aware of the following when you're looking at log files:

- There may be multiple logs, so be sure to look through all that might apply.
- Some messages are written to the logs only when the *REST.Config* and *REST.Request* loggers are set to `DEBUG`. When troubleshooting auth issues, Guidewire recommends setting these loggers to `DEBUG`.

Tips for reviewing log files

- If the most recent error in the log doesn't seem to highlight the problem or help you determine a solution, look back at earlier errors. Sometimes processes continue to run after an initial error has been logged. That error could spawn additional errors, but it's that first one that will give you the clues you need to determine what really went wrong.
- If you're troubleshooting authentication issues, see "Troubleshooting auth issues" on page 415.

Common issues found in the log files

The following issues arise frequently and are reported in the log files.

JSON formatting issues

Cloud API relies on the JSON file format for endpoint requests and responses, and for many configuration files. If you have an endpoint that isn't behaving properly, or you've made changes to a configuration file (such as the schema, mapper, or updater files) and you're receiving errors, make sure the JSON is correctly formatted. A missing comma or extra curly brace causes anything relying on the JSON file to fail.

In addition to the information from the log files, various IDEs (such as IntelliJ) will flag errors in JSON files. You can also copy and paste your JSON into an online validator (such as https://jsonlint.com/) to validate your JSON formatting.

Configuration file expressions don't compile

In addition to ensuring you're using valid JSON, you also need to make sure you're using the proper syntax for your schema, updater, and mapper files. See "Syntax for schema configuration files" on page 21 for information on configuration file syntax.

part 2

Adding properties to resources

You can add new properties to Cloud API resources. This can be a new property in a base configuration resource, or a new property in a custom resource. The data mapping requirements for the new property depend on the value that the property stores. The following table summarizes the different use cases.

Value stored in property	Example Property	More information
Scalar	`Activity.LegacySystemID_Ext`, which stores a string identifying the ID for the object in a legacy system	"Adding scalars" on page 33
Compound value, such as a typekey or monetary amount	`Activity.SLAPriority_Ext`, which stores a value from the Priority typelist (such as `urgent`, `high`, or `low`)	"Adding compound datatypes" on page 39
Foreign key	`Activity.ApprovalUser_Ext`, which stores a foreign key to the `User` data model entity. There are separate CRUD endpoints for the `User` entity.	"Adding foreign keys" on page 43
One-to-ones	`Activity.ForeignMarketFees_Ext`, which stores a one-to-one key to the `ForeignMarketFees_Ext` data model entity. The `ForeignMarketFees_Ext` entity does not have its own set of CRUD endpoints. Instead, the object is inlined into the `Activity` schema.	"Adding one-to-ones" on page 57

For a detailed overview of schema architecture and extension file syntax in Cloud API, see "Syntax for schema configuration files" on page 21

If you want to interact directly with the concepts in this topic, go to the following tutorial: "Tutorial: Schema configuration with scalars" on page 67

Note: Guidewire does not recommend adding inline arrays to schemas. If a given resource has an array of related resources and you want to expose those related resource in Cloud API, Guidewire recommends doing so by creating a separate set of child endpoints for the related resources. For information on how to create endpoints for a resource, see "The REST endpoint generator" on page 121.

chapter 7

Adding scalars

Extension files

Every API in Cloud API has a set of three extension files that are used to extend resources used by the endpoints in that API. This includes:

- A schema extension file, where you can define new properties for a resource
- A mapper extension file, where you can define how data from the database is used to populate extension fields
- An updater extension file, where you can define how data from extension fields is written to the database

The names of the extension files are defined in the table below. Note that the file name always starts with the API name. For example, the schema extension file for the Common API is `common_ext-1.0.schema.json`.

Filename	Guidewire Studio node
<API>`_ext-1.0.schema.json`	integration -> schemas -> ext -> *<API>*.v1
<API>`_ext-1.0.mapping.json`	integration -> mappers -> ext -> *<API>*.v1
<API>`_ext-1.0.updater.json`	integration -> updaters -> ext -> *<API>*.v1

Scalars

A scalar is a single simple value, such as a string, number, datetime, or Boolean.

To add a scalar property to a resource, you must modify the schema, mapping, and updater file for the API that contains the endpoints that use the schema. For example, suppose you want to add an extension property for the User schema. This schema is used by endpoints in the Common API. Therefore, you need to modify the following files:

- `common_ext-1.0.schema.json`
- `common_ext-1.0.mapping.json`
- `common_ext-1.0.updater.json`

Note: See also “Tutorial: Schema configuration with scalars” on page 67

The schema extension file

Each schema extension file for every API contains a `definitions` section. In the base configuration, this section is typically blank, as shown below.

```
"definitions": { }
```

You can add the name of one or more schemas to the `definitions` section. In each schema, you can define a `properties` section, which lists one or more properties defined for the schema.

If the property is a base configuration property that is not exposed to Cloud API, Guidewire recommends naming the schema property with an "_Ext" suffix. This is to prevent any possible future conflicts if Guidewire adds the property to the base configuration in a later release.

For scalar values, each property must also have a `type`.

For example, in the base configuration data model, the `User` entity has a `JobTitle` field (this user's job title) and an `ExternalUser` field (whether the user works for the insurer or not). The Admin API has `/user` endpoints, but `JobTitle` and `ExternalUser` are not exposed in the Cloud API `User` resource. Suppose you want to expose them, and you want `JobTitle` to be writeable and `ExternalUser` be read-only. You would add the following to the `admin_ext-1.0.schema.json` extension file.

```
"definitions": {
  "User": {
    "properties": {
      "jobTitle_Ext" : {
        "title": "jobTitle_Ext",
        "description": "The user's job title",
        "type": "string",
      },
      "externalUser_Ext" : {
        "title": "externalUser_Ext",
        "description": "Whether the user is internal (is an employee of the insurer) or not",
        "type": "boolean",
        "readOnly": true
      }
    }
  }
}
```

Schema property types (values stored in the database)

The following table identifies the different ways to specify `type` (and `format`, if necessary) for the data types supported by the base configuration data model. If a give `type` has no required or optional `format`, then the cell in the final column is blank.

Data model column type	Java object type	Schema `type`	Schema `format`
`bit`	`Boolean`	`boolean`	
`CurrencyAmount`		(see below)	
`dateonly`	`Date`	`string`	`date` (required)
`datetime`	`Date`	`string`	`date-time` (required)
`decimal`	`BigDecimal`	`string`	`gw-bigdecimal` (required)
`integer`	`Integer`	`integer`	`int32` (optional)
`longint`	`Long`	`integer`	`int64` (required)
`longtext`	`String`	`string`	
`mediumtext`	`String`	`string`	
`MonetaryAmount`		(see below)	
`money`	`BigDecimal`	`string`	`gw-bigdecimal` (required)
`percentage`	`Integer`	`integer`	`int32` (optional)
`shorttext`	`String`	`string`	
`text`	`String`	`string`	
`varchar`	`String`	`string`	

CurrencyAmount and MonetaryAmount datatypes

In InsuranceSuite, `CurrencyAmount` (used in ClaimCenter) and `MonetaryAmount` (used in PolicyCenter and BillingCenter) are not scalar datatypes. They are the combination of two values: `amount` and `currency`. For more

information on how to configure schema fields that use these datatypes, see “Adding compound datatypes” on page 39.

Schema property types (dynamically calculated values)

There are some types which are not supported for storage in the database, but they may be returned by entity extensions or other methods. The following table identifies the different ways to specify `type` (and `format`, if necessary) for these data types.

Data model column type	Java object type	Schema type	Schema format
n/a	`BigInteger`	`string`	`gw-biginteger` (required)
n/a	`byte[]`	`string`	`byte` (required; this will be treated as the Base64-encoded value)
n/a	`Double`	`number`	You can optionally specify `double` (optional)
n/a	`Float`	`number`	`float` (required)
n/a	`LocalDate`	`string`	`date` (required)
n/a	`LocalTime`	`string`	`time` (required)

Additional properties that are direct children of the property

Attribute	Description	Example
`readOnly`	Boolean identifying if the field as read-only. (The default is `false`.)	`"readOnly": true`
`x-gw-nullable`	Boolean identifying if the field can be explicitly set to null. (The default is `false`.) See “Making properties required by the database” on page 79 for more information.	`"x-gw-nullable": true`
`x-gw-sinceVersion`	String identifying the first version of the API to include the property	`"x-gw-sinceVersion": "1.1.0"`

For example, suppose the `CustomEntity_Ext` entity has an `ExpirationDate` field. The corresponding resource property is read-only, nullable, and was added in version 1.1.0. The property declaration would be:

```
"definitions": {
  "CustomEntityExt": {
    "properties": {
      "expirationDate": {
        "type": "string",
        "format": "date-time",
        "readOnly": true,
        "x-gw-nullable": true,
        "x-gw-sinceVersion": "1.1.0"
      },
      ...
```

Additional properties declared in the x-gw-extension object

Attribute	Description	Example
`createOnly`	Boolean identifying if the field can be specified only when the object is created. (The default is `false`.)	`"createOnly": true`
`filterable`	Boolean identifying if the `filter` query parameter can be used on this field. In other words, collections can be filtered using this property. (The default is `false`.)	`"filterable": true`
`requiredForCreate`	Boolean identifying if the field must be specified when the object is created. (The	`"requiredForCreate": true`

	default is `false`.) See “Making properties required by the database” on page 79 for more information.	
`sortable`	Boolean identifying if the `sort` query parameter can be used on this field. In other words, collections can be sorted using this property. (The default is `false`.)	`"sortable": true`

For example, suppose the `CustomEntity_Ext` entity has an `SeverityType` field, which is a typekey set to a value in the `SeverityType` typelist. The corresponding resource property must be specified at create time, cannot be modified after creation, and is both filterable and sortable. The property declaration would be:

```
"definitions": {
  "CustomEntityExt": {
    "properties": {
      "severityType": {
        "$ref": "#/definitions/TypeKeyReference",
        "x-gw-extensions": {
        "typelist": "SeverityType",
        "createOnly": true,
        "requiredForCreate": true,
        "filterable": true,
        "sortable": true
      },
      ...
```

The mapping extension file

Each mapping extension file for every API contains a `mappers` section. In the base configuration, this section is typically blank, as shown below.

```
"mappers": { }
```

You can add the name of one or more schemas to the `mappers` section. In each schema, you can define a `properties` section, which lists one or more properties defined in the schema.

For each property, you must identify how data is mapped from the Guidewire data model to the property. This is done using a `path` attribute.

For example, in the base configuration data model, the `User` entity has a `JobTitle` field and an `ExternalUser` field. The Admin API has `/user` endpoints, but `JobTitle` and `ExternalUser` are not exposed in the Cloud API `User` resource. If you wanted to expose them, you would first add them to the schema. Then, you would add the following to the `admin_ext-1.0.mapping.json` extension file.

```
"mappers": {
  "User": {
    "properties": {
      "jobTitle_Ext": {
        "path": "User.JobTitle"
      },
      "externalUser_Ext": {
        "path": "User.ExternalUser"
      }
    }
  }
}
```

Mappers for custom entities

If you are adding a custom entity, then in addition to the business properties you wish to expose, the mapper must also contain an "id" property that maps to the data model entity's `RestId`.

For example, suppose you are defining a mapper for a custom entity named `CustomEntityExt` with the following business properties: `isActive`, `customDescription`, `expirationDescription`. The mapper would be as follows:

```
"mappers": {
  "CustomEntityExt": {
    "properties": {
```

```
      "id": {
        "path": "CustomEntity_Ext.RestId"
      },
      "isActive": {
        "path": "CustomEntity_Ext.IsActive"
      },
      "customDescription": {
        "path": "CustomEntity_Ext.CustomDescription"
      },
      "expirationDescription": {
        "path": "CustomEntity_Ext.ExpirationDate"
      }
    }
  }
}
```

Note: Within the Guidewire data model, every entity has a virtual field named `DisplayName`. An *entity name* is a Gosu expression that determines the value for an entity's `DisplayName` field. For example, the `ABPerson` entity might have its entity name set to `ABPerson.LastName + ", " + ABPerson.FirstName`, which for a given `ABPerson` would render as "Newton, Ray". If an entity has no defined entity name, the default behavior is to return the concatenation of every field in the entity.

If you add a property to a schema that maps to an entity's display name, be sure that there is an entity name defined for that entity. If there is not, then the application will return a concatenation of every field in the entity. This could potentially make information available in Cloud API that you do not want exposed through Cloud API.

For more information on entity names, see the *Configuration Guide*.

The updater extension file

Each updater extension file for every API contains a `updaters` section. In the base configuration, this section is typically blank, as shown below.

```
"updaters": { }
```

You can add the name of one or more schemas to the `updaters` section. In each schema, you can define a `properties` section, which lists one or more properties defined in the schema.

For each writeable property, you must identify how data is mapped from the property to the Guidewire data model. Similar to the mapping file, this is done using a `path` attribute. For scalars, it is not unusual for a given property to have the same `path` value on both the mapping file and the updater file.

For example, in the base configuration data model, the `User` entity has a `JobTitle` field and an `ExternalUser` field. The Admin API has `/user` endpoints, but `JobTitle` and `ExternalUser` are not exposed in the Cloud API `User` resource. Suppose you want to expose these fields to Cloud API, and you want to make `JobTitle` writeable and `ExternalUser` read-only. You would first add these properties to the schema and the mapping file. Then, you would add the `JobTitle` property to the `admin_ext-1.0.updater.json` extension file. (`ExternalUser` is omitted from the updater file because it is read-only.

```
"updaters": {
  "User": {
    "properties": {
      "jobTitle_Ext": {
        "path": "User.JobTitle"
      }
    }
  }
}
```

chapter 8

Adding compound datatypes

InsuranceSuite includes several datatypes where multiple values are stored as a unit. This includes the following:

- `Typekey` (containing a `code` and a `name`)
- `MonetaryAmount` and `CurrencyAmount` (containing a `currency` and an `amount`)
- `SpatialPoint` (containing a `longitude` coordinate and a `latitude` coordinate)

For example, an activity's `assignmentStatus` property is a typekey. Thus, the response payload for an activity's assignment status has two sub-fields (`code` and `name`):

```
"assignmentStatus": {
  "code": "assigned",
  "name": "Assigned"
}
```

Use the schema, mapping, and updater extension files to add a compound datatype to a resource.

Filename	Guidewire Studio node
<API>`_ext-1.0.schema.json`	integration -> schemas -> ext -> *<API>*.v1
<API>`_ext-1.0.mapping.json`	integration -> mappers -> ext -> *<API>*.v1
<API>`_ext-1.0.updater.json`	integration -> updaters -> ext -> *<API>*.v1

Note: See also “Tutorial: Schema configuration with compound datatypes” on page 69

The schema extension file

Each schema extension file for every API contains a `definitions` section. In the base configuration, this section is typically blank. You can add the name of one or more schemas to the `definitions` section. In each schema, you can define a `properties` section, which lists one or more properties defined for the schema.

For compound values, each property must also have a `$ref`. It specifies an existing definition for the datatype.

For example, the mapping for the Contact data model entity's `SpatialPoint` field (which is a spatial point) looks like this:

```
  "properties": {
    "spatialPoint": {
      ...
      "$ref": "#/definitions/SpatialPoint"
    }
  }
```

Compound datatype definitions

Set the `$ref` attribute to a datatype definition from the following table.

Compound datatype	$ref value	Additional required attributes
`Typekey`	`#/definitions/TypeKeyReference`	A `x-gw-extensions` attribute with a `typelist` child attribute that identifies the relevant typelist.
`MonetaryAmount`	`#/definitions/MonetaryAmount`	
`CurrencyAmount`	`#/definitions/MonetaryAmount` (Note that there is no `CurrencyAmount` schema. This datatype uses the `MonetaryAmount` schema.)	
`SpatialPoint`	`#/definitions/SpatialPoint`	

If you are defining a typekey property, you must also define an `x-gw-extensions` property with a child `typelist` property set to the name of the typelist. For example:

```
    "properties": {
      "assignmentStatus": {
        ...
        "$ref": "#/definitions/TypeKeyReference",
        "x-gw-extensions": {
          "typelist": "AssignmentStatus"
        }
      }
    }
```

The mapping extension file

Each mapping extension file for every API contains a `mappers` section. In the base configuration, this section is typically blank. You can add the name of one or more schemas to the `mappers` section. In each schema, you can define a `properties` section, which lists one or more properties defined in the schema.

For each property, you must identify how data is mapped from the Guidewire data model to the property. This is done using the following attributes:

- `path` - defines the path to the data model field that stores the data
- `mapper` - names a mapper that defines how the compound data type is mapped into its schema

The following table lists the mapper attribute values for the common compound datatypes.

Compound datatype	mapper value
`Typekey`	`#/mappers/TypeKeyReference`
`MonetaryAmount`	`#/mappers/MonetaryAmount`
`CurrencyAmount`	`#/mappers/CurrencyAmount`
`SpatialPoint`	`#/mappers/SpatialPoint`

For example, the mapping for the `Activity` data model entity's `AssignmentStatus` field looks like this:

```
"mappers": {
  "Activity": {
    "properties": {
      "assignmentStatus": {
        "path": "Activity.AssignmentStatus",
        "mapper": "#/mappers/TypeKeyReference"
      }
    }
  }
}
```

The updater extension file

Each updater extension file for every API contains a `updaters` section. In the base configuration, this section is typically blank. You can add the name of one or more schemas to the `updaters` section. In each schema, you can define a `properties` section, which lists one or more properties defined in the schema.

For each writeable property, you must identify how data is mapped from the Guidewire data model to the property. This is done using the following attributes:

- `path` - defines the path to the data model field that stores the data
- `valueResolver` - names a mapper that defines how the compound data type is mapped into its schema

The syntax for compound datatype properties is:

```
"<property>": {
  "path": "<pathValue>",
  "valueResolver": {
    "typeName": "<resolverName>"
  }
}
```

The following table lists the resolver names for the common compound datatypes.

Compound datatype	Mapper value
`Typekey`	`TypeKeyValueResolver`
`MonetaryAmount`	`MonetaryAmountValueResolver`
`CurrencyAmount`	`CurrencyAmountValueResolver`
`SpatialPoint`	`SpatialPointValueResolver`

For example, the updater for the `Activity` resource's `assignmentStatus` property looks like this:

```
"updaters": {
  "Activity": {
    "properties": {
      "assignmentStatus": {
        "path": "Activity.AssignmentStatus",
        "valueResolver": {
          "typeName": "TypeKeyValueResolver"
        }
      },
      ...
```

If you are using the TypeKeyValueResolver, you can optionally include a property `allowRetired`. If set to `true`, retired entries in the typelist can still be used. The `valueResolver` field would look look like this:

```
"valueResolver": {
  "typeName": "TypeKeyValueResolver",
  "allowRetired": false
}
```

chapter 9

Adding foreign keys

A *foreign key property* is a property that maps to another object. For example, every activity has an assigned user. In the data model, the `Activity` entity has an `AssignedUser` field. This is a foreign key that associated each activity with its assigned user. Similarly, in the Cloud API `Activity` resource, there is an `assignedUser` foreign key property.

Restrictions and limitations

You can add foreign key fields to a Cloud API resource if the foreign key entity is any of the following:

- A base configuration entity with a set of CRUD endpoints in the base configuration of Cloud API
- A base configuration entity with a set of CRUD endpoints generated by the REST endpoint generator
- A custom entity with a set of CRUD endpoints generated by the REST endpoint generator

However, you cannot add foreign key fields to a Cloud API resource if all of the following are true:

- The foreign key entity is a base configuration entity with no CRUD endpoints.
- The REST endpoint generator does not allow generation of endpoints for that entity.

For example, you cannot add a foreign key field to a resource if the foreign key points to the `Credential` entity. This is because there are currently no CRUD endpoints in Cloud API for `Credential`, and the REST endpoint generator will not generate endpoints for `Credential`.

Tools for configuring foreign keys

When you configure a schema, you typically modify three files:

- The schema extension file, which defines the structure of the property
- The mapping extension file, which defines how data maps from the database to the property
- The updater extension file, which defines how data maps from the property to the database

Schema configuration for foreign keys make use of several tools, one for each of these files:

- The schema extension file references the `SimpleReference` schema
- The mapping extension file references the `ResourceReference` mapper
- The updater extension file references one of the value resolvers

In some cases, you may also need to explicitly define an URI Mapping.

The SimpleReference schema

The *SimpleReference schema* is a schema used to simplify how information from other resources is embedded in a schema. The `SimpleReference` schema has the following fields, which are common to all resources:

- `displayName`
- `id`
- `jsonPath`
- `refid`
- `type`
- `uri`

Whenever you add a foreign key property to a schema extension file, the property's definition uses the `SimpleReference` schema.

The ResourceReference mapper

The *ResourceReference mapper* is a mapper that maps information from an entity into the fields defined by the `SimpleReference` schema. Whenever you add a foreign key property to a mapping extension file, the property's mapper references the `ResourceReference` mapper.

Value resolvers

A *value resolver* is a class used in updaters to return the resource that a foreign key property points to. Within the context of a foreign key property:

- The *root* (or the *root resource*) is the resource that has the foreign key property.
- The *resolved value* is the resource that the foreign key points to.

For example, suppose the `Activity` resource has a foreign key field named `AssignedUser` that points to a `User` resource. In this case:

- The *root* is `Activity`.
- The *resolved value* is the `User` that `Activity.AssignedUser` points to.

The term *resolving the reference* refers to the act of determining the resource that the foreign key property points to.

When adding a foreign key property to an updater extension file, the property's updater references a value resolver.

Cloud API provides multiple value resolvers. Some of them are for a specific base configuration entity. Others provide more generic functionality. Some of them are abstract, in which case you must extend the value resolver for a specific type of entity. The following is a brief overview of each resolver. The following topics go into greater detail about when and how to use each resolver.

Entity-specific value resolvers

An *entity-specific value resolver* is a base configuration value resolver that resolves values for a specific type of base configuration entity. For example, the `UserJsonValueResolver` can be used for foreign key fields that point to instances of `User`.

You can find a list of entity-specific value resolvers in Studio by executing a **Navigate** -> **File** command (CTRL + SHIFT + N) and entering "valueresolver".

Be aware that some entity-specific value resolvers have special behaviors to enable specific Cloud API use cases. These behaviors may not be appropriate for custom foreign key properties. Whenever you use an entity-specific value resolver, Guidewire recommends testing the associated PATCH and POST behaviors thoroughly.

For more information on the use cases for entity-specific value resolvers, see “Updater case 1: Root and resolved value have no common ancestor” on page 49 and “Updater case 2: Root and resolved value have a common ancestor” on page 51.

KeyableBeanJsonValueResolver

The `KeyableBeanJsonValueResolver` is a generic value resolver that can be used when either:

- There is no entity-specific value resolver for the entity the foreign key references.
- There is an entity-specific value resolver for the entity the foreign key references, but it has behaviors that are not appropriate for the custom foreign key.

The `KeyableBeanJsonValueResolver` is declared in the `gw.rest.core.pl.framework.v1.refs` package.

To use this resolver, the schema property for the foreign key must include a `x-gw-extensions.resourceType` field that identifies the type of resource the foreign key points to. For example, if you had a `backupUser_Ext` foreign key property on `Activity` that points to `User`, then the schema property declaration would be as follows:

```
"Activity": {
  "properties": {
    "backupUser_Ext": {
      ...
      "x-gw-extensions": {
        "resourceType": "User"
      }
    }
  }
```

Also, in the updater, the path attribute must resolve to a `KeyableBean`.

By default, this updater supports both reference by `id` and `refid`. You can disable referencing by `refid`.

This resolver is final and cannot be subclassed. If you need a KeyableBean value resolver with behavior that can be subclassed, use `AbstractKeyableBeanJsonValueResolver`.

For more information on using the `KeyableBeanJsonValueResolver`, see “Updater case 1: Root and resolved value have no common ancestor” on page 49 and “Updater case 2: Root and resolved value have a common ancestor” on page 51.

AbstractKeyableBeanJsonValueResolver

The `AbstractKeyableBeanJsonValueResolver` is also a generic value resolver. It is abstract and therefore is never used directly. But it can be extended, and the subclass resolver can apply to any entity.

Subclassing this resolver gives you the ability to specify "isEntityViewable logic". This logic can control access to an object based on business-specific conditions of the object. For example, this logic could mandate that a foreign key property to `User` can reference only users with the "Manager" role.

For more information on using the `AbstractKeyableBeanJsonValueResolver`, see “Update case 3: Accessibility of resolved value is conditional” on page 52.

CollectionBasedJsonValueResolver

> **Note:** This value resolver is theoretically applicable in all applications. However, the primary use case for this value resolver occurs most often in PolicyCenter.

In some situations, the resolved value cannot be determined using id only. In these situations, the correct resolved value can be selected only by selecting a KeyableBean from a collection contained on some ancestor entity. The most common situation that this occurs is when resolving effdated entities.

For more information on using the `CollectionBasedJsonValueResolver`, see “Update case 4: Resolved value cannot be easily resolved by id alone” on page 53.

The URI Mapping

A *URI mapping* is an entry that defines, for a given resource type, the resource's URI and its parent. If a schema defines one or more foreign keys, the root resource needs an URI Mapping.

URI mappings exist for all base configuration resource types in Cloud API. URI mappings are also generated automatically if you create endpoints using the REST endpoint generator and you add the new resource to an integration graph. Therefore, the only time you need to a URI mapping manually is if you have a resource with one or more foreign keys, you generated the resource using the REST endpoint generator, and you did not add the new resource to an integration graph.

Foreign keys in the schema configuration files

When you configure a schema, you typically modify the following files:

- The schema extension file
  - The property definition references the `SimpleReference` schema
- The mapping extension file
  - The mapper references the `ResourceReference` mapper
- The updater extension file
  - The updater references a value resolver
  - In some cases, the referenced value resolver is one of the base configuration value resolvers
  - In other cases, you must first extend one of the abstract value resolvers, and then reference your extension value resolver

If you created the root resource through the REST endpoint generator and did not add the resource to an integration graph, you must also update the shared apiconfig extension file.

The following table summarizes the names and locations of these files.

Filename	Guidewire Studio node
`<API>_ext-1.0.schema.json`	integration -> schemas -> ext -> <*API*>.v1
`<API>_ext-1.0.mapping.json`	integration -> mappers -> ext -> <*API*>.v1
`<API>_ext-1.0.updater.json`	integration -> updaters -> ext -> <*API*>.v1
`shared_ext-1.0.apiconfig.yaml`	integration -> apis -> ext -> shared.v1

The schema extension file

Each schema extension file for every API contains a `definitions` section. In the base configuration, this section is typically blank. You can add the name of one or more schemas to the `definitions` section. In each schema, you can define a `properties` section, which lists one or more properties defined for the schema.

For a foreign key property, the schema definition must specify `$ref` and `resourceType`.

The $ref property

`$ref` must be set to `#/definitions/SimpleReference`. `SimpleReference` refers to the *`SimpleReference` schema*, which is a schema used to simplify how foreign key information is embedded in a schema. The `SimpleReference` schema has the following fields, which are common to all resources:

- `displayName`
- `id`
- `jsonPath`

- `refid`
- `type`
- `uri`

The resourceType property

`resourceType` is a child property of the `x-gw-extensions` property. `resourceType` must be set to the name of the associated resource.

The syntax for setting these values is:

```
  "definitions": {
    "<resourceName>": {
      "properties": {
        "<foreignKeyPropertyName>": {
          "$ref": "#/definitions/SimpleReference",
          "x-gw-extensions": {
            "resourceType": "<resourceThatForeignKeyPointsTo>"
          }
        }
      }
    }
  }
```

Example schema definition

For example, suppose you want to designate a backup user for each activity. In the data model, you extend the `Activity` entity with a foreign key named `BackupUser_Ext` that points to the `User` entity. You also want to provide access to this `BackupUser_Ext` field in Cloud API. To do this, you would add the following to the `common_ext-1.0.schema.json` file:

```
  "definitions": {
    "Activity": {
      "properties": {
        "backupUser_Ext": {
          "title": "BackupUser_Ext",
          "description": "The backup user for the activity",
          "$ref": "#/definitions/SimpleReference",
          "x-gw-extensions": {
            "resourceType": "User"
          }
        }
      }
    }
  }
}
```

The mapping extension file

Each mapping extension file for every API contains a `mappers` section. In the base configuration, this section is typically blank. You can add the name of one or more schemas to the `mappers` section. In each schema, you can define a `properties` section, which lists one or more properties defined in the schema.

When adding a foreign key property to a resource, the mapping file must specify a `path` and `mapper`.

The `path` attribute identifies the field from the data model entity that stores the foreign key value. It also uses an extension named `RestV1_AsReference`. This extension provides information on how to map the various `SimpleReference` schema properties, such as `jsonPath` and `uri`. The syntax for the `path` attribute is:

```
<pathToForeignKeyField>.RestV1_AsReference
```

The `mapper` attribute is set to `"#/mappers/ResourceReference"`. The `ResourceReference` mapper maps information from an entity into the fields defined by the `SimpleReference` schema.

The syntax for setting these values is:

```
    "<resourceName>": {
      "properties": {
        "<foreignKeyPropertyName>": {
          "path": "<pathToForeignKeyField>.RestV1_AsReference",
```

```
          "mapper": "#/mappers/ResourceReference"
        }
      }
    }
```

For example, suppose you want to implement the mapping for the `BackupUser_Ext` extension added to the `Activity` entity in the previous example. To do this, you would add the following to the `common_ext-1.0.mapping.json` file:

```
  "mappers": {
    "Activity": {
      "properties": {
        "backupUser_Ext": {
          "path": "Activity.BackupUser_Ext.RestV1_AsReference",
          "mapper": "#/mappers/ResourceReference"
        }
      }
    }
  }
```

The updater extension file

Each updater extension file for every API contains a `updaters` section. In the base configuration, this section is typically blank. You can add the name of one or more schemas to the `updaters` section. In each schema, you can define a `properties` section, which lists one or more properties defined in the schema.

In the updater file, you must specify a `path` and a `valueResolver`.

The `path` attribute specifies the path from the root entity to the foreign key field. For example, to specify the updater for the `BackupUser_Ext` field from the previous example, you would add the following to the updater file:

```
"Activity": {
  "properties": {
    "backupUser_Ext": {
      "path": "Activity.BackupUser_Ext",
      ...
```

A *value resolver* is a class used in updaters to return the resource that a foreign key property points to. The resource that the foreign key points to is sometimes referred to as the *resolved value*. The term *resolving the reference* refers to the act of determining the resource that the foreign key property will point to.

Cloud API provides value resolvers for some base configuration entities. It also provides a set of more generic value resolvers. The best value resolver to use depends on the type of relationship the root resource has with the foreign key resource. There are several possible relationships:

- “Updater case 1: Root and resolved value have no common ancestor” on page 49
- “Updater case 2: Root and resolved value have a common ancestor” on page 51
- “Update case 3: Accessibility of resolved value is conditional” on page 52
- “Update case 4: Resolved value cannot be easily resolved by id alone” on page 53

The shared apiconfig file

The `shared_ext-1.0.apiconfig.yaml` maps shared element resource and collection resources to Gosu Resource files. It also specifies URI mappings for integration graphs. When you specify that you want to add a custom resource to a graph, the REST endpoint generator adds code to the corresponding `shared_ext-1.0.apiconfig.yaml` file.

If you do not add a resource to an integration graph, then you must manually add the URI mapping to the `shared_ext-1.0.apiconfig.yaml`. The syntax for the entry is as follows:

```
entityURIMappings:
 <resourceName>:
  uri: "${parentUri}/<endpointPathEnd>/${<pathToRestId>}"
  parent: "<pathFromRootResourceToParent>"
```

For example, suppose you had a custom entity that was not part of an integration graph. Its name is `CustomEntity_Ext`, the end of its endpoint path is `custom-entities-ext`, and its parent is `Activity`. The URI mapping would look like this:

```
entityURIMappings:
 CustomEntity_Ext:
  uri: "${parentUri}/custom-entities-ext/${CustomEntity_Ext.RestId}"
  parent: "CustomEntity_Ext.Activity"
```

Updater case 1: Root and resolved value have no common ancestor

In some cases, the root entity and the resolved value do not share a common ancestor. This typically occurs when one or both entities are not part of any graph, such as the claim graph or policy graph.

For example, suppose you add a foreign key from `Activity` to `User` to track an activity's backup user. In this situation, there is no entity (such as a claim or an account) that owns both the activity and the user.

Using the KeyableBeanJsonValueResolver

When the two entities do not share a common parent, you can use the `KeyableBeanJsonValueResolver` (`gw.rest.core.pl.framework.v1.refs.KeyableBeanJsonValueResolver`). For example:

```
"updaters": {
  "Activity": {
    "properties": {
      "backupUser_Ext": {
        "path": "Activity.BackupUser_Ext",
        "valueResolver": {
          "typeName": "gw.rest.core.pl.framework.v1.refs.KeyableBeanJsonValueResolver"
        }
      }
    }
  }
}
```

Disabling reference by refid

By default, the `KeyableBeanValueResolver` supports both reference by id and `refid`. You can disable reference by `refid` by adding the following code to the `valueResolver` property:

```
"setByRefid": false
```

For example, the following updater for `backupUser_Ext` disables reference by `refid`.

```
"updaters": {
  "Activity": {
    "properties": {
      "backupUser_Ext": {
        "path": "Activity.BackupUser_Ext",
        "valueResolver": {
          "typeName": "gw.rest.core.pl.framework.v1.refs.KeyableBeanJsonValueResolver",
          "setByRefid:" false
        }
      }
    }
  }
}
```

Using an entity-specific value resolver

Cloud API also provides value resolvers for specific base configuration entities. For example, there is a `UserJsonValueResolver` for foreign key fields that reference the `User` entity. You could use this in place of the `KeyableBeanJsonValueResolver`. For example:

```
"updaters": {
  "Activity": {
    "properties": {
```

```
      "backupUser_Ext": {
        "path": "Activity.BackupUser_Ext",
        "valueResolver": {
          "typeName": "gw.rest.core.pl.admin.v1.group.UserJsonValueResolver"
        }
      }
    }
  }
}
```

You can find a list of entity-specific value resolvers in Studio by executing a **Navigate** > **File** command (CTRL + SHIFT + N) and entering "valueresolver". Be aware that the entity-specific value resolvers may have special behaviors to enable common Cloud API use cases. These behaviors may not be appropriate for an updater to a custom foreign key. Whenever you use a entity-specific value resolver, Guidewire recommends testing the associated PATCH and POST behaviors thoroughly.

Complete code sample for case 1

The following code snippets define a new foreign key property where the root and foreign key entities have no common ancestor. In this example, the `Activity` resource has a `backupUser_Ext` property that references the `User` entity.

Data model extension

File name: `Activity.etx`

```
<extension
  xmlns="http://guidewire.com/datamodel"
  entityName="Activity">
    ...
    <foreignkey
      fkentity="User"
      name="BackupUser_Ext"
      nullok="true"/>
    ...
```

Schema file extension

File name: `common_ext-1.0.schema.json`

```
"definitions": {
  "Activity": {
    "properties": {
      "backupUser_Ext": {
        "title": "BackupUser_Ext",
        "description": "The backup user who can complete the activity if the assigned user is on vacation",
        "$ref": "#/definitions/SimpleReference",
        "x-gw-extensions": {
          "resourceType": "User"
        }
      },
      ...
```

Mapping extension file

File name: `common_ext-1.0.mapping.json`

```
"mappers": {
  "Activity": {
    "properties": {
      "backupUser_Ext": {
        "path": "Activity.BackupUser_Ext.RestV1_AsReference",
        "mapper": "#/mappers/ResourceReference"
      },
```

Updater extension file

File name: `common_ext-1.0.updater.json`

(This example assumes use of the entity-specific `UserJsonValueResolver`, and reference by `refid` is allowed.)

```
"updaters": {
  "Activity": {
```

```
    "properties": {
      "backupUser_Ext": {
        "path": "Activity.BackupUser_Ext",
        "valueResolver": {
          "typeName": "gw.rest.core.pl.admin.v1.group.UserJsonValueResolver"
        }
      },
```

Updater case 2: Root and resolved value have a common ancestor

In some cases, the root entity and the resolved value have a common ancestor. This typically occurs when both entities are part of a graph, such as the claim graph or policy graph, and you want to ensure that both entities share the same ancestor. (In other words, both entities are owned by the same claim or the same policy.)

For example, suppose underwriters periodically upload documents to a customer portal. To manage this work, you have implemented "Upload to portal" activities. You want each activity to references the document to upload. In the `Activity` entity, you have created a `DocumentToUpload_Ext` field that points to the `Document` entity. However, you do not want to let a caller application associate any document in PolicyCenter with a given activity. Both the activity and the document must belong to the same account. In other words, they must have a common ancestor.

When the two entities share a common ancestor, you can use the `KeyableBeanJsonValueResolver` (`gw.rest.core.pl.framework.v1.refs.KeyableBeanJsonValueResolver`). However you must add two additional fields to the updater:

- `resolvedValueToAncestorPath` - This specifies the path that traverses from the foreign key entity to the common ancestor.
  - The path starts with the `resolvedValue` symbol, which represents the foreign key entity.
  - The path ends with the common ancestor, which may be any number of levels away.
  - For example, `resolvedValue.Account`.
- `rootToAncestorPath` - This specifies the path that traverses from the root entity to the common ancestor.
  - The path starts with the name of the root resource.
  - The path ends with the common ancestor, which may be any number of levels away.
  - For example, `Activity.Account`.

For example, the updater for the "Upload to portal" activity example would be:

```
"updaters": {
  "Activity": {
    "properties": {
      "documentToUpload_Ext": {
        "path": "Activity.DocumentToUpload_Ext",
        "valueResolver": {
          "typeName": "gw.rest.core.pl.framework.v1.refs.KeyableBeanJsonValueResolver"
          "resolvedValueToAncestorPath": "resolvedValue.Account",
          "rootToAncestorPath": "Activity.Account"
        }
      }
    }
  }
}
```

If you specify `resolvedValueToAncestorPath` but do not specify `rootToAncestorPath`, then the root of the updater will be used as the ancestor to match.

Cloud API also provides value resolvers for specific base configuration entities. These resolvers may automatically implement common ancestor validation.

You can find a list of entity-specific value resolvers in Studio by executing a **Navigate** > **File** command (CTRL + SHIFT + N) and entering "valueresolver". Be aware that the entity-specific value resolvers may have special behaviors to enable common Cloud API use cases. These behaviors may not be appropriate for an updater to a custom foreign

key. Whenever you use a entity-specific value resolver, Guidewire recommends testing the associated PATCH and POST behaviors thoroughly.

Validation when root and resolved value ancestors do not match

For this type of updater, if a caller application attempts to set a foreign key field to an object that does not share a common ancestor, an error similar to the following is returned:

```
data.attributes.documentToUpload_Ext.id - The id 'cc:Suyx2KNC82n2MiAXGml8M' is not
a valid 'Document'"
```

Update case 3: Accessibility of resolved value is conditional

Making access to a resolved object conditional

In some cases, a foreign key field can reference a given object only if the object meets certain criteria. For example, suppose that the `Activity` entity has an `ApprovalManager_Ext` field. This field identifies the manager of the activity, who is responsible for addressing the situation when an activity is still open past its due date. The `ApprovalManager_Ext` field is set to a `User`, but it can only be set to a `User` who has the "Manager" role.

When you configure a foreign key in Cloud API, you can specify this type of conditional foreign key logic. This is done in a layer of logic known as "`isEntityViewable` logic". This optional logic specifies that you can set a foreign key property to an object only if the object meets certain conditions.

The `KeyableBeanJsonValueResolver` does not have any `isEntityViewable` logic. If you want to implement `isEntityViewable` logic for a custom foreign key, you must use the `AbstractKeyableBeanJsonValueResolver`.

The AbstractKeyableBeanJsonValueResolver

The `AbstractKeyableBeanJsonValueResolver` is abstract and therefore is never used directly. If you want to specify `isEntityViewable` logic, you must implement a concrete subclass of this resolver. The subclass has an `isEntityViewable` method where you can specify your logic. You then reference this subclass in the appropriate updater file.

Extending the AbstractKeyableBeanJsonValueResolver

When you extend the `AbstractKeyableBeanJsonValueResolver`, do the following.

- Guidewire recommends putting the class in the `gw.rest.ext.pc.<api>.v1` package. Guidewire also recommends putting the class in a sub-package named after the resolved value. For example, a resolver used for foreign keys that reference `User` instances would be in the `gw.rest.ext.pc.common.v1.users` package.
- Guidewire recommends you name the class `<ResolvedEntity>ExtJsonValueResolver`.
- The class must extend `AbstractKeyableBeanJsonValueResolver<V>`, where "`V`" is the entity type of the resolved value. (For example, `AbstractKeyableBeanJsonValueResolver<User>`.)
- The class must include the `@NotNull` annotation.
- The class must override the `ResolvedValueType` method.
  - The method declaration must return `Class<V>`, where "`V`" is the entity type of the resolved value. (For example, `Class<User>`.)
  - The method must return `V`. (For example, `return User`)
- The class must override the `isEntityViewable` method.
  - The method declaration's input parameter must be set to `keyableBean: V`, where "`V`" is the entity type of the resolved value. (For example, `keyableBean : User`.)
  - The method must return a Boolean value.
    - Returning true indicates the resolved value is accessible, and the corresponding foreign key property can be set to it.

- Returning false indicates the resolved value is not accessible, and the corresponding foreign key property cannot be set to it.

For example, suppose that the Activity entity has an `ApprovalManager_Ext` field. This field identifies the manager of the activity, who is responsible for addressing the situation when an activity is still open past its due date. The `ApprovalManager_Ext` field is set to a `User`, but it can only be set to a user who has the "Manager" role. The value resolver for this business requirement would be as follows.

```
package gw.rest.ext.xc.common.v1.users

uses gw.rest.core.pl.framework.v1.refs.AbstractKeyableBeanJsonValueResolver
uses org.jetbrains.annotations.NotNull

class UserExtJsonValueResolver extends AbstractKeyableBeanJsonValueResolver<User> {

  @NotNull
  protected override property get ResolvedValueType() : Class<User> {
    return User
  }

  protected override function isEntityViewable(keyableBean : User) : boolean {
    return keyableBean.Roles.hasMatch(\role -> role.Role.Name == "Manager")
  }
}
```

Extending the updater

The mapper must have a `valueResolver` property whose `typeName` is set to the concrete subclass of `AbstractKeyableBeanJsonValueResolver`. For example, the updater from the previous example would be as follows:

```
"updaters": {
  "Activity": {
    "properties": {
      ...
      "activityManager_Ext": {
        "path": "Activity.ActivityManager_Ext",
        "valueResolver": {
          "typeName": "gw.rest.ext.cc.common.v1.users.UserExtJsonValueResolver"
        }
      }
```

Validation when the foreign key resource is not accessible

If you try to set a foreign key property to a resource, and the value resolver's `isEntityViewable` method returns false, an error similar to the following is returned:

```
data.attributes.activityManager_Ext.id - The id 'demo_sample:2' is not a valid 'User'",
```

Using isEntityViewable logic with common ancestors

You can create a foreign key field that makes use of `isEntityViewable` logic and that requires the foreign key entity to share a common ancestor. To do this:

1. Implement a concrete subclass of `AbstractKeyableBeanJsonValueResolver`.
2. Add the following to your updater declaration:
   a. `typename` set to your value resolver.
   b. `resolvedValueToAncestorPath` set to the path from the foreign key entity to the common ancestor
   c. `rootToAncestor` set to the path from the root entity to the common ancestor

Update case 4: Resolved value cannot be easily resolved by id alone

In some situations, the target of a foreign key field cannot be easly determined using id only. There are several reasons this could occur:

- The entity might be effdated. It is difficult to load effdated entities using id alone.
- The id might be unique only within the context of its parent. For example, a coverage's id is a coverage pattern, but the same coverage (with the same id) can exist under multiple vehicles or buildings on the same policy. The id alone does not uniquely identify the desired coverage.
- The id might be computed, which would make it possible but impractical to load the entity directly from the database. For example, for account contacts, the id is the id of the linked Contact entity. Querying on the AccountContact is possible, but it would require at least a join on the Contact table.
- The id might not related to an entity. For example, the Cloud API PolicyContact maps to a PolicyContactWrapper, which is a POGO object and not a database entity.

For example, suppose you are using the Personal Auto line of business in the base configuration of PolicyCenter. In this line, there is a `PersonalVehicle` entity which has an array of `VehicleDrivers`. You want to be able to identify one of the drivers as the "primary driver" of the vehicle. To do this, you add a `PrimaryDriver_Ext` edge foreign key to `PersonalVehicle` that points to the primary driver. However, `VehicleDriver` is an effdated entity, and therefore the correct object to reference cannot be loaded from the bundle or database using only its id.

The `KeyableBeanJsonValueResolver` resolves foreign keys using only the target's id. This will not work for the circumstances mentioned in the previous list. To load these types of entities, you must use the `CollectionBasedJsonValueResolver`.

The CollectionBasedJsonValueResolver

The `CollectionBasedJsonValueResolver` can identify targets when the target cannot be easily identified by id alone, and when necessary, it can ensure that the root object and target object share a common ancestor. It has two getters:

- `get ResolvedValueType` - Returns the type of the target object
- `get AncestorType` - Returns the type of the common ancestor

It also has two methods for identifying the target:

- `getPossibleResolvedValues` - Returns the collection that contains the target object
- `idMatchExpression` - Returns a predicate that, when run on the collection, will identify the target

The `CollectionBasedJsonValueResolver` is abstract and therefore is never used directly. If you want to use it, you must implement a concrete subclass of this resolver. In the subclass, you must override the previously mentioned getters and methods. You then reference this subclass in the appropriate updater file.

Extending the CollectionBasedJsonValueResolver

When you extend the `CollectionBasedJsonValueResolver`, do the following:

- Guidewire recommends putting the class in an appropriate package in the `gw.rest.ext.<product>` package. Guidewire also recommends naming the package after the target entity. For example, a resolver used for foreign keys that reference `PersonalDriver` instances could be in the `gw.rest.ext.pc.policyperiod.pa.v1.driver` package.
- Guidewire recommends you name the class `<ResolvedEntity>JsonValueResolver`.
- The class must extend `CollectionBasedJsonValueResolver<V, A>`, where:
  1. "V" is the entity type of the target. (For example, `VehicleDriver`.)
  2. "A" is the entity type of the ancestor that owns the relevant collection. (For example, `PersonalVehicle`.)

The class must include overrides of the following getters and methods using the following syntax:

```
protected override property get ResolvedValueType(): Class<TargetEntityType> {
  return TargetEntityType
}

protected override property get AncestorType(): Class<AncestorEntityType> {
  return AncestorEntityType
}

protected override function getPossibleResolvedValues(ancestor : AncestorEntity) : Collection<TargetEntity> {
```

```
  return ancestor.ArrayContainingPotentialResolvedValue
}

protected override function idMatchExpression(id: String) : Predicate<TargetEntity> {
  return <predicate containing the set of elements with matching ids>
}
```

where:

- *TargetEntityType* is the target entity type (such as `VehicleDriver`)
- *AncestorEntityType* is the ancestor entity type (such as `PersonalVehicle`)
- *ArrayContaininPotentialResolvedValue* is the array that contains the target object (such as *`Drivers`*)

For example, suppose you are implementing the previously mentioned use case where you want to have a `PrimaryDriver_Ext` field on `PersonalVehicle` that points to an instance of `VehicleDriver`. The value resolver for this business requirement would be as follows.

```
package gw.rest.ext.pc.policyperiod.pa.v1.driver

uses gw.rest.core.pl.framework.v1.refs.CollectionBasedJsonValueResolver
uses java.util.function.Predicate

class PADriverJsonValueResolver extends CollectionBasedJsonValueResolver <VehicleDriver, PersonalVehicle> {

  protected override property get ResolvedValueType(): Class<VehicleDriver> {
    return VehicleDriver
  }

  protected override property get AncestorType(): Class<PersonalVehicle> {
    return PersonalVehicle
  }

  protected override function getPossibleResolvedValues(ancestor : PersonalVehicle) : Collection<VehicleDriver> {
    return ancestor.Drivers
  }

  protected override function idMatchExpression(id: String) : Predicate<VehicleDriver> {
    return \elt -> elt.RestV1EffDatedId == id
  }
}
```

The isEntityViewable method

An extension of the `CollectionBasedJsonValueResolver` can also include an `isEntityViewable` method. This method can be used to control whether the resolved value can be viewed. For example, you could use it to enforce business logic that states a foreign key can point to a object only if the object is in a draft or open state.

The implementation of an `isEntityViewable` method for the `CollectionBasedJsonValueResolver` is the same as that of the `AbstractKeyableBeanJsonValueResolver`. For more information, see “Update case 3: Accessibility of resolved value is conditional” on page 52.

Extending the updater

The updater must have a `valueResolver` property whose `typeName` is set to the concrete subclass of `CollectionBasedJsonValueResolver`. For example, the updater from the previous example would be as follows:

```
"updaters": {
  "PersonalVehicle": {
    "properties": {
      ...
      "primaryDriver_Ext": {
        "path": "PersonalVehicle.PrimaryDriver_Ext",
        "valueResolver": {
          "typeName": "gw.rest.ext.pc.policyperiod.pa.v1.driver.PADriverJsonValueResolver",
          "resolvedValueToAncestorPath": "resolvedValue.Vehicle"
        }
      },
      ...
```

With effdated entities, the object that has the foreign key property (the root object) and the object that the foreign key points to (the target object) must have a common ancestor.

- You will always need a `resolvedValueToAncestorPath` property, as shown above, to identify how to map from the target object to the common ancestor.

- If the common ancestor is some object above the root object, then you also need a `rootToAncestorPath` value that identifies how to navigate from the root to the common ancestor.

In this example, the root object (`PersonalVehicle`) is the common ancestor. Hence, the `rootToAncestorPath` is not required.

chapter 10

Adding one-to-ones

A *one-to-one relationship* is a relationship that two entities have. One acts as the parent, and the other as the child. For the parent, each instance may have an association with up to one instance of the child entity. For the child, each instance must be associated with exactly one parent.

The Guidewire data model supports a `<one-to-one>` element. This element enforces the parent's "up-to-one" cardinality and the child's "exactly-one" cardinality. Many one-to-one relationships are implemented using this element. However, in some situations, the one-to-one relationship is built using only a `<foreignKey>` element.

Common use cases

A common use case for one-to-one relationships is to separate extension fields that apply to only a sufficiently small subset of the entities. This is done to avoid database tables that are wide and sparsely populated.

For example, suppose an insurer has a regulatory requirement to have a set of activities periodically reviewed by the insurer's legal team. There are several fields needed to track this legal review, but they are applicable to only about 5% of all activities.

The insurer could add these fields directly to the `Activity` entity. But given that so few activities need these fields, this would make the `xc_activity` table in the database wider and more sparsely populated.

To improve database performance, the insurer opts to create a second entity, `ActivityLegalInfo_Ext`, which has a one-to-one relationship with `Activity`.

- The `Activity` entity is the parent. Each activity can be associated with up to one `ActivityLegalInfo_Ext`.
- `ActivityLegalInfo_Ext` is the child. Each instance must be associated with exactly one `Activity`.

Inline objects: Foreign key fields

In Cloud API, foreign key relationships follow a typical pattern:

- Typically, there are CRUD endpoints for both the parent and the child.
- The child appears in the parent schema using the `SimpleReference` schema. (This schema has a small set of fields applicable to all entities, such as `displayName`, `id`, `type`, and `uri`.)

For example, the `Activity` entity has an `assignedUser` field, which is a foreign key that references `User`.

- There are CRUD endpoints for both `Activity` and `User`.
- In the `Activity` schema, there is an `assignUser` field which includes information about the associated `User`. It uses the `SimpleReference` schema, as shown in the example below.

```
"attributes": {
    "activityPattern": "contact_insured",
```

```
    "assignedUser": {
        "displayName": "Andy Applegate",
        "id": "demo_sample:1",
        "type": "User",
        "uri": "/admin/v1/users/demo_sample:1"
    },
    "id": "cc:SO8RJJZtEa-Tyqlrxw97y",
    "subject": "Contact insured"
},
```

Inline objects: One-to-one fields

One-to-one relationships follow a different pattern:

- Typically, there are CRUD endpoints for the parent only.
  - Because there are no separate endpoints for the child object, all operations on the child take place in the context of the parent. You GET the child along with the parent. When you POST the parent, you can also POST the child.
- The child appears in the parent schema as an inline object. The inline object can include any set of fields from the child entity.

For example, an insurer extends the data model so that the `Activity` entity shared a one-to-one relationship with a custom `ActivityLegalInfo_Ext` entity.

- There are CRUD endpoints for `Activity`.
- There are no CRUD endpoints for `ActivityLegalInfo_Ext`.
- In the `Activity` schema, there is an `ActivityLegalInfo_Ext` field which includes information about the associated `ActivityLegalInfo_Ext`. It is an inline object and includes a custom set of fields from the child entity, as shown in the example below.

```
"attributes": {
    "activityLegalInfo_Ext": {
        "id": "cc:S2qhgxij6HvhXz6K3t39K",
        "legalCaseNumber": "0003",
        "legalReviewDate": "2022-05-05T07:00:00.000Z"
    },
    "activityPattern": "contact_insured",
    "assignedUser": {
        "displayName": "Andy Applegate",
        "id": "demo_sample:1",
        "type": "User",
        "uri": "/admin/v1/users/demo_sample:1"
    },
    "id": "cc:SO8RJJZtEa-Tyqlrxw97y",
    "subject": "Contact insured"
},
```

Example one-to-one

The rest of the topics in this section use the following business example.

Suppose the insurer had a regulatory requirement to have a set of activities periodically reviewed by the insurer's legal team. There are several fields needed to track this legal review, but they are applicable to only about 5% of all activities.

To improve database performance, the insurer opts to store this information in an `ActivityLegalInfo_Ext` entity and define it as a one-to-one with `Activity`.

- The `Activity` entity is the parent. Each activity can be associated with up to one `ActivityLegalInfo_Ext`.
- `ActivityLegalInfo_Ext` is the child. Each instance must be associated with exactly one `Activity`.

To implement this, the following has been added to the data model:

- `Activity` entity
  - `ActivityLegalInfo_Ext`

        - one-to-one
        - fkentity: `ActivityLegalInfo_Ext`
        - nullok: true
- `ActivityLegalInfo_Ext` entity
    - `LegalCaseNumber`
        - varchar with size 30
    - `LegalReviewDate`
        - datetime
    - `Activity`
        - foreign key
        - fkentity: `Activity`
        - nullok: false

Note the following:

- On `Activity`, the `ActivityLegalInfo_Ext` field has nullok set to true. This is because some `Activity` instances will have an associated `ActivityLegalInfo_Ext`, but most will not.
- On `ActivityLegalInfo_Ext`, the `Activity` field has nullok set to false. This is because every `ActivityLegalInfo_Ext` must be associated to an `Activity`.

For a complete list of all code used to build this example, see “Complete code sample for one-to-ones” on page 64.

One-to-one relationships in the schema configuration files

When you configure a schema, you typically modify the following files:

- The schema extension file: This file specifies extensions for two schemas - one for the parent and one for the child.
- The mapping extension file: This file specifies extensions for two mappers - one for the parent and one for the child.
- The updater extension file: This file specifies extensions for two updaters - one for the parent and one for the child.

Filename	Guidewire Studio node
<API>`_ext-1.0.schema.json`	integration -> schemas -> ext -> *<API>*.v1
<API>`_ext-1.0.mapping.json`	integration -> mappers -> ext -> *<API>*.v1
<API>`_ext-1.0.updater.json`	integration -> updaters -> ext -> *<API>*.v1

The schema extension file

Each schema extension file for every API contains a `definitions` section. In the base configuration, this section is typically blank. You can add the name of one or more schemas to the `definitions` section. In each schema, you can define a `properties` section, which lists one or more properties defined for the schema.

For a one-to-one relationship, you must add or modify two schema definitions:

- The parent schema definition must have a property that references the child schema definition. This property must specify a `$ref`.
- You must also add a new schema definition for the child that specifies all the properties to be inlined and a `type` for each property.

The parent schema

In the schema extension file, if the parent schema is not already present, then you must add it to the `definitions` section, and you must add a child `properties` section.

You must define the one-to-one property. The title and description attributes are optional. The `$ref` property is required and it must be set to "`#/definitions/<nameOfChildSchema>`".

For example, the following defines an `activityLegalInfo_Ext` property in the `Activity` schema.

```
"definitions": {
  "Activity": {
    "properties": {
      ...
      "activityLegalInfo_Ext": {
        "title": "ActivityLegalInfo_Ext",
        "description": "One-to-one association to ActivityLegalInfo_Ext",
        "$ref": "#/definitions/ActivityLegalInfo_Ext"
      }
    }
  },
```

The child schema

In the schema definition file, you must add the new schema definition to the `definitions` section for the child. You must add a `properties` section to that definition. Then you must add the following properties:

- A read-only `id` property whose type is `string`.
- One property declaration for each field in the one-to-one child that is to be exposed to Cloud API.

For example, the following defines an `ActivityLegalInfo_Ext` schema, with three properties: `id`, `legalCaseNumber`, and `legalReviewDate`.

```
"definitions": {
  ...
  "ActivityLegalInfo_Ext": {
    "properties": {
      "id": {
        "title": "ID",
        "description": "Object ID",
        "type": "string",
        "readOnly": true
      },
      "legalCaseNumber": {
        "title": "LegalCaseNumber",
        "description": "Legal case number",
        "type": "string"
      },
      "legalReviewDate": {
        "title": "LegalReviewDate",
        "description": "Legal review date",
        "type": "string",
        "format": "date-time"
      }
    }
  }
```

The inline child properties are typically scalars, compound datatypes, or foreign keys. For more information on how to configure schemas for these type of properties, see:

- “Adding scalars” on page 33
- “Adding compound datatypes” on page 39
- “Adding foreign keys” on page 43

The mapping extension file

Each mapping extension file for every API contains a `mappers` section. In the base configuration, this section is typically blank. You can add the name of one or more schemas to the `mappers` section. In each schema, you can define a `properties` section, which lists one or more properties defined in the schema.

For a one-to-one relationship, you must add or modify two mappers:

- The parent mapper must specify the path mapping for the one-to-one property that points to the child.
- You must also add a new mapper for the child that specifies the path mappings for the inline child properties.

The parent mapper

In the mapping file, if the parent mapper is not already present, then you must add it to the `mappers` section, and you must add a child `properties` section. For the one-to-one field, you must specify the `path` and `mapper`.

For example, the following defines an `activityLegalInfo_Ext` mapper for the `Activity` schema.

```
"mappers": {
  "Activity": {
    "properties": {
      ...,
      "activityLegalInfo_Ext": {
        "path": "Activity.ActivityLegalInfo_Ext",
        "mapper": "#/mappers/ActivityLegalInfo_Ext"
      }
    }
  },
```

The child mapper

In the mapping file, you must add the child mapper to the `mappers` section with the following properties:

- `schemaDefinition` - set to the name of the child schema
- `root` - set to `entity.`*`<nameOfChildEntity>`*

Then, you must add a `properties` section to that mapper with the following properties:

- A read-only `id` property whose path is typically set to "*`<childEntity>`*`.RestId`".
- One property for each inline child field. For each property, you must specify its `path`.

For example, the following defines the mapper for `ActivityLegalInfo_Ext`, with three properties: `id`, `legalCaseNumber`, and `legalReviewDate`.

```
"mappers": {
  ...
  "ActivityLegalInfo_Ext": {
    "schemaDefinition": "ActivityLegalInfo_Ext",
    "root": "entity.ActivityLegalInfo_Ext",
    "properties": {
      "id": {
        "path": "ActivityLegalInfo_Ext.RestId"
      },
      "legalCaseNumber": {
        "path": "ActivityLegalInfo_Ext.LegalCaseNumber"
      },
      "legalReviewDate": {
        "path": "ActivityLegalInfo_Ext.LegalReviewDate"
      }
    }
  }
}
```

The inline child properties are typically scalars, compound datatypes, or foreign keys. For more information on how to configure mappers for these type of properties, see:

- “Adding scalars” on page 33
- “Adding compound datatypes” on page 39
- “Adding foreign keys” on page 43

The updater extension file

Each updater extension file for every API contains a `updaters` section. In the base configuration, this section is typically blank. You can add the name of one or more schemas to the `updaters` section. In each schema, you can define a `properties` section, which lists one or more properties defined in the schema.

For a one-to-one relationship, you must add or modify two updaters:

- The parent updater must specify the path mapping for the one-to-one property that points to the child.
- You must also add a new updater for the child that specifies the path mappings for the inline child properties.

The parent updater

In the updater file, if the parent is not already present, then you must add it to the `updaters` section, and you must add a child `properties` section. For the one-to-one field, you must specify the following attributes:

- `path` - set to `<root>.<foreignKeyToChild>`
- `create` - set to `new <childEntity>(<root>)`
- `updateRef` - set to `#/updaters/<childUpdater>`

For example, the following defines an `activityLegalInfo_Ext` updater for the `Activity` schema.

```
"updaters": {
  "Activity": {
    "properties": {
      ...
      "activityLegalInfo_Ext": {
        "path": "Activity.ActivityLegalInfo_Ext",
        "create": "new ActivityLegalInfo_Ext(Activity)",
        "updaterRef": "#/updaters/ActivityLegalInfo_Ext"
      }
    }
  },
  ...
```

The child updater

In the updater file, you must add the child updater to the `updaters` section with the following properties:

- `schemaDefinition` - set to the name of the child schema
- `root` - set to `entity.<nameOfChildEntity>`

Then, you must add a `properties` section to that updater with one property for each inline field that is to be writeable from Cloud API. For each property, you must specify its `path`.

For example, the following defines the updater for the `ActivityLegalInfo_Ext` schema, with two properties: `legalCaseNumber` and `legalReviewDate`.

```
"updaters": {
  ...,
  "ActivityLegalInfo_Ext": {
    "schemaDefinition": "ActivityLegalInfo_Ext",
    "root": "entity.ActivityLegalInfo_Ext",
    "properties": {
      "legalCaseNumber": {
        "path": "ActivityLegalInfo_Ext.LegalCaseNumber"
      },
      "legalReviewDate": {
        "path": "ActivityLegalInfo_Ext.LegalReviewDate"
      }
    }
  }
}
```

The inline child properties are typically scalars, compound datatypes, or foreign keys. For more information on how to configure updaters for these type of properties, see:

- “Adding scalars” on page 33
- “Adding compound datatypes” on page 39
- “Adding foreign keys” on page 43

Reserving IDs and checksums

When you add a one-to-one child to a parent resource, there are two additional behaviors to configure:

- Reserving ID values for one-to-one child instances
- Modifying the parent's checksum calculation to include fields in the child

Reserving IDs in composite requests

Normally, ID values for new objects are not specified until the data is committed to the database.

In the context of a composite request, multiple objects can be created in each subrequest, but none of the objects are committed until the end of the composite request. There may be situations where you need to know the ID of an object created in a subrequest, even though that object has not yet been committed to the database.

To address this issue, when a base configuration entity is used in a composite request, it can reserve an ID value. This allows an ID value to be assigned to the object, even though the data has not yet been committed.

You can configure a one-to-one child so that it also reserves ID values in composite requests.

Note that IDs are reserved in composite requests only when both the parent and child are created in the same POST. If a POST specifies only fields on the parent and the child fields are specified in a later composite subrequest PATCH, then the child ID does not get reserved and is set when the composite request completes.

Adding child fields to parent checksums

A *checksum* is a string value that identifies the "version" of a particular resource. Checksums are useful when you want to modify or delete an object and you want to verify that the object has not been modified by some other process since the last time you read it.

In most cases, an object's checksum is calculated using only the fields on the corresponding data model entity. For example, the checksum for an `Activity` is calculated using only the fields declared directly on the `Activity` entity.

When you add a one-to-one child to a parent schema, you typically want the checksum to include fields from the one-to-one child. This means that the checksum value changes if there are changes to fields on the parent entity or the one-to-one child entity. For example, if the `Activity` resource is enhanced with a one-to-one `ActivityLegalInfo_Ext` child resource, then the checksum for an `Activity` resource ought to be calculated using fields declared directly on the `Activity` entity and fields declared on the `ActivityLegalInfo_Ext` entity.

Configuring ID and checksum behaviors

Every base configuration entity has an `ExtResource` class that you can use to extend the functionality of the resource. These classes are located in the `gw.rest.ext` package.

To reserve IDs for new one-to-one child entities, add the following method override to the class. The *`<childResource>`* reference must be changed to the name of the one-to-one child resource.

```
override function finishCreate(data : DataEnvelope, batchUpdateMap : BatchUpdateMap) {
  super.finishCreate(data, batchUpdateMap)
  reserveIdsIfNecessary({this.Element.<childResource>})
}
```

To extend checksums to include child fields, add the following getter override to the class. The *`<childResource>`* reference must be changed to the name of the one-to-one child resource.

```
override property get Checksum() : String {
  return ChecksumUtil.extendChecksumWithEntities(super.Checksum, {this.Element.<childResource>})
}
```

For example, the following code is the complete `ActivityExtResource` class with extensions for the `ActivityLegalInfo_Ext` one-to-one child.

```
package gw.rest.ext.cc.common.v1.activities

uses gw.api.modules.rest.framework.v1.batch.BatchUpdateMap
uses gw.api.modules.rest.framework.v1.checksum.ChecksumUtil
uses gw.api.modules.rest.framework.v1.json.DataEnvelope
uses gw.rest.core.cc.common.v1.activities.ActivityCoreResource

@Export
```

```
class ActivityExtResource extends ActivityCoreResource {

  override function finishCreate(data : DataEnvelope, batchUpdateMap : BatchUpdateMap) {
    super.finishCreate(data, batchUpdateMap)
    reserveIdsIfNecessary({this.Element.ActivityLegalInfo_Ext})
  }

  override property get Checksum() : String {
    return ChecksumUtil.extendChecksumWithEntities(super.Checksum, {this.Element.ActivityLegalInfo_Ext})
  }

}
```

One-to-ones in responses and requests

Responses

In responses, inline child objects appear along with fields declared directly on the primary data model entity.

For example, the following shows the response to a GET `/activities` call. Note that the response includes an `activityLegalInfo_Ext` object with its own fields as well as fields declared directly on `Activity`, such as `activityPattern` and `activityType`.

```
GET /common/v1/activities/xc:20201

{
    "data": {
        "attributes": {
            "activityLegalInfo_Ext": {
                "id": "cc:S2qhgxij6HvhXz6K3t39K",
                "legalCaseNumber": "0003",
                "legalReviewDate": "2022-05-05T07:00:00.000Z"
            },
            "activityPattern": "90_day_diary",
            "activityType": {
                "code": "general",
                "name": "General"
            },
```

Keep in mind that Cloud API does not include fields in responses when the values of those fields is NULL. Thus, in order to test the behavior of one-to-one child fields, you must GET an object that already has a one-to-one child with at least one non-NULL field.

Requests

In requests, inline child objects can be included along with fields declared directly on the primary data model entity.

For example, the following shows the request body for a PATCH `/activities` call. Note that the request includes an `activityLegalInfo_Ext` object with its own fields as well as a field declared directly on `Activity` (`priority`).

```
PATCH /common/v1/activities/xc:20207

{
  "data": {
    "attributes": {
        "priority": {
            "code": "low"
            },
        "activityLegalInfo_Ext": {
            "legalCaseNumber": "0004",
            "legalReviewDate": "2022-05-07"
        }
    }
  }
}
```

Complete code sample for one-to-ones

The following code snippets define a new entity and a new one-to-one referencing that entity. In this example, the `Activity` resource has a `activityLegalInfo_Ext` inline object whose fields map to the `ActivityLegalInfo_Ext` entity.

Data model extension

File name: `Activity.etx`

```
<extension
  xmlns="http://guidewire.com/datamodel"
  entityName="Activity">
  ...
  <onetoone
    fkentity="ActivityLegalInfo_Ext"
    name="ActivityLegalInfo_Ext"
    nullok="true"/>
  ...
```

File name: `ActivityLegalInfo_Ext.eti`

```
<?xml version="1.0"?>
<entity
  xmlns="http://guidewire.com/datamodel"
  entity="ActivityLegalInfo_Ext"
  table="activitylegalinfo_ext"
  type="retireable">
  <column
    name="LegalCaseNumber"
    nullok="true"
    type="varchar">
    <columnParam
      name="size"
      value="30"/>
  </column>
  <column
    name="LegalReviewDate"
    nullok="true"
    type="datetime"/>
  <foreignkey
    fkentity="Activity"
    name="Activity"
    nullok="false"/>
  <implementsEntity
    name="Extractable"/>
</entity>
```

Schema extension file

File name: `common_ext-1.0.schema.json`

```
  "definitions": {
    "Activity": {
      "properties": {
        "activityLegalInfo_Ext": {
          "title": "ActivityLegalInfo_Ext",
          "description": "One-to-one association to ActivityLegalInfo_Ext",
          "$ref": "#/definitions/ActivityLegalInfo_Ext"
        }
      }
    },
    "ActivityLegalInfo_Ext": {
      "properties": {
        "id": {
          "title": "ID",
          "description": "Object ID",
          "type": "string",
          "readOnly": true
        },
        "legalCaseNumber": {
          "title": "LegalCaseNumber",
          "description": "Legal case number",
          "type": "string"
        },
        "legalReviewDate": {
          "title": "LegalReviewDate",
          "description": "Legal review date",
          "type": "string",
          "format": "date-time"
        }
      }
    }
  }
  ...
```

Mapping extension file

File name: `common_ext-1.0.mapping.json`

```
  "mappers": {
    "Activity": {
      "properties": {
        "activityLegalInfo_Ext": {
          "path": "Activity.ActivityLegalInfo_Ext",
          "mapper": "#/mappers/ActivityLegalInfo_Ext"
        }
      }
    },
    "ActivityLegalInfo_Ext": {
      "schemaDefinition": "ActivityLegalInfo_Ext",
      "root": "entity.ActivityLegalInfo_Ext",
      "properties": {
        "id": {
          "path": "ActivityLegalInfo_Ext.RestId"
        },
        "legalCaseNumber": {
          "path": "ActivityLegalInfo_Ext.LegalCaseNumber"
        },
        "legalReviewDate": {
          "path": "ActivityLegalInfo_Ext.LegalReviewDate"
        }
      }
    }
  }
```

Updater extension file

File name: `common_ext-1.0.updater.json`

```
  "updaters": {
    "Activity": {
      "properties": {
        "activityLegalInfo_Ext": {
          "path": "Activity.ActivityLegalInfo_Ext",
          "create": "new ActivityLegalInfo_Ext(Activity)",
          "updaterRef": "#/updaters/ActivityLegalInfo_Ext"
        }
      }
    },
    "ActivityLegalInfo_Ext": {
      "schemaDefinition": "ActivityLegalInfo_Ext",
      "root": "entity.ActivityLegalInfo_Ext",
      "properties": {
        "legalCaseNumber": {
          "path": "ActivityLegalInfo_Ext.LegalCaseNumber"
        },
        "legalReviewDate": {
          "path": "ActivityLegalInfo_Ext.LegalReviewDate"
        }
      }
    }
  }
```

Resource extension (to reserve IDs and extend checksums)

File name: `ActivityExtResource.gs`

```
package gw.rest.ext.cc.common.v1.activities

uses gw.api.modules.rest.framework.v1.batch.BatchUpdateMap
uses gw.api.modules.rest.framework.v1.checksum.ChecksumUtil
uses gw.api.modules.rest.framework.v1.json.DataEnvelope
uses gw.rest.core.cc.common.v1.activities.ActivityCoreResource

@Export
class ActivityExtResource extends ActivityCoreResource {

  override function finishCreate(data : DataEnvelope, batchUpdateMap : BatchUpdateMap) {
    super.finishCreate(data, batchUpdateMap)
    reserveIdsIfNecessary({this.Element.ActivityLegalInfo_Ext})
  }

  override property get Checksum() : String {
    return ChecksumUtil.extendChecksumWithEntities(super.Checksum, {this.Element.ActivityLegalInfo_Ext})
  }

}
```

chapter 11

Tutorials: Adding Properties

See the following tutorials for adding properties to resources.

Tutorial: Schema configuration with scalars

In this tutorial, you will add new scalar properties to the base configuration `User` schema in the Admin API, and provide mapping and updater information as appropriate. To reduce the number of steps required, this tutorial uses fields that already exist on the `User` data model entity that are not exposed to Cloud API in the base configuration. The data model entity fields used in this tutorial are:

- `CreateTime` (this datetime field will be readable but not writeable)
- `Department` (this string field will be readable and writeable)

Tutorial steps

1. In Studio, open the `admin_ext-1.0.schema.json` file.

2. The file contains the following line of code.

```
"definitions": { }
```

Replace that line with the following, which defines three new properties.

```
"definitions": {
  "User": {
    "properties": {
      "createDate_Ext" : {
        "title": "CreateDate",
        "description": "The date on which this user was created",
        "type": "string",
        "format": "date-time",
        "readOnly": true
      },
      "departmentName_Ext": {
        "title": "DepartmentName",
        "description": "The name of the department this user works in",
        "type": "string"
      }
    }
  }
}
```

There is an additional `readOnly` property specified for `createDate`. For more information, see “Making properties required by the database” on page 79.

At this point, you have defined the structure of the new properties. But there is no information on how data flows into and out of these properties.

3. In Studio, open the `admin_ext-1.0.mapping.json` file.

4. The file contains the following line of code.

```
"mappers": { }
```

Replace that line with the following, which defines how data is mapped from the database to the `createTime` and `departmentName` properties.

```
"mappers": {
  "User": {
    "properties": {
      "createDate_Ext" : {
        "path" : "User.CreateTime"
      },
      "departmentName_Ext": {
        "path": "User.Department"
      }
    }
  }
}
```

You now have new properties with information on how data flows into each property. If these properties were needed only for GETs (and not POSTs or PATCHes), you could restart PolicyCenter now and test your work. However, the `departmentName_Ext` property needs to be writeable.

5. In Studio, open the `admin_ext-1.0.updater.json` file.

6. The file contains the following line of code.

```
"updaters": { }
```

Replace that line with the following, which defines how data is mapped from the database to the `departmentName` property.

```
"updaters": {
  "User": {
    "properties": {
      "departmentName_Ext": {
        "path": "User.Department"
      }
    }
  }
}
```

The `departmentName_Ext` property can now be used for GETs, POSTs, and PATCHes. The `createDate_Ext` property has been omitted from the updater file. Therefore, it is read-only and only appears in responses.

7. Start (or restart) PolicyCenter.

Testing your work

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
   a. On the **Authorization** tab, select *Basic Auth* using user `su` and password `gw`.
2. Enter the following call, but do not click **Send** yet:
   a. POST http://localhost:8180/pc/rest/admin/v1/users
3. Specify the request payload.
   a. In the first row of tabs (the one that starts with Params), click **Body**.
   b. In the row of radio buttons, select *raw*.

c. At the end of the row of radio buttons, change the drop-down list value from *Text* to *JSON*.

d. Paste the following into the text field underneath the radio buttons.

```
{
  "data": {
    "attributes": {
      "username": "scalarTestUser",
      "departmentName_Ext": "schema config tester"
    }
  }
}
```

e. Click **Send**.

Results

The response should be "201 Created". The response body should contain information about the new user, including the create date and department. Note that:

- The structure of these fields (their names and types) comes from the schema file.
- The data in the POST response comes from the mapping file.
- The setting of the user's department was accomplished by the updater file.

Tutorial: Schema configuration with compound datatypes

In this tutorial, you will add a new compound datatype properties to the base configuration `User` schema in the Admin API, and provide mapping and updater information as appropriate. To reduce the number of steps required, this tutorial uses a field that already exist on the `User` data model entity that is not exposed to Cloud API in the base configuration. The data model entity field used in this tutorial is:

- `ExperienceLevel` (this typekey field will be readable and writeable)

Tutorial steps

1. In Studio, open the `admin_ext-1.0.schema.json` file.

2. The file contains the following line of code.

```
"definitions": { }
```

Replace that line with the following, which defines the new property. (If you have already done the tutorial for scalars, then add the `experienceLevel` property to the existing extensions.)

```
"definitions": {
  "User": {
    "properties": {
      "experienceLevel_Ext": {
        "title": "ExperienceLevel_Ext",
        "description": "The user's level of experience (high, mid, low)",
        "$ref": "#/definitions/TypeKeyReference",
        "x-gw-extensions": {
          "typelist": "UserExperienceType"
        }
      }
    }
  }
}
```

At this point, you have defined the structure of the new property. But there is no information on how data flows into and out of this property.

3. In Studio, open the `admin_ext-1.0.mapping.json` file.

4. The file contains the following line of code.

```
"mappers": { }
```

Replace that line with the following, which defines how data is mapped from the database to the `experienceLevel` property. (If you have already done the tutorial for scalars, then add the `experienceLevel` property to the existing extensions.)

```
"mappers": {
  "User": {
    "properties": {
      "experienceLevel_Ext": {
        "path": "User.ExperienceLevel",
        "mapper": "#/mappers/TypeKeyReference"
      }
    }
  }
}
```

You now have a new property with information on how data flows into it. If this property were needed only for GETs (and not POSTs or PATCHes), you could restart PolicyCenter now and test your work. However, the `experienceLevel` property needs to be writeable.

5. In Studio, open the `admin_ext-1.0.updater.json` file.

6. The file contains the following line of code.

```
"updaters": { }
```

Replace that line with the following, which defines how data is mapped from the database to the `experienceLevel` property. (If you have already done the tutorial for scalars, then add the `experienceLevel` property to the existing extensions.)

```
"updaters": {
  "User": {
    "properties": {
      "experienceLevel_Ext": {
        "path": "User.ExperienceLevel",
        "valueResolver": {
          "typeName": "TypeKeyValueResolver"
        }
      }
    }
  }
}
```

This property can now be used for GETs, POSTs, and PATCHes.

7. Start (or restart) PolicyCenter.

Testing your work

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
   a. On the **Authorization** tab, select *Basic Auth* using user `su` and password `gw`.
2. Enter the following call, but do not click **Send** yet:
   a. POST http://localhost:8180/pc/rest/admin/v1/users
3. Specify the request payload.
   a. In the first row of tabs (the one that starts with Params), click **Body**.
   b. In the row of radio buttons, select *raw*.
   c. At the end of the row of radio buttons, change the drop-down list value from *Text* to *JSON*.
   d. Paste the following into the text field underneath the radio buttons.

```
{
  "data": {
    "attributes": {
      "username": "compoundTestUser",
      "experienceLevel_Ext" : {
          "code" : "mid"
      }
    }
```

```
  }
}
```

e. Click **Send**.

Results

The response should be "201 Created". The response body should contain information about the new user, including the experience level. Note that:

- The structure of the field (its name and type) comes from the schema file.
- The experience level in the POST response comes from the mapping file.
- The setting of the user's experience level was accomplished by the updater file.

part 3

Modifying endpoint behaviors

You can configure schema properties with additional behaviors, such as:

- Setting the property to read-only
- Indicating the property as required by the database
- Making the property sortable or filterable by the corresponding collection
- Adding additional metadata, such as the first version of the API to include the property

These additional behaviors are typically defined using one or more property attributes.

Where are property attributes configurable?

Most attributes are configurable from the schema file. Within the schema file, some property attributes are declared directly on the property itself. This includes:

- Standard JSON properties, such as `readOnly`
- Guidewire extension properties declared directly off the attribute, such as `x-gw-sinceExtensionVersion`

Other property attributes are Guidewire extensions declared in an object named `x-gw-extension`. This includes:

- `filterable` and `sortable`
- `requiredForCreate`

For example, if you had an `expirationDate` property that was filterable, sortable, and required for creation, the syntax for the declaration of these properties would be as follows:

```
"properties": {
  "expirationDate": {
    ...
    "x-gw-extensions": {
      "requiredForCreate": true,
      "filterable": true,
      "sortable": true
    }
  }
```

Still other properties can only be modified through the use of resource files, such as some custom sorts and filters.

Summary of schema property attributes

The following attributes are direct children of the `property` declaration:

Attribute	Description	Example
`readOnly`	Boolean identifying if the property as read-only. (The default is false.)	`"readOnly": true`
`x-gw-nullable`	Boolean identifying if the property can be explicitly set to null. (The default is true.) Used to set a property as required by the database. For more information, see "Making properties required by the database" on page 79.	`"x-gw-nullable": false`
`x-gw-sinceExtensionsVersion`	String identifying the first version of the API to include the extension property	`"x-gw-sinceExtensionsVersion": "1.1.0"`

The following attributes are declared in the `x-gw-extension` object:

Attribute	Description	Example
`createOnly`	Boolean identifying if the property can be specified only when the object is created. (The default is false.)	`"createOnly": true`
`filterable`	Boolean identifying if the `filter` query parameter can be used on this property. In other words, collections can be filtered using this property. (The default is false.)	`"filterable": true`
`requiredForCreate`	Boolean identifying if the property must be specified when the object is created. (The default is false.) Used to set a property as required by the database. For more information, see "Making properties required by the database" on page 79.	`"requiredForCreate": true`
`sortable`	Boolean identifying if the `sort` query parameter can be used on this property. In other words, collections can be sorted using this property. (The default is false.)	`"sortable": true`

chapter 12

Collection-level behaviors

Collections have a unique set of behaviors. Commands on a collection can include sorting and filtering on specific properties, and command responses can differ based on the caller's access permissions to the resource. Each of these behaviors can be impacted by the collection type.

There are three collection types: query-backed, stream-backed, and dual-backed. The type of collection determines how you extend the collection resource through schema files and resource files, and can also determine which resources and properties are returned when using additional accessible fields filters.

Collection types

Collections can be query-backed, stream-backed, or both (dual-backed).

Query-backed collections

A query-backed collection returns results from the database one page at a time. Depending on the size of the dataset, you might not ever have all the results in memory at once. For performance reasons, this type of collection can be useful when working with large datasets.

Stream-backed collections

A stream-backed collection reads in everything at once and stores it all in memory. This allows for in-memory processing of complete datasets, which can enhance resource access capabilities. However, stream-backed collections could have negative impacts on performance when working with large datasets.

Dual-backed collections

A dual-backed collection has the functionality to perform both stream-backed and query-backed operations.

Determining the collection type

You can determine the type of the collection by looking at the class the resource collection class extends.

- Query-backed collections extend `RestQueryCollectionResource`.
- Stream-backed collections extend `RestStreamCollectionResource`.
- Resources that have both stream-backed and query-backed functionality extend `RestDualCollectionResource`.

For example, the `Users` collection class, through a series of other classes, extends `RestQueryCollectionResource`, making it a query-backed collection. You can follow the extension path like this:

UsersExtResource > UsersCoreResource > RootKeyableBeanRestCollectionResource > KeyableBeanRestQueryCollectionResource > **RestQueryCollectionResource**

Determining which collection type to use in a dual-backed collection

Because dual-backed collections have both query- and stream-backed functionality, a determination needs to be made as to which one to use. That determination is made based on a combination of a configuration default and custom overrides.

The PolicyCenter base configuration defaults to using either query-backed or stream-backed for dual-backed collections. The default can differ based on the PolicyCenter version, whether the version was installed as an upgrade or as a new install, or configuration preferences.

You can override the default for a specific dual-backed collection resource type or for an instance of a resource type.

- To set the type to be used for a particular dual-backed collection resource, modify the collection's resource settings in the shared_ext-1.0.apiconfig.yaml file.
- The type can also be set based on a particular instance of the collection resource. This is done through custom logic that overrides the `getCollectionBacking` method of the `RestDualCollectionResource`.

IMPORTANT:

If your configuration defaults to stream-backed data retrieval for dual-backed collections, Guidewire highly recommends overriding collections with large datasets to use query-backed.

Extending sorting and filtering

It's important to know what type of collection you're working with when you want to extend the collection resource sorting and filtering behaviors.

Specifying a property as sortable gives callers the ability to sort the collection based on that property. You can extend the schema or resource file to make properties on a collection sortable. For more information, see "Making properties sortable" on page 83.

Specifying a property as filterable gives callers the ability to filter the collection based on that property. You can extend the schema or resource file to make properties on a collection filterable. For more information, see "Making properties filterable" on page 87.

Resource access

When additional accessible fields filters are in place, a sortable or filterable field may not always be available to the caller. See "Sorting and filtering on accessible fields" in "Resource access files: additional accessible fields filters" for more information.

REST endpoint generator collection types

The REST endpoint generator supports the creation of only query-backed or stream-backed collections. It does not support dual-backed collections.

chapter 13

Making properties read only

A *read-only property* is a property whose value cannot be set or modified through Cloud API. This can occur for the following reasons:

- The property is read-only in the base application.
- The property is settable in the base application, but there is a business requirement that it cannot be modified through Cloud API.

To set a property as read-only:

1. In the schema file, set the `readOnly` property to true
2. In the mapping file, specify a mapper as normal.
3. Omit the property from the updater file.

For example, suppose the `CustomEntity_Ext` entity has an `ExpirationDate` field. The corresponding resource property is read-only. The property declaration would be:

```
"definitions": {
  "CustomEntityExt": {
    ...
    "properties": {
      ...
      },
      "expirationDate": {
        "type": "string",
        "format": "date-time",
        "readOnly": true
      }
      ...
```

The mapping declaration would be:

```
"mappers": {
  "CustomEntityExt": {
    "expirationDate": {
      "path": "CustomEntity_Ext.ExprationDate"
    }
```

There would be no updater declaration.

chapter 14

Making properties required by the database

In the data model, some entity fields are required. You cannot create an instance of the entity without specifying a value for the field, and the value can never be null. For example, suppose that an insurer has a business rule stating every activity must have an end date (the date by which it is expected to be completed). To enforce this, the `Activity` entity's `EndDate` field is required. These fields are sometimes referred to as *required by the database*.

In a schema, you can configure a resource property to reflect that it is required by the database. To do this, you must set the following properties:

- `"requiredForCreate": true`
  - This `x-gw-extensions` attribute indicates the property must be included in POST payloads.
- `"x-gw-nullable": false`
  - This attribute indicates that when the property is specified, its value cannot be set to null

The `"requiredForCreate": true` attribute, by itself, only mandates that the property must be specified in a POST. There is nothing to prevent a caller from including the property but setting the property's value to null.

The `"x-gw-nullable": false` attribute, by itself, only mandates that if the property is specified, the property's value cannot be set to null. There is nothing to prevent a caller from omitting the property.

By combining the two expressions, you are stating that the property must be specified in a POST and set to a non-null value, and anytime thereafter that the property is specified, it must be set to a non-null value. This is the equivalence of setting a data model entity field to required.

For example, suppose the `CustomEntity_Ext` entity has an `CustomDescription` field, which is a string. The field is required. The property declaration would be:

```
"definitions": {
  "CustomEntityExt": {
    ...
    "properties": {
      ...
      "customDescription": {
        "type": "string",
        "x-gw-nullable": false,
        "x-gw-extensions": {
          "requiredForCreate": true
        }
      }
      ...
```

Note: There is also a concept of requiredness at the schema level. You can specify a property is required at the schema level by specifying `"required": true`. When a schema property is set to required, it must be provided every time a payload is submitted. In most cases, this is not something you would want to specify in Cloud API schemas, as it forces you to specify a value for PATCHes, even if the property in question is not being changed.

Overriding base configuration properties

If a base configuration property is not required by the database, you can make it required by the database by adding it to the appropriate schema extension file and setting the appropriate properties as shown in the previous example.

If a base configuration property is required by the database, you cannot make it non-required.

chapter 15

Making properties writeable at creation only

A "*writeable only at creation" property* is a property whose value can only be specified in a POST and not in any PATCH. This is used for fields that, once set, cannot be modified.

To specify a property is writeable only at creation, set the `x-gw-extensions` object's `create-only` property to true:

```
"x-gw-extensions": {
  "create-only": true
}
```

For example, suppose the `CustomEntity_Ext` entity has an `ExpirationDate` field. This field can be set on a POST, but not in any PATCH. The property declaration would be:

```
"definitions": {
  "CustomEntityExt": {
    ...
    "properties": {
      ...
    "expirationDate": {
      "x-gw-extensions": {
        "create-only": true
      }
      }
    }
  }
}
```

chapter 16

Making properties sortable

A *sortable property* is a property that can be used to sort the elements of a collection resource. You can use the instructions in this topic to make properties sortable that are:

- Part of custom entities
- Custom properties that extend base configuration entities
- Base configuration properties that are not already sortable

Note that if a base configuration property is sortable, you can't make it unsortable.

IMPORTANT: Adding new sortable columns can have performance implications, particularly if the collection to be sorted tends to have a large number of elements. In these cases, you might need to add new database indexes to maintain acceptable performance.

Query-backed collections vs stream-backed collections

The way in which you make a property sortable depends not only on the type of property, but also on the type of the collection. Collections can be query-backed or stream-backed.

- Query-backed collections extend `RestQueryCollectionResource`.
- Stream-backed collections extend `RestStreamCollectionResource`.
- Resources that have both stream-backed and query-backed collections extend `RestDualCollectionResource`.

For all collection types, use the schema file to make a property sortable when the property exists directly on the data model entity.

For query-backed collections, use resource files to sort on a property that is derived at run time or derived from a join of multiple data model entities.

For stream-backed collections, always use the schema files. If you want to sort based on a property that's accessed through a join of data model entities, create an enhancement property with the join, then add that property to the schema file (as well as the mapping file).

Note:

See *Gosu Reference* for more information on enhancements. See "Syntax for schema configuration files" on page 21 for information on adding properties to mapping files.

For dual collections, perform the required actions for both query-backed and stream-backed. For example, if you create a sort based on a property that requires a join between multiple data model entities, you need to update the resource file (for query-backed) *and* create an enhancement and update the schema file (for stream-backed).

Adding sortable properties through schema files

Properties that are directly on a data model entity (or created through an enhancement for stream-backed collections) can easily be made sortable by updating the schema file associated with the API to which the entity belongs. For example, to make a property on the User data model entity sortable, you update the `admin_ext-1.0.schema.json` file. Within this file, set the `x-gw-extensions` object's sortable property to `true`:

```
"x-gw-extensions": {
  "sortable": true
}
```

For example, suppose the `CustomEntity_Ext` entity has an `ExpirationDate` field. When retrieving a collection of `CustomEntity_Ext` instances, the caller application can opt to sort the collection based on the expiration date using a call such as:

```
GET /common/v1/customentity-exts?sort=expirationDate
```

In the common-1.0.schema.json file, the property declaration would be:

```
"definitions": {
  "CustomEntityExt": {
    ...
      "properties": {
     ...
       "expirationDate": {
         "x-gw-extensions": {
           "sortable": true
         }
       }
```

Adding sortable properties through resource files

To make a property sortable that is not directly mapped to an entity, you need to update the resource file associated with the collection you want to sort. For example, if the property is returned as part of the Users collection, you'll need to update the UsersExtResource.gs file.

The following is the simplest example of making a property in a query-backed collection sortable:

```
override property get CustomSortColumnMap() : Map<String, QuerySortColumn> {
    var customSort = new HashMap<String, QuerySortColumn>(super.CustomSortColumnMap)
    customSort.put("sortProperty_Ext", new SimpleSortColumn(Resource#Property))
    return customSort
}
```

This example overrides `getCustomSortColumnMap`. Notice it returns a type of `QuerySortColumn`, as this is a query-backed collection.

The first line is a call to the `super` method. You need to start your custom sorts with this call to avoid losing any sorts already defined in the core resource class.

Next is a call to `SimpleSortColumn`. Use `SimpleSortColumn` for sorting simple properties that cannot be sorted by adding them to the schema.

In this example, make the following replacements:

- *`sortProperty_Ext`*: Replace with the name of the property you're sorting on, such as `customProperty_Ext`.
- *`Resource#Property`*: Replace with the name of the property and the resource where it can be found. For example, `CustomEntity#CustomProperty`.

Here's an example of adding a custom sort on the ActivityPattern property of an Activity:

```
override property get CustomSortColumnMap() : Map<String, QuerySortColumn> {
   var customSort = super.CustomSortColumnMap
   customSort.put("activityPattern_Ext", new SimpleSortColumn(Activity#ActivityPattern))
   return customSort
 }
```

After adding this code to the JobActivitiesExtResource class (in the JobActivitiesExtResource.gs file), you can run the following:

```
GET /job/v1/jobs/pc:101/activities?sort=activityPattern
```

Some properties are derived from other resources. The sort for these properties is a little more complex. Instead of calling a simple sort on a single property, you need to join tables to retrieve the property you want to sort on. Here's what that might look like:

```
private static var sortSortColumnPath = Lists.newArrayList({Resource1#JoinProperty.getPropertyInfo() as
IEntityPropertyInfo, Resource2#SortProperty.PropertyInfo as IEntityPropertyInfo})

 override property get CustomSortColumnMap() : Map<String, QuerySortColumn> {
   var customSort = new HashMap<String, QuerySortColumn>(super.CustomSortColumnMap)
   customSort.put("sortProperty_Ext", new PathSortColumn(abuidSortColumnPath))
   return customSort
 }

}
```

You need to replace the placeholders in the preceding example with the following:

- *Resource1*: The data model entity of the collection you're sorting on.
- *JoinProperty*: The foreign key property used to join this data model entity to another entity.
- *Resource2*: The data model entity that contains the property on which you want to sort.
- *SortProperty*: The property on which you want to sort.
- *sortProperty_Ext*: The name you'll use in your sort command on your endpoint to sort on the property.

Here's a real-world example:

```
class UsersExtResource extends UsersCoreResource {

 private static var enSortColumnPath = Lists.newArrayList({User#Contact.getPropertyInfo() as IEntityPropertyInfo,
UserContact#EmployeeNumber.PropertyInfo as IEntityPropertyInfo})


 override property get CustomSortColumnMap() : Map<String, QuerySortColumn> {
   var customSort = new HashMap<String, QuerySortColumn>(super.CustomSortColumnMap)
customSort.put("employeeNumber_Ext", new PathSortColumn(enSortColumnPath))

   return customSort
 }

}
```

In this example, you want to sort users based on their employee number, like this:

GET `/admin/v1/users?sort=employeeNumber_Ext`

However, you can't use a simple sort because the EmployeeNumber property isn't part of the User data model entity; User retrieves that property from UserContact. So before you can sort, you need to join these two resources together. You do that by creating an ArrayList that defines how the entities will be joined:

```
private static var enSortColumnPath = Lists.newArrayList({User#Contact.getPropertyInfo() as IEntityPropertyInfo,
UserContact#EmployeeNumber.PropertyInfo as IEntityPropertyInfo})
```

In this example the Contact property from the User data model entity will be used to join to the UserContact entity, where the EmployeeNumber property is located.

You then pass this array to `PathSortColumn` to perform the sort.

```
new PathSortColumn(enSortColumnPath)
```

Notice that in the simple example we showed previously you used `SimpleSortColumn`, passing in the resource and property, but here you need to use `PathSortColumn` so you can pass in an array of information about the resources required to retrieve the information.

Set a default sort order

You can specify that a collection has a default sort order. This sort order is used when the caller does not specify a sort of their own. This is done in the API's apiconfig extension file.

Every API has an apiconfig extension file named `<api_collection>_ext-1.0.apiconfig.yaml`. This file maps both element resource and collection resources to Gosu impl files. It can also define default sort orders for collections.

For example, the following code specifies that, by default, `CustomEntitiesExt` collections are sorted by `expirationDate` (ascending). If any elements have the same value, they are sorted by `customDescription` (ascending).

```
CustomEntitiesExt:
  defaultSort:
    - expirationDate
    - customDescription
```

To specify a descending order, use the following syntax:

```
- "-<propertyName>"
```

For example, the following code specifies that, by default, CustomEntitiesExt collections are sorted by expirationDate (descending). If any elements have the same value, they are sorted by customDescription (ascending).

```
CustomEntitiesExt:
  defaultSort:
    - "-expirationDate"
    - customDescription
```

chapter 17

Making properties filterable

A *filterable property* is a property that can be used to narrow down the results returned from a request to retrieve a resource collection. Many properties are filterable in the base configuration, but there might come a time when you need to add a custom filter, either to extend the base configuration or when adding a custom entity. There are two ways to add a custom filter, depending on the type of property.

- Schema files. For properties directly on a data model entity, you can make the property filterable by updating the appropriate schema file.
- Resource files. Properties with values that are either generated at run-time or derived from multiple data model entities can be made filterable by updating the resource file of the collection entity on which you're filtering. (Note that this applies only when you are filtering a query-backed collection.)

Keep in mind that if a base configuration property is filterable, you cannot make it unfilterable.

Performance considerations

Before getting into the details of how to add filters, it's a good idea to think about the size of the collection you're filtering on. Filtering on very large collections can come with performance costs. Here are some suggestions for minimizing or avoiding this issue.

Add database indexes. Filtering on a property that's indexed in the database is much more efficient than filtering on a property that is not indexed. If you're adding a filter to a collection that is or could potentially be very large, the best option for avoiding system problems is to index the property. See *Configuration* for more information on adding indexes.

However, in some cases indexing alone doesn't solve the problem. You must take additional steps to be certain that your filter works in such as way that the index will be leveraged by the database.

Limit the operators that can be used with the filter. When you update the schema to specify a property as filterable, the filter is automatically assigned all possible filter operators, such as starts with (`sw`) and greater than (`gt`). However, these operators will likely not be able to leverage the index at the database level. Filtering on large collections aught to be limited to allow only the equals (`eq`) operator. To enforce this limitation, you must create the filter using resource files rather than the schema file. This will enable you to specify which operators are allowed on the property when you filter.

Make the filter case sensitive. By default, filtering is not case sensitive. However, case-insensitive searches will likely not be able to leverage the index at the database level. This means that even if you've indexed a property and limited filter operators to only `eq`, your filter can still cause serious performance issues. You must take the additional step of making the filter case-sensitive to ensure the index is used, thus avoiding most performance issues. As with limiting operators, making filters case-sensitive requires you to use resource files rather than the schema to define your filter.

Query-backed collections vs stream-backed collections

The way in which you make a property filterable depends not only on the type of property, but also on the type of the collection. Collections can be query-backed or stream-backed.

- Query-backed collections extend `RestQueryCollectionResource`.
- Stream-backed collections extend `RestStreamCollectionResource`.
- Resources that have both stream-backed and query-backed collections extend `RestDualCollectionResource`.

For all collection types, use the schema file to make a property filterable when the property exists directly on the data model entity.

For query-backed collections, use resource files to filter on a property that is derived at run time, derived from a join of multiple data model entities, or when you want to limit the filter operators.

For stream-backed collections, always use the schema files. If you want to filter based on a property that's accessed through a join of data model entities, create an enhancement property with the join, then add that property to the schema file (as well as the mapping file).

> **Note:** See *Gosu Reference* for more information on enhancements. See "Syntax for schema configuration files" on page 21 for information on adding properties to mapping files.

For dual collections, perform the required actions for both query-backed and stream-backed. For example, if you create a filter based on a property that requires a join between multiple data model entities, you need to update the resource file (for query-backed) and create an enhancement and update the schema file (for stream-backed).

Adding filters through the schema file

Properties that are directly on a data model entity (or created through an enhancement for stream-backed collections) can easily be made filterable by updating the schema file associated with the API to which the entity belongs. For example, to make a property on the User data model entity filterable, you update the `admin_ext-1.0.schema.json` file.

Suppose you've created a `CustomEntity_Ext` entity with an `ExpirationDate` field, and you want to filter on that field. Within the schema file, you need to set the entity's filterable property within the `x-gw-extensions` object to true:

```
"definitions": {
  "CustomEntityExt": {
    ...
      "properties": {
        ...
        "expirationDate": {
          "x-gw-extensions": {
            "filterable": true
          }
        }
```

Now when retrieving a collection of `CustomEntity_Ext` instances, the caller application can opt to filter the collection based on the expiration date using a call such as this:

```
GET /common/v1/customentity-exts?filter=expirationDate:gt:2020-05-11T07::00::00.000Z
```

Adding filters through resource files

Some properties cannot be made filterable by simply updating the schema file, either because their values are calculated at run-time and not stored in the database, or their values are the result of a join on multiple datasets. In these cases, updating the schema file won't make the property filterable. Instead, you need to create a new filter class and instantiate it from the resource file associated with the collection on which you want to filter. (This applies only to query-backed collections.)

As an example, suppose you want to filter users based on their employee number. You would need to use the /admin/v1/users endpoint to retrieve the collection of users, then apply a filter to that collection. However, although the Users collection returns an employeeNumber (in the default configuration), that number does not exist on the User data model entity itself. Instead, EmployeeNumber is being pulled from the UserContact data model entity, which User

references through a foreign key to the UserContact#ID. So in order to create a filter, you need to do the same thing: query for EmployeeNumber in UserContact by matching the User#Contact field to the UserContact#ID field.

Here's an example of how to do that.

Create a new class

```
package gw.rest.ext.cc.admin.v1.user

uses gw.api.database.IQueryBuilder
uses gw.api.database.InOperation
uses gw.api.database.Queries
uses gw.api.database.Relop
uses gw.api.filters.IQueryFilter
uses gw.api.filters.StandardQueryFilter
uses gw.api.modules.rest.framework.v1.query.filters.QueryFilterOp
uses gw.api.modules.rest.framework.v1.query.filters.RestQueryFilter

class EmployeeNumberFilter_Ext extends RestQueryFilter {

   override function createQueryFilter(propertyName : String, queryFilterOp : QueryFilterOp, parseValue : Object) :
IQueryFilter {

 return new StandardQueryFilter("employeeNumber_Ext", \q -> {
   var empQuery = Queries.createQuery(entity.UserContact).compare(UserContact#EmployeeNumber, Relop.Equals, parseValue)
   q.subselect(User#Contact, InOperation.CompareIn, empQuery, UserContact#ID)
 })
}

   override property get AllowedOperators() : Set<QueryFilterOp> {

     return {
         QueryFilterOp.eq

     }
   }

   override function parseValue(op : QueryFilterOp, employeeNumber : String) : Object {
     if (employeeNumber == "null" && op.getAllowNulls()) {
       return null;
     }

     return employeeNumber
   }

}
```

Walking through the example a step at a time, here's how it works.

The first thing you need to do is create a new class that extends `RestQueryFilter`.

```
class EmployeeNumberFilter_Ext extends RestQueryFilter {
```

When you create the class, you need to decide in which package to store it. In this example, it's stored in the same place as the resource file that's being extended:

```
package gw.rest.ext.cc.admin.v1.user
```

Next you need to add references to the objects you're going to use:

```
uses gw.api.database.IQueryBuilder
uses gw.api.database.InOperation
uses gw.api.database.Queries
uses gw.api.database.Relop
uses gw.api.filters.IQueryFilter
uses gw.api.filters.StandardQueryFilter
uses gw.api.modules.rest.framework.v1.query.filters.QueryFilterOp
uses gw.api.modules.rest.framework.v1.query.filters.RestQueryFilter
```

Within this new class, you override the `createQueryFilter` function, which is where you'll define the query that retrieves the employee number.

```
override function createQueryFilter(propertyName : String, queryFilterOp : QueryFilterOp, parseValue : Object) :
IQueryFilter {
```

In this example the query is created as a `StandardQueryFilter`. In creating this filter you first give it a name (in this case **exmployeeNumber_Ext**) and then a query. This query retrieves the UserContact information for each User,

matching the Contact property foreign key to the ID, then compares the UserContact EmployeeNumber to the filter value (parseValue).

```
return new StandardQueryFilter("employeeNumber_Ext", \q -> {
   var empQuery = Queries.createQuery(entity.UserContact).compare(UserContact#EmployeeNumber, Relop.Equals, parseValue)
   q.subselect(User#Contact, InOperation.CompareIn, empQuery, UserContact#ID)
 })
```

Notice in the query that UserContact#EmployeeNumber is being compared to the filter value with `Relop.Equals`. This means that any filter applied will look only for employee numbers that exactly match the filter value. To enforce this restriction, you must also include `getAllowedOperators` and specify the equality operator:

```
getAllowedOperators and specify the equality operator:

   override property get AllowedOperators() : Set<QueryFilterOp> {

     return {
         QueryFilterOp.eq
     }
   }
```

By specifying only the `eq` operator as an allowed operator, any other operator used with this filter will produce an error.

See “Add filter operators” below for an example of how to allow multiple operators.

Next, you need to include the `parseValue` function. This function ensures that the filter value entered into the query string is converted to the proper datatype. (For example, a value of 2023-06-13T20:07:44.243Z would need to be parsed from a string to a Date object in order to filter on a Date field.) In this example, there is also a check for null to ensure the string "null" is parsed to an actual null (or empty) value rather than interpreted as the literal string "null".

```
override function parseValue(op : QueryFilterOp, employeeNumber : String) : Object {
  if (employeeNumber == "null" && op.getAllowNulls()) {
     return null;
  }

  return employeeNumber
}
```

Update the collection class

The final step is to update the resource file associated with the collection. For this example you’ll update the UsersExtResource.gs file.

In the resource file, override `getCustomFilters` to enable the new filter:

```
override function getCustomFilters(config: ApiConfiguration) : Map<String, RestQueryFilter> {
    var customFilters = new HashMap<String, RestQueryFilter>(super.getCustomFilters(config))
    customFilters.put("employeeNumber_Ext", new EmployeeNumberFilter_Ext())
    return customFilters;
  }
```

Be sure to call `super` first, to ensure you don’t overwrite any existing filters that could already be in place. Then add your new filter, giving it a name (**employeeNumber_Ext**) and an instance of the new class you just made.

Grant access to the filter

API role files determine which endpoints and fields a caller can access. By default, a caller has no access to any endpoints nor to any fields on resources returned by any endpoint. You must specifically grant access to each endpoint and then to one or more fields in the resources returned by an endpoint.

In an API role file, the `accessibleFields` section identifies which fields are accessible. There are different ways that you can grant access to fields:

- Listing each accessible field explicitly
- Using the "*" wildcard, which grants access to every field in the resource

When you add a custom filter through resource files, a caller will be able to access it if they have a role that grants view access to all fields using the "*" wildcard, including customer filters. If they do not have view access to all fields, then the filter must be added explicitly. If neither of these things are done, the filter will be inaccessible to the caller.

For example, to grant access to the employeeNumber filter, the relevant API role file needs to have either...

```
Users:
  view:
    - "*"
```

...or...

```
Users:
  view:
    - "employeeNumber"
    ...
```

For more information on how to grant access to fields in API role files, see “API role accessible fields” on page 375.

Using the filter

You can now filter on a specific employee number within the Users collection, like this:

GET `/admin/v1/users?filter=employeeNumber_Ext:eq:1001`

Add filter operators

The preceding example created a filter that allows you to filter on an exact match of an employee number. But in some cases, you might not want an exact match, you might want to find all values that are greater than or less than a certain value, or that contain or start with a string of characters. The following example demonstrates how to allow multiple operators with your filter.

> **Note:** If the collection you’re applying the filter to is or could potentially become very large, adding certain operators could cause performance issues. For large collections, consider limiting your available operators.

This example modifies the previous example by adding the “starts with” (`sw`) operator to the allowed operators when filtering on an employee number.

```
class EmployeeNumberFilter_Ext extends RestQueryFilter {

   override function createQueryFilter(propertyName : String, queryFilterOp : QueryFilterOp, parseValue : Object) :
IQueryFilter {

     var eFilter : StandardQueryFilter

     switch (queryFilterOp) {
       case QueryFilterOp.eq:
         eFilter = new StandardQueryFilter("employeeNumber_Ext", \q -> {
           var empQuery = Queries.createQuery(entity.UserContact).compare(UserContact#EmployeeNumber, Relop.Equals,
parseValue)
           q.subselect(User#Contact, InOperation.CompareIn, empQuery, UserContact#ID)
         })
         break
       case QueryFilterOp.sw:
         eFilter = new StandardQueryFilter("employeeNumber_Ext", \q -> {
           var empQuery = Queries.createQuery(entity.UserContact).startsWith(UserContact#EmployeeNumber,
String.valueOf(parseValue), true)
           q.subselect(User#Contact, InOperation.CompareIn, empQuery, UserContact#ID)
         })
         break

     }
     return eFilter
   }

   override property get AllowedOperators() : Set<QueryFilterOp> {
     return {
         QueryFilterOp.sw,
         QueryFilterOp.eq
     }
   }

   override function parseValue(op : QueryFilterOp, employeeNumber : String) : Object {
     if (employeeNumber == "null" && op.getAllowNulls()) {
       return null;
     }

     return employeeNumber
   }

}
```

When you allow multiple operators on a filter, you need to create a separate query for each filter. This example uses a `switch` statement to return the query that matches the operator that is being used.

```
switch (queryFilterOp) {
```

Note: For a list of valid operators, see *Cloud API Consumer Guide*

The first case applies to the `eq` operator, and is identical to the employee number query shown previously.

```
case QueryFilterOp.eq:
  eFilter = new StandardQueryFilter("employeeNumber_Ext", \q -> {
    var empQuery = Queries.createQuery(entity.UserContact).compare(UserContact#EmployeeNumber, Relop.Equals, parseValue)
    q.subselect(User#Contact, InOperation.CompareIn, empQuery, UserContact#ID)
  })
  break
```

The second case applies to the `sw` operator. This query is similar to the equality case, but instead of performing a query using `Relop.Equals`, you need to use the `startsWith` method on the query.

```
case QueryFilterOp.sw:
  eFilter = new StandardQueryFilter("policyNumber_Ext", \q -> {
    var empQuery = Queries.createQuery(entity.UserContact).startsWith(UserContact#EmployeeNumber,
String.valueOf(parseValue), true)
    q.subselect(User#Contact, InOperation.CompareIn, empQuery, UserContact#ID)
  })
  break
```

Note that the last parameter in the call to `startsWith` is set to `true`. This parameter specifies whether the filter is case insensitive. Set this value to `false` to make the filter case sensitive.

In addition to adding a new query in the `createQueryFilter` function, you also need to add the additional operator to the list of `AllowedOperators`:

```
override property get AllowedOperators() : Set<QueryFilterOp> {
  return {
     QueryFilterOp.eq,
     QueryFilterOp.sw
  }
}
```

The rest of the code is created in the same way as in the earlier example.

Now you can use endpoints like either of the following to retrieve users based on their employee number:

GET `/admin/v1/users?fields=lastName,employeeNumber&filter=employeeNumber_Ext:eq:US-10011`

GET `/admin/v1/users?fields=lastName,employeeNumber&filter=employeeNumber_Ext:sw:US`

chapter 18

Excluding properties from responses

By default, properties are returned in endpoint responses for both collections (summary response) and single elements (detail response). You can change this behavior with the `defaultViews` property.

For more information on default and summary responses, see the *Cloud API Consumer Guide*.

IMPORTANT: The `defaultViews` property is for use only with custom properties. Guidewire does not recommend using this property to modify the behavior of base configuration properties.

Return a field only in detail responses

If you want a field to be returned by default in endpoint responses only in detail view but not in the collection summary, set the `defaultViews` property to `detail`:

```
"x-gw-extensions": {
 "defaultViews": [
   "detail"
 ]
```

Do not return a field in a response

There might be some fields that you never want returned in default responses, either for summary or detail responses. For example, fields requiring calculations could cause performance issues if they were always included in responses. You can define a field as not being returned by default by setting `defaultViews` to `none`:

```
"x-gw-extensions": {
 "defaultViews": [
   "none"
 ]
```

When you do this, there are only two ways to include the property in the response:

- Specify the property explicitly with the `fields` query parameter.
- Specify `*all` in the `fields` query parameter. (Not recommended for production environments.)

chapter 19

Adding additional metadata for properties

The x-gw-sinceExtensionsVersion attribute

Every property can have an `x-gw-sinceExtensionsVersion` attribute. This is a string value that specifies the first version of the API that included the extension property.

For example, the following specifies that the `CustomEntity_Ext` entity has an `ExpirationDate` field that was added in version 1.1.0.

```
"definitions": {
  "CustomEntityExt": {
    ...
    "properties": {
      ...
      "expirationDate": {
        "x-gw-sinceExtensionsVersion": "1.1.0"
      }
      ...
```

There is also an `x-gw-sinceVersion` used by Guidewire to identify the first version of the API that included a base configuration property.

chapter 20

Obfuscating response data

Cloud API supports the ability to obfuscate data that is included in a response. Response data can be either nullified or masked.

- When response data is *nullified*, its value is returned as null.
- When response data is *masked*, a portion of its value is returned with placeholder characters, such as a tax ID being returned as "xxx-xx-1781".

The primary type of response data that is obfuscated is *Personally Identifiable Information* (PII). Insurers must comply with any data protection and privacy regulations of the jurisdictions in which they operate. For example, companies operating in the European Union must abide by the General Data Protection Regulation (GDPR) within that jurisdiction. These regulations often specify that Personally Identifiable Information (PII) must be obfuscated.

Nullifying response data

You can nullify the response data by configuring the mapper for the relevant property. This is done in a mapping extension file. For more information on mapping extension files, see “Syntax for schema configuration files” on page 21.

For example, an account has AccountContacts. Depending on business purposes, it might have been necessary to obtain tax identification information for an AccountContact. Later, a system API caller could request the AccountContact and then view the contact's tax ID in the response. To prevent the exposure of this data, you can nullify the value in the resource mapper.

The schema for the AccountContact resource contains a `taxId` property:

```
"AccountContact": {
  "type": "object",
  "x-gw-extensions": {
    "discriminatorProperty": "contactSubtype"
  },
  "properties": {
    . . .
    "taxId": {
      "type": "string"
    },
    . . .
  }
}
```

To nullify the value of the `taxId` property, you can modify that property in the AccountContact mapper as follows:

```
"AccountContact": {
  "schemaDefinition": "AccountContact",
  "root": "entity.AccountContact",
```

```
  "properties": {
    . . .
    "taxId": {
      "path": "null as String",
      "predicate": "false"
    },
    . . .
  }
}
```

Setting the `taxId.path` property to `"null as String"` converts the expected value to a null string. Setting `taxId.predicate` to false prevents the original value, in this case the PII, from being evaluated.

Masking response data

You can mask response data by writing a Gosu method and modifying the mapper for the relevant resource property to use that method.

- For details on implementing Gosu code, see the *Configuration Guide*.
- For more information on mapping extension files, see “Syntax for schema configuration files” on page 21.

For example, an account has AccountContacts. Depending on business purposes, it might have been necessary to obtain tax identification information for an AccountContact. Later, a system API caller could request the AccountContact and then view the contact's tax ID in the response. To limit the exposure of this data, you can mask that value.

The schema for the AccountContact resource contains a `taxId` property:

```
"AccountContact": {
  "type": "object",
  "x-gw-extensions": {
    "discriminatorProperty": "contactSubtype"
  },
  "properties": {
    . . .
    "taxId": {
      "type": "string"
    },
    . . .
  }
}
```

This property is mapped to the `TaxID` field of the `AccountContact.Contact` entity. You must create a Gosu method that masks the tax ID string. In this example, the method is named `maskTaxId`.

You then modify the `taxId` property in the AccountContact mapper as follows:

```
"Contact": {
  "schemaDefinition": "Contact",
  "root": "entity.Contact",
  "properties": {
    . . .
    "taxId": {
      "path": "AccountContact.Contact.maskTaxId(AccountContact.Contact.TaxID)"
    },
    . . .
  }
}
```

With the `taxId.path` property set to `AccountContact.Contact.maskTaxId(AccountContact.Contact.TaxID)` , the value of `TaxID` is passed through the `maskTaxId` method before being exposed to the caller.

Changing the masking pattern

To change the masking pattern applied to a resource property, you can either revise the existing masking Gosu method or write a new one.

Unmasking PII

Conversely, you can unmask PII that has been masked in the base configuration. This can be necessary when you need to expose the PII to a specific internal role, such as administrator. In such circumstances, Guidewire recommends that

you create a new schema extension for the masked property. For example, if you wish to unmask the `taxId` property, you would create a `taxIdUnmasked_Ext` schema property that is mapped directly to the `TaxID` entity field. In such a case, Guidewire recommends that you also allowlist the extended property to make it visible only to authorized roles. For details on creating resource extensions, see “Syntax for schema configuration files” on page 21. For details on allowlisting fields, see “Endpoint access” on page 373.

IMPORTANT: Nothing in the Cloud API infrastructure prevents configuration that could expose PII in a sensitive way. For example, if you specify `taxId` as a filterable parameter or sortable, it can be included as part of the URL in a request and is more likely to appear in application logs.

Unmasking the base configuration taxID field

Contact resources have a `taxID` field which stores the tax ID of the contact. In the Cloud API base configuration, this field is masked in responses so that only the last four digits appear. For example, the following is the response for a GET that retrieves a contact.

```
{
    "data": {
        "attributes": {
            "displayName": "Ray Newton",
            "taxId": "***-**-6789"
        }
    }
}
```

For some callers, such as internal or external users, the masking of tax ID may be appropriate as it protects personally identifiable information. For other callers, such as services, this masking may cause a problem as the callers may reference contacts internally using the tax ID.

There are two ways that the `taxId` field can be unmasked:

- You can configure the field so that it is always unmasked, as described in the previous topic.
- You can grant the caller the `restunmasktaxid` system permission. Any caller who has a role with this permission will get responses with unmasked tax IDs. For information on how to configure this, see “Endpoint access” on page 373.

Note that the `restunmasktaxid` system permission changes the behavior of the base configuration `taxId` field only. It has no impact on any other masked data.

part 4

Localizing schemas

Schema definition files define the behavior of REST APIs and their endpoints. They also provide API *definition documentation*. This is information in an API definition file to help users better understand API functionality.

For example, one of the schemas defined in the `common_pl-1.0.schema.json` file is the `Activity` schema. The definition includes this `description` property:

```
"description": "An `Activity` is an assignable item that represents a task to be done, a
            decision to be made, or information to be aware of"
```

API definition documentation is included in the responses when using the `/openapi.json` and `/swagger.json` endpoints in Cloud API. Users can view this information directly, through API definition tools such as Swagger UI, or in other contexts such as contextual help in a rules editor. Different users working with the same REST APIs could be working in different languages. Therefore, the Guidewire REST API Framework (and Cloud API, which rests on top of that framework) supports the ability to localize schema definition documentation. This capability is loosely referred to as "schema localization".

This topic discusses how to add locale-specific text to schema definition documentation.

chapter 21

Architecture of localized text

API definition documentation that has been localized is defined in a set of "schema.display.properties" files.

Location of schema.display.properties files

For a given InsuranceSuite application, the "schema.display.properties" files are stored in the `/modules/configuration/config/locale` directory.

- There is one default file named `schema.display.properties`.
  - This file contains schema information in English.
- There is one `schema.display_LOCALE.properties` file for each locale. Each file contains schema information for that locale. For example:
  - The `schema.display_fr.properties` file contains schema information in French.
  - The `schema.display_es.properties` file contains schema information in Spanish (for Latin American locales).
  - The `schema.display_es_ES.properties` file contains schema information in Spanish (for Spain).

In a given situation, if the locale is not English but the given display value cannot be found for the desired locale, the value in the default `schema.display.properties` is used. If for some reason the value cannot be found in the default `schema.display.properties` either, the text from the corresponding schema file property is used. For example, if localized text for an API title cannot be found in any `schema.display` file, the value of the schema file's `title` property is used.

The schema.display.properties files can be viewed in Studio by accessing the **config** > **Localizations** > **Resource Bundle 'schema.display'** node.

Note that the base configuration does provide `schema.display_LOCALE.properties` files for several locales, but these files may not be complete. Before using them in a production environment, Guidewire recommends reviewing the contents to verify completeness and correctness of the translations.

schema.display.properties keys

Every schema localization file contains a series of key/value pairs. For example:

```
json.common.v1.definitions.Activity.properties.activityType.description =
The type of this activity, such as `general` or `approval`
```

The *key* is the text that comes before the "=". It defines a piece of localizeable text in a locale-generic way. Keys for schema information are written in the following way:

- Keys for values defined in a schema.json file start with `json`.

- Keys for values defined in a swagger.yaml file start with `swagger`.
- The remainder is a dot-notation string that follows a specific pattern to identify the corresponding schema component, such as "common.v1.definitions.Activity.properties.activityType.description".

Each key must adhere to a specific pattern in order to be associated with the correct schema element. In most cases, the pattern for the key corresponds to the path to the localized property within the schema file. But there are additional nuances to the patterns. For more information, see “Associating display keys with API elements” on page 107.

> **Note:** Display keys used in an InsuranceSuite user interface have key values that are arbitrary. You can define a key any way you desire, provided that any user interface element using the key references the same value. Display keys for API definition documentation do not behave this way. The key must adhere to a specific pattern to be associated with the correct aspect of the API definition.

schema.display.properties values

The *value* is the localized text to use for that key when the user is working in the given locale. For example, suppose the key/value pair for the description of an Address is as follows:

```
json.common.v1.definitions.Address.description = An `Address` represents a postal address. The
fields available on an `Address` will depend upon the `country` value for the `Address`.
```

If a key must be shown in multiple languages, then it is defined in each locale-specific file as needed. The key remains constant across all files. Only the value varies.

For example, the following key/value pairs come from the default `schema.display.properties` file:

```
json.common.v1.definitions.Address.properties.CEDEX.description = The CEDEX bureau of the address.
Only applicable in certain countries.
```

The following key/value pairs come from the base configuration `schema.display_fr.properties` file:

```
json.common.v1.definitions.Address.properties.CEDEX.description = Bureau CEDEX de l'adresse.
Applicable uniquement dans certains pays.
```

Determining which value to use

Cloud API returns API definition documentation through:

- The `/openapi.json` and `/swagger.json` endpoints in each API.
- The schemas returned by the Cloud API-specific `/job/v1/graph-schema`, `/claim/v1/graph-schema`, `/billing/v1/graph-schema`, and `/common/v1/entity-schemas` endpoints.

When calling these endpoints, the request object can include a GW-Language header set to a given language, such as "fr_FR". For more information on this header, see the *Cloud API Consumer Guide*.

When the GW-Language header is present, Cloud API attempts to return API definition documentation in the request language. For each piece of documentation, a localized value is returned if all of the following are true:

- There is a `schema.display.properties` file for that language.
- The `schema.display.properties` file defines a value for the corresponding key.

In all other circumstances, a default value is used. The value defined in the default `schema.display.properties` file is used, if it exists. If it does not exist, the corresponding value from the schema definition file itself is used.

For example, suppose an InsuranceSuite application has an `Address` schema that includes properties for `addressLine1`, `arrondissement`, and `CEDEX`. The following information exists in each of the following files:

The schema definition file

```
"Address": {
  ...
  "properties": {
    "addressLine1": {
      "description": "The first line of the address",
      "type": "string",
```

```
    },
    "arrondissement_Ext": {
      "description": "The administrative district of the address. Used in certain large
                      French cities, in particular Paris.",
      "type": "string",
    },
    "CEDEX": {
      "description": "The CEDEX bureau of the address. Only applicable in certain countries.",
      "type": "string",
      }
```

The `schema.display.properties` file

```
json.common.v1.definitions.Address.properties.addressLine1.description = The first line of the
address, such as "123 Main Street"
json.common.v1.definitions.Address.properties.CEDEX.description = The CEDEX bureau of the
address. Only applicable in certain countries.
```

The `schema.display_fr.properties` file:

```
json.common.v1.definitions.Address.properties.CEDEX.description = Bureau CEDEX de l'adresse.
Applicable uniquement dans certains pays.
```

Now, suppose a caller requests schema information in French for the following keys:

- `json.common.v1.definitions.Address.properties.CEDEX.description`
- `json.common.v1.definitions.Address.properties.Address.description`
- `json.common.v1.definitions.Address.properties.Arrondissement_Ext.description`

Cloud API returns the following values:

- First key
  - Return value: "Bureau CEDEX de l'adresse. Applicable uniquement dans certains pays."
  - Reason: There is a schema,display.properties file for French and the file defines this key.
- Second key
  - Return value: "The first line of the address, such as "123 Main Street"
  - Reason: Even though there is a schema,display.properties file for French, the file does not define this key. However, the key is defined in the default `schema,display.properties` file.
- Third key
  - Return value: "The administrative district of the address. Used in certain large French cities, in particular Paris."
  - Reason: The key is not defined in either the locale-specific schema,display.properties file or the default schema,display.properties file. Therefore, the value of the `description` property in the schema definition is returned.

chapter 22

Associating display keys with API elements

Every API is defined in multiple files. For a given API:

- There are one or more `schema.json` files that define the schemas.
- There are one or more `swagger.yaml` files that define endpoints and their behaviors.

In order to pair localized keys with the correct API elements:

- Every `schema.json` file and `swagger.yaml` file must provide or inherit a localization key prefix.
  - The prefix can be declared added to a file.
  - The prefix can be inherited from another file when schemas are combined. For example, the schema extension files, such as `account_ext-1.0.swagger.yaml`, are combined with base configuration files. Localization prefixes declared in the base configuration files are inherited by the schemas in the extension files.
- Every display key must adhere to a specific pattern that uses the localization key prefix.

Localization key prefixes

If a `schema.json` file or `swagger.yaml` file contains API definition documentation that must be localized, the file must include an `x-gw-localizationKeyPrefix` property. This property defines a prefix used to map display keys to properties in that file.

Syntax

In the base configuration, the convention for the localization key prefix is to name it `<APIname>.<majorVersion>`. For example: `common.v1`.

To add a localization key prefix to a `schema.json` file, use:

```
"x-gw-localizationKeyPrefix": "<localizationKeyPrefix>"
```

For example, in the base configuration, the `common_pl-1.0.schema.json` files includes the following:

```
"x-gw-localizationKeyPrefix": "common.v1"
```

To add a localization key prefix to a `swagger.yaml` file, use:

```
x-gw-localizationKeyPrefix: <localizationKeyPrefix>
```

For example, in the base configuration, the `common_pl-1.0.swagger.yaml` files includes the following:

```
x-gw-localizationKeyPrefix: common.v1
```

Display key patterns for schema.json-files

`schema.json` files have two basic types of documentation text: titles and descriptions. They can be declared at four levels:

- The API itself
- API resources
- API resource properties
- The API resource `additionalProperties` property

Title and description for the schema

The following display keys map to the title and description of a schema:

- `json.<localizationPrefix>.title`
- `json.<localizationPrefix>.description`

There are no examples of this in the base configuration.

Titles and descriptions for schema definitions

The following display keys map to the title and description of each schema definition:

- `json.<localizationPrefix>.definitions.<definitionName>.title`
- `json.<localizationPrefix>.definitions.<definitionName>.description`

For example, the description property for the Common API's `Activity` schema definition (declared in `common_pl-1.0.schema.json`) maps to `json.common.v1.definitions.Activity.description`.

Titles and descriptions for schema definition properties

The following display keys map to the title and description of each property on a given schema definition:

- `json.<localizationPrefix>.definitions.<definitionName>.properties.<propertyName>.title`
- `json.<localizationPrefix>.definitions.<definitionName>.properties.<propertyName>.description`

For example, the description property for the Common API's `Activity` schema definition's `activityType` property (declared in `common_pl-1.0.schema.json`) maps to `json.common.v1.definitions.Activity.properties.activityType.description`.

Titles and descriptions for a schema definition "additionalProperties" property

The following display keys map to the title and description of the `additionalProperties` property on a given schema definition:

- `json.<localizationPrefix>.definitions.<definitionName>.additionalProperties.<propertyName>.title`
- `json.<localizationPrefix>.definitions.<definitionName>.additionalProperties.<propertyName>.description`

For example, the description property for the Composite API's `Headers` schema definition's `additionalProperties` (declared in `composite_pl-1.0.schema.json`) maps to `json.composite.v1.definitions.Headers.additionalProperties.description`.

Summary of title and description mappings

Level	Property	Display key pattern	Example
Schema	title	`json.<localizationPrefix>.title`	`json.common.v1.title`
Schema	description	`json.<localizationPrefix>.description`	`json.common.v1.description`
Schema definition	title	`json.<localizationPrefix>.definitions.<definitionName>.title`	`json.common.v1.definitions.Activity.title`
Schema definition	description	`json.<localizationPrefix>.definitions.<definitionName>.description`	`json.common.v1.definitions.Activity.description`
Schema definition property	title	`json.<localizationPrefix>.definitions.<definitionName>.properties.<propertyName>.title`	`json.common.v1.definitions.Activity.properties.activityType.title`
Schema definition property	description	`json.<localizationPrefix>.definitions.<definitionName>.properties.<propertyName>.description`	`json.common.v1.definitions.Activity.properties.activityType.description`
Schema definition additionalProperty	title	`json.<localizationPrefix>.definitions.<definitionName>.additionalProperties.title`	`json.composite.v1.definitions.Headers.additionalProperties.title`
Schema definition additionalProperty	description	`json.<localizationPrefix>.definitions.<definitionName>.additionalProperties.description`	`json.composite.v1.definitions.Headers.additionalProperties.description`

Display key patterns for swagger.yaml files

Schema definitions and their properties (except for the top-level title and description properties) can be defined in `swagger.yaml` files. Therefore, the key patterns that can be used in `schema.json` files can also be used in `swagger.yaml` files. The one difference is that the display key patterns start with `schema.<localizationPrefix>`, rather than `json.<localizationPrefix>.` For a list of these key patterns, refer to the previous topic.

`swagger.yaml` files also support an additional set of documentation text. The following table lists the different types of text and the patterns for the display keys. Note that the base configuration makes use of some of these patterns, but not all of them.

Display key pattern	Example	Notes
info.title	swagger.<localizationPrefix>. swagger.claim.v1.info.title	
info.description	swagger.<localizationPrefix>. swagger.claim.v1.info.description	
info.termsOfService	swagger.<localizationPrefix>. swagger.claim.v1.info.termsOfService	
info.license.name	swagger.<localizationPrefix>. swagger.claim.v1.info.license.name	
tags.<name>.description	swagger.<localizationPrefix>.	"name" refers to the "name" property of the tag itself

Display key pattern	Example	Notes
	swagger.claim.v1.tags.claim.description	
tags.<name>. externalDocs.description	swagger.<localizationPrefix>. swagger.claim.v1.tags.claim. externalDocs.description	
securityDefinitions.<RefName>. description	swagger.<localizationPrefix>. swagger.claim.v1.securityDefinitions. oauth.description	"RefName" refers to the name the top-level security definition is referenced by
parameters.<RefName>. description	swagger.<localizationPrefix>. swagger.claim.v1.parameters.claimId. description	"RefName" refers to the name the top-level parameter is referenced by, not to the "name" property of the parameter
parameterSets.<RefName>. <in>.<name>.description	swagger.<localizationPrefix>. swagger.claim.v1.parameterSets. standardParameters. query.prettyPrint.description	"RefName" refers to the name of the parameter set, "in" and "name" refer to the respective values of the parameter within the set
responses.<RefName>. description	swagger.<localizationPrefix>. swagger.claim.v1.responses. standard200.description	"RefName" refers to the name the top-level response is referenced by
responses.<RefName>. headers.<HeaderName>. description	swagger.<localizationPrefix>. swagger.claim.v1.responses. standard200.headers. GW-Checksum.description	"RefName" refers to the name the top-level response is referenced by
externalDocs.description	swagger.<localizationPrefix>. swagger.claim.v1. externalDocs.description	
operations.<operationId>. summary	swagger.<localizationPrefix>. swagger.claim.v1.operations. getClaim.summary	
operations.<operationId>. description	swagger.<localizationPrefix>. swagger.claim.v1.operations. getClaim.description	
operations.<operationId>. externalDocs.description	swagger.<localizationPrefix>. swagger.claim.v1.operations. getClaim.externalDocs.description	
operations.<operationId>. parameters.<in>.<name>. description	swagger.<localizationPrefix>. swagger.claim.v1.operations. getClaim.parameters.query. customParam.description	Note that parameter names are not unique, only the combination of "in" and "name" is unique, so the property path for operation-level parameters must include both pieces
operations.<operationId>. responses.<Code>.description	swagger.<localizationPrefix>. swagger.claim.v1.operations. getClaim.responses. 200.description	

Display key pattern	Example	Notes
operations.<operationId>. responses.<Code>.headers. <HeaderName>.description	swagger.<localizationPrefix>. swagger.claim.v1.operations. getClaim.responses.200.headers. GW-Checksum.description	
paths.<Path>.parameters. <in>.<name>.description	swagger.<localizationPrefix>. swagger.claim.v1.paths.claims. _claimId_.parameters.path. claimId.description	

chapter 23

Providing locale specific content for a given locale

You can add localized content to a new or existing API element, or add a new locale.

Adding localized text for existing API elements

The base configuration provides several `schema.display_LOCALE.properties` files, but these files are incomplete. You can complete these files by adding localized text for existing API elements.

In this case, you must do the following:

- In each `schema.display_LOCALE.properties` file, add the necessary key/value pairs.
  1. The keys must match the required pattern to identify the appropriate API element.
  2. The value must be the localized text.

Adding localized text for new API elements

You can add localized text for new API elements for a locale that is already present in the base configuration. This could be required in the following situations:

- You extend a base configuration resource to include additional fields.
- You generate new endpoints using the REST endpoint Generator.

In this case, you must do the following:

1. In the relevant `schema.json` files, add the localization key prefix.
2. In the relevant `swagger.yaml` files, add the localization key prefix.
3. In each `schema.display_LOCALE.properties` file, add the necessary key/value pairs.
   a. The keys must match the required pattern to identify the appropriate API element.
   b. The value must be the localized text.

Adding a new locale

You can add localized text for a new locale. This could be required when you need to support a locale that is not present in the base configuration.

In this case, you must do the following:

1. Create a new `schema.display_LOCALE.properties` file in the `/modules/configuration/config/locale` directory.
2. In this file, add the necessary key/value pairs.
   a. The keys must match the required pattern to identify the appropriate API element.
   b. The value must be the localized text.

Note that each InsuranceSuite instance can have only one `schema.display.properties` file for a given locale.

chapter 24

Localized properties for multiple languages

InsuranceSuite apps can be configured to store localized values for multiple languages. PolicyCenter stores localized values in separate database tables.

You can work with localized values through Cloud API by marking the corresponding schema properties as localized. You can retrieve all or some language values for localized properties, and you can set localized values in POSTs and PATCHes.

For example, consider that an instance of PolicyCenter is configured to use both US English (en_US) and France French (fr_FR). The `name` property of a role is localized for both languages. You can make requests and responses that contain localized values for `name`:

```
{
  "data": {
    "attributes": {
        ...
        "name": "Privileged Administrator",
        "name_Localizations": {
            "en_US": "Privileged Administrator"
            "fr_FR": "Administrateur privilégié"
        ...
        }
    }
  }
}
```

Enabling localized properties

The schema property `x-gw-enableLocalizedProperties` must be `true` to work with localized properties.

This is a top-level schema property that is set to `true` by default in the `configuration/config/integration/schemas/gw/core/pl/framework/v1/framework_pl-1.0.schema.json` file. All Cloud API schemas in PolicyCenter inherit this schema directly or through another schema, so the feature is enabled by default for all Cloud APIs.

Note: Guidewire recommends that you not set `x-gw-enableLocalizedProperties` to `false` in Cloud API schema extension files. However, it can be set to `false` if you write an entirely new schema.

Configuring localized properties

To work with localized values in Cloud API:

- In the entity file, the column for the localizable value must have the `<localization/>` attribute
- The property in the schema must have `x-gw-localized` set to `true`
- The property in the schema must have `x-gw-localized` set to `true`
- The property in the schema must be a `string` and cannot have the `$ref` attribute

For example, the name of the assessment summary entity is localizable. The localized values are kept in the `assessmentsum_name_l10n` database table. Therefore, the entity's .eti file has a column that looks like this:

```
<entity
  ...
  <column
    desc="Name of the specific assessment service, model, or solution that generated this assessment summary"
    name="Name"
    nullok="true"
    type="shorttext">
    <localization
      tableName="assessmentsum_name_l10n"/>
  </column>
  ...
</entity>
```

The `name` property must also be marked as localized in the schema:

```
...
"AssessmentSummary: {
  "properties": {
    "name": {
      "title": "Name",
      "type": "string",
      "x-gw-localized": true,
      "x-gw-extensions": {
        "securityLevel": "public"
      }
...
```

Note that for this feature to work as expected, mappers and updaters must be properly configured. Specifically, the `path` attribute for the mapper and updater files for the property must be set to the entity property that corresponds to the database column (for example, `AssessmentSummary.Name`).

As a result of these configurations, you can now send GET, POST, and PATCH requests with the localized fields. Use the `includeLocalizations` query parameter to view localized values in responses. See the *Cloud API Consumer Guide* for more.

How localized properties are processed

PolicyCenter automatically generates the appropriate schemas, mappers, and updaters at runtime to handle localized properties. The following occurs automatically:

- **Schemas** - When JSON schemas are loaded, properties marked with `x-gw-localized` generate schema properties for localized values with `_Localizations` appended to the original property name. `securityLevel` and `defaultViews` attributes are copied over from the original property. A new `LocalizedValues` schema is generated to hold all localized values. For nullable values, the generated schema is `LocalizedValuesNullable`.
- **Mappers** - For each localized property, a corresponding mapping property is generated to map localized values into the JSON response. The localized properties inherit the `predicate` attribute from the original property.
- **Updaters** - Updater properties for localized values are generated to update localized values. These properties use the `LocalizedValues` schema and follow the same path as the original property. The following updater attributes are copied from the original updater's localized property to all updater localization properties:
  - `allowed`
  - `ignore`
  - `deferUpdate`
  - `afterGraphProperty`

- `handlerName`
- `preUpdateValidators`
- `postUpdateValidators`
- `handlerConfig`

Permissions to access localized values are the same as permissions to access original values. These permissions are configured in `<role name>_role.yaml` files. Localized properties do not have to be explicitly listed in these files, because localized values are automatically treated the same as values in the original language.

part 5

Generating extension endpoints

InsuranceSuite Cloud API provides endpoints for executing CRUD operations on base configuration entities. However, the base configuration endpoints may not be sufficient for insurers. Insurers may need Cloud API endpoints for extension entities. Insurers may also need endpoints for certain base configuration entities for which there are no base configuration endpoints.

The *REST endpoint generator* is a tool that generates endpoints for extension entities and for most base configuration entities that do not already have endpoints. The following topics discuss how to use the REST endpoint generator.

> **Note:** The REST endpoint generator is not used to generate LOB-specific endpoints for a line of business. For information on how to generate LOB-specific endpoints, see “Generating LOB-specific endpoints”.

chapter 25

The REST endpoint generator

The REST endpoint generator is a tool that insurers can use to create a set of generated endpoints for a data model entity that does not have endpoints. This could be either a custom entity, or a base configuration entity for which there are no endpoints. The tool generates a series of files that define the majority of the functionality for the endpoints. However, the insurer must complete some additional configuration.

- This topic provides an overview of the REST endpoint generator.
- For more information on how to run the REST endpoint generator, see "Running the REST endpoint generator" on page 125.
- For more information on configuring the resource definition files, see "Configuring the resource definition files" on page 135.
- For more information on configuring the glue and impl classes, see "Configuring glue and impl classes for generated endpoints" on page 147.
- For more information on configuring authorization, see "Configuring authorization for generated endpoints" on page 155.

  Note: Guidewire does not recommend adding inline arrays to schemas. If a given resource has an array of related resources and you want to expose those related resource in Cloud API, Guidewire recommends using the REST endpoint generator to create a separate set of child endpoints for the related resources.

REST endpoint generator overview

Insurers typically extend the data model of an InsuranceSuite application with new entities specific to their business model. These data model entities are referred to as custom entities.

There may also be entities in the base configuration which have no endpoints, but the insurer wants to expose information in these entities to Cloud API.

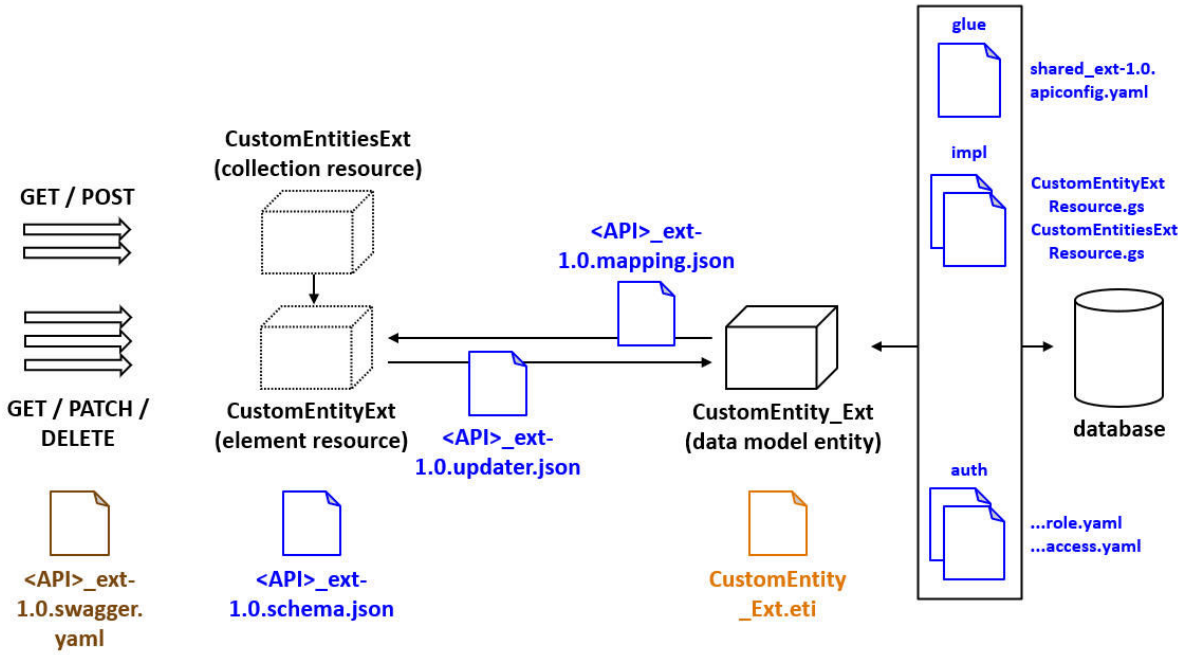
Insurers may want to expose these entities to Cloud API so that caller applications can:

- Retrieve data stored in these entities.
- Create new instances of these entities.
- Modify or delete these entities as needed.

To address this need, Cloud API includes the *REST endpoint generator*. It is a tool that generates a set of CRUD endpoints for an entity that does not have CRUD endpoints.

Architecture and the REST endpoint generator

When you run the REST endpoint generator for a given entity, the tool creates or modifies the files needed to support CRUD endpoints for the entity. The following diagram summarizes the files that are created or modified when running the tool for an entity whose name is `CustomEntity_Ext`.



Note the following conventions:

- <API> refers to the name of the API in which the endpoints have been placed.
- The data model entity file (`CustomEntity_Ext.eti`) appears in orange because it is not created or modified by the REST endpoint generator. It must exist before the generator is run.
- The swagger file (`<API>_ext-1.0.swagger.yaml`) appears in brown because it is modified by the REST endpoint generator, but further configuration is not required.
- The remaining files, which appear in blue, are created or modified by the REST endpoint generator. Further configuration to these files is either required or recommended.

REST endpoint generator restrictions

Which entities can you generate endpoints for?

The REST endpoint generator can be used only on entities that do not yet have endpoints. This includes:

- Custom entities for which you have not yet generated endpoints
- Base configuration entities that do not have endpoints in the base configuration

Note that, for some base configuration entities without endpoints, the REST endpoint generator will prevent you from generating endpoints.

How are generated resources named?

When generating resources, if the entity does not have an "_Ext" suffix, the REST endpoint generator appends an "Ext" or "ext" suffix to the name of the resource in file names and file contents. This is done for both custom entities and base configuration entities without endpoints. It is done to prevent conflicts between insurer configurations and future enhancements of Cloud API.

Which fields are included in the generated endpoints?

The REST endpoint generator generates fields for scalars only. It does not generate fields for foreign keys or arrays.

- If the entity has one or more foreign keys, each foreign key must be added manually. For more information, see "Adding foreign keys" on page 43.
- If the entity has an array to a custom entity, Guidewire recommends generating a separate set of endpoints for the custom entity. (For example, if `CustomParent_Ext` has an array of `CustomChild_Ext`, after you generate endpoints for `CustomParent_Ext`, generate endpoints for `CustomChild_Ext` as a child of `CustomParent_Ext`. The second set of endpoints would have paths that included `.../CustomParent_Ext/{customParentExtId}/CustomChilds_Ext...`.

  Note: Guidewire does not recommend adding inline arrays to schemas. If a given resource has an array of related resources and you want to expose those related resource in Cloud API, Guidewire recommends using the REST endpoint generator to create a separate set of child endpoints for the related resources.

What kind of endpoints are generated?

The REST endpoint generator generates CRUD endpoints only. It cannot be used to create business action POSTs, such as an `/assign` or `/submit` endpoint.

Process for generating CRUD endpoints for an entity

To generate CRUD endpoints for an entity, you must do the following:

1. Run the REST endpoint generator
   - Initially, the tool provides a series of prompts. The answers determine how the endpoints are defined.
   - Once all prompts have been answered, the tool generates and modifies files as needed. To help identify where configuration is needed, the tool adds "`TODO RestEndpointGenerator`" comments to files needing configuration.
2. Configure the files that define the resource. This may include any or all of the following:
   a. The swagger file
   b. The schema file
   c. The mapping file
   d. The updater file
3. Configure the glue and impl Gosu files. This includes:
   a. The apiconfig.yaml file.
   b. The element Resource file.
   c. The collection Resource file.
4. Configure the authorization files. This includes:
   a. The associated role.yaml files.
   b. The associated access.yaml files.

The following topics describe each of these steps in detail.

Special use cases

Integration graphs

An *integration graph* is a data model graph used by Guidewire App Events. It defines a set of business information to be sent to an external application as part of outbound integrations. For example, the Claim graph defines what information to send about a claim. For more information on integration graphs, see the *App Events Guide*.

When you generate endpoints for a custom entity, you can also add the custom entity's resource to an integration graph if the resource has a parent resource and that parent resource belongs to the integration graph. For more information, see “Additional considerations for generated endpoints” on page 163.

Base configuration entities

In most cases, you can generate endpoints for base configuration entities that do not have endpoints. In this case, there are some minor differences in the REST endpoint generator behavior. For more information, see “Base configuration entities” on page 165.

Subtyped entities

You can generate endpoints for a custom entity with subtypes. When you do, you can choose either "shared" or "separate" endpoints.

- You can generate "shared" endpoints, which may be appropriate when the majority of the information is declared at the supertype level. In this case, the resource has the supertype fields and the subtype fields from each subtype.
- You can generate "separate" endpoints, which may be appropriate when there is a sufficient amount of information at each subtype level. In this case, the resource has only the supertype fields. Subtype fields are omitted. To access information at the subtype level, you must re-run the REST endpoint generator for each subtype.

For information on generating endpoints for subtyped entities, see “Additional considerations for generated endpoints” on page 163.

Root resources

In most cases, when you generate endpoints for a custom entity, the custom entity is child to some existing parent entity. You access the associated REST resource through that parent resource.

For example, suppose you have a custom entity named `CustomEntity_Ext`. It is a child of the existing `Activity` entity. Information about `CustomEntity_Ext` instances are accessed through the parent `Activity`. In this case, the endpoints have this structure:

- GET `/common/v1/activities/{activityId}/custom-entity-ext`
- GET `/common/v1/activities/{activityId}/custom-entity-ext/{CustomEntityExtID}`

However, it is possible to generate endpoints for a custom entity as a root resource. In this case, you access the associated REST resource directly.

For example, suppose you have a custom entity named `CustomEntity_Ext`. The endpoints are generated with the associated resource being a root resource. In this case, the endpoints have this structure:

- GET `/common/v1/custom-entity-ext`
- GET `/common/v1/custom-entity-ext/{CustomEntityExtID}`

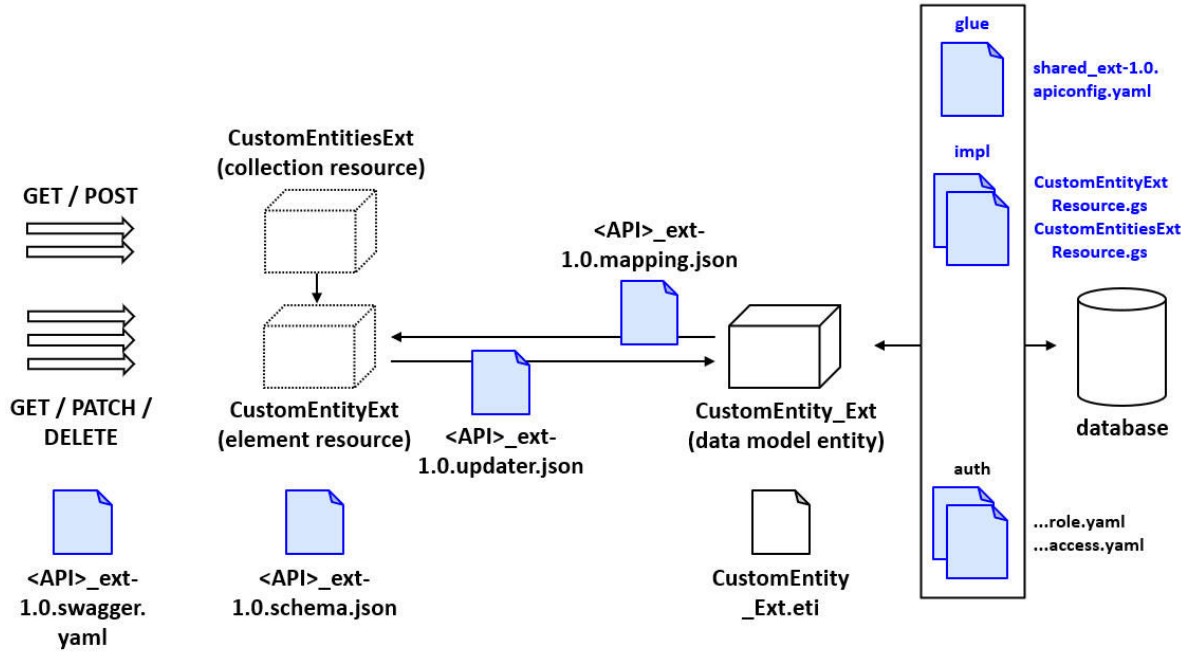
For more information on generating endpoints for root resources, see “Additional considerations for generated endpoints” on page 163.

chapter 26

Running the REST endpoint generator

The REST endpoint generator is a tool that creates a set of generated endpoints for custom data model entities. The tool generates a series of files that define the majority of the functionality for the endpoints. However, the insurer must complete some additional configuration.

This topic discusses how to run the REST endpoint generator. It creates or modifies the files highlighted in the following architecture diagram.



Configuration of the files are discussed in other parts of the documentation.

- For an overview of the REST endpoint generator, see “The REST endpoint generator” on page 121.
- For more information on configuring the resource definition files, see “Configuring the resource definition files” on page 135.
- For more information on configuring the glue and impl classes, see “Configuring glue and impl classes for generated endpoints” on page 147.

- For more information on configuring authorization, see “Configuring authorization for generated endpoints” on page 155.

Issues to consider before running the generator

The REST endpoint generator asks a series of questions, which determine the behaviors of the endpoints. Some questions have significant implications on this behavior. Guidewire encourages developers to consider the following issues prior to running the generator.

The API for the new endpoints

You must identify which API (such as Common or Admin) to add the endpoints to.

The Job API

Guidewire recommends adding endpoints to the Job API only if the underlying entity is effective-dated. Adding endpoints to the Job API for non-effective-dated entities is not recommended.

When you add endpoints to the Job API, the REST endpoint generator also adds the corresponding GET endpoints to the Policy API. The generator also adds appropriate PolicyCenter-specific query parameters, such as `asOfDate` and `jobVersion`, to the endpoints.

Any roles that are given GET access to endpoints in the Job API will also have access to the GET endpoints in the Policy API.

Schemas, mappers, updaters, Swagger parameters, and Swagger request/response envelope definitions are added to the `policyperiod` files, rather than `job` or `policy` files.

The Policy API

You cannot add endpoints directly to the Policy API. The only way to add endpoints to the Policy API is to add them to the Job API. Any GET endpoints added to the Job API are automatically copied to the Policy API.

The Account API, Admin API, and Common API

Endpoints for a custom entity can be added to the Account API, Admin API, or Common API only if the custom resource is not effective-dated.

The Product Definition API

If the entity corresponds to a system data table, it is automatically added to the Product Definition API. This is the only type of custom endpoint you can add to the Product Definition API.

The Composite API, the System Tools API, and the Test Util API

You cannot add custom endpoints to these APIs.

Summary of entity types and APIs

API	Add endpoints for effective-dated entities?	Add endpoints for non- effective-dated entities?
Account API	No	Yes
Admin API	No	Yes
Common API	No	Yes
Composite API	No	No
Job API	Yes	Not recommended

Policy API	Yes, but only by adding them to the Job API. Endpoints cannot be added directly to the Policy API.	No
Product Definition API	No	System tables only
System Tools API	No	No
Test Util API	No	No

The parent of the custom resource

In most cases, the custom entity is the child of an existing entity (such as a claim or an account). The custom entity may be a direct child of a single parent. But the custom entity could also be part of a hierarchy that is several levels deep. When the custom entity has multiple ancestors, you have a choice in the structure of the endpoint path.

For example, suppose you have a `CustomEntity_Ext` entity which is the child of AccountContact, and AccountContact is the child of Account. When you generate the endpoints, you can make the custom resource a descendent of its immediate parent (AccountContact), or a distant ancestor (Account).

Choosing the immediate parent

If you choose the immediate parent, then your endpoints look like this:

- GET `/account/{accountId}/contacts/{contactId}/custom-entities-ext`
- POST `/account/{accountId}/contacts/{contactId}/custom-entities-ext`
- GET `/account/{accountId}/contacts/{contactId}/custom-entities-ext/{customEntityExtId}`
- PATCH `/account/{accountId}/contacts/{contactId}/custom-entities-ext/{customEntityExtId}`
- DELETE `/account/{accountId}/contacts/{contactId}/custom-entities-ext/{customEntityExtId}`

With this approach, you cannot retrieve all `CustomEntity_Ext` instances for a single Account in one call. You would need to make multiple calls, one for each AccountContact.

When you create a new `CustomEntity_Ext`, the immediate parent (the AccountContact) is specified in the URL. Therefore, the new resource is linked to its immediate parent.

Choosing a distant ancestor

If you choose a distant ancestor, such as Account, then your endpoints look like this:

- GET `/account/{accountId}/custom-entities-ext`
- POST `/account/{accountId}/custom-entities-ext`
- GET `/account/{accountId}/custom-entities-ext/{customEntityExtId}`
- PATCH `/account/{accountId}/custom-entities-ext/{customEntityExtId}`
- DELETE `/account/{accountId}/custom-entities-ext/{customEntityExtId}`

With this approach, you can retrieve all `CustomEntity_Ext` instances for a single Account in one call.

When you create a new `CustomEntity_Ext`, the immediate parent (the AccountContact) is not specified in the URL. Therefore, the new resource is not automatically linked to its immediate parent. (It may be possible to link the new resource to its immediate parent by explicitly adding it to the request payload.)

Making the choice

From a technical standpoint, either choice is valid. The best choice depends on the types of GETs and POSTs you plan to execute most frequently.

Populating collections

One of the GET endpoints retrieves a collection. This collection can be retrieved using either a stream or a Gosu query.

A stream loads the entire collection into memory before manipulating it. Streams have the advantage of being objects that developers may be more familiar with. Filtering and sorting may be easier with stream-backed resources as the whole collection is loaded into memory. However, streams may degrade performance if the size of the collection is too large.

Gosu queries are expressions that are converted into SQL queries. Queries have a maximum number of elements that are loaded into memory at one time. Queries have the advantage of preserving performance if the size of the entire collection is large, as the collection is loaded in portions. However, if you are not familiar with Gosu, you may find it harder to write complex query logic.

Guidewire has the following recommendations:

- If the collection is likely to be large, use a Gosu query.
- Otherwise, use a stream.

With streams, you will need to write code to populate the stream. This is easy to do when the parent entity has an array of custom entities, as the array can be converted into a stream. Therefore, if you decide to use streams, you may want to add an array of custom entities to the parent, even if the application does not otherwise require an array.

External callers (such as insureds or producers) may not have access to third-party data on claims (such as contact and vehicle information related to a vehicle that the insured damaged). For these types of callers, access to specific fields may be restricted by additional accessible fields filters. For fields restricted by additional accessible fields filters, the behavior of sorts and filters changes based on whether the collection is populated by a stream or query. For more information, see “Sorting and filtering on accessible fields”.

Additional considerations

You must identify the API roles that will have GET, POST, PATCH, and DELETE access to the custom endpoints.

You can add the custom resource to an integration graph. For more information, see “Integration graphs” on page 163.

If the custom entity has subtypes, you can generate either a single set of shared endpoints to work with supertype and subtype behavior simultaneously, or you generate separate sets of shared endpoint that work with each supertype or subtype independently. For more information, see “Supertype entities” on page 167.

Naming conventions during generation

Entity Ext prefix or suffix

The REST endpoint generator expects custom entities to conform to the naming convention of ending with "Ext". (This convention is designed to prevent custom entities from conflicting with base configuration entities added in future releases.) If the custom entity does not end with Ext, the REST endpoint generator adds an "ext" suffix to the name. For example, suppose you have a custom entity named `CustomEntity`. After the REST endpoint generator runs:

- The element resource is named `CustomEntityExt`
- The collection resource is named `CustomEntitiesExt`

Plural name for resource

The REST endpoint generator provides a default name for the resource collection. Guidewire recommends naming this using the plural form of the custom entity name. The REST endpoint generator guesses the correct plural. If the guess is incorrect (such as guessing `CustomChildsExt` instead of `CustomChildrenExt`), you can modify the default.

Hyphenation in paths

The endpoint paths consist of custom entity name converted into lower case letters, numbers, and hyphens. Certain characters or character combinations in the entity name cause the path to be generated in a particular way.

- Upper-case letters are converted to lower letters. (The path for `Inspector_Ext` is `/inspector-ext`.)
- Underscores are converted to hyphens. (The path for `Inspector_Ext` is `/inspector-ext`.)
- Single capital letters in the middle of the name are preceded with a hyphen. (The path for `SecuritySystem_Ext` is `/security-system-ext`.)

- If the entity name contains multiple capital letters in a row (such as `BOPLine_Ext`), the REST endpoint generator guesses at the correct way to apply hyphenation (such as `/bop-line-ext`). If the guess is incorrect (such as being given `USBOPLine_Ext` and guessing `/usbop-line-ext` instead of `/us-bop-line`), you can modify the default.

Effective-dated entities

Guidewire recommends added only effective-dated entities to the Job API. Effective-dated entities use the PolicyCenter-specific `EffDatedRestElementResource` and `EffDatedRestListCollectionResource` classes.

When you generate endpoints for an effective-dated entity, the following prompts are suppressed:

- The prompt asking if the resource is a root resource or a child resource.
  - Effective-dated entities must be child to an existing parent resource.
- The prompt asking if the collection resource is backed by a stream or a query.
  - Effective-dated entities are always assumed to be stream-backed resources.
- The prompt asking if the resource is to be added to an integration graph.
  - Effective-dated entities are always added to the policyperiod graph.

System tables

If an entity is a system table, the REST endpoints generator automatically creates a single GET collection endpoint. The endpoint is a root resource endpoint in the Product Definition API, and its path starts with `/reference-data`.

When you generate endpoints for system tables, the following prompts are suppressed:

- The prompt asking which API to create the endpoint in.
  - System table endpoints are always created in the Product Definition API.
- The prompt asking if the resource is a root resource or a child resource.
  - System table endpoints are always root resource endpoints.
- The prompt asking if the collection resource is backed by a stream or a query.
  - System table endpoints are always query-backed resources.
- The prompt asking if the resource is to be added to an integration graph.
  - System table resources cannot be added to any graph.

Running the REST endpoint generator

You can run the REST endpoint generator either from Studio or from a command prompt. Once you start the REST endpoint generator, you cannot stop it until either it completes or you force it to quit (such as by pressing CTRL+C in a command prompt).

Running the REST endpoint generator from Studio

You can run the REST endpoint generator from Studio by creating a "run configuration" for it and then using that "run configuration".

Create a run configuration for the REST endpoint generator

Procedure

1. In Studio, select **Run** > **Edit Configurations**. Studio opens a **Run/Debug Configuration** dialog box.
2. In the upper right corner of the dialog box, click the +. In the **Add New Configuration** popup, click **Application**.
3. Provide the following values for each field. Note that some fields in the dialog box do not have labels:

a. **Name**: RESTEndpointGenerator

b. **JDK/JRE field**: (use the default value)

c. **VM options**:

```
-server
-ea
-Xdebug
-Djava.awt.headless=true
-Dgw.port=8180
-Xmx4g
-Dgw.server.mode=dev
-Dgwdebug=true
-Dgw.webapp.dir=idea/webapp
-Dgw.classpath.jar=true
-Dgw.plugins.gclasses.dir=idea-gclasses
-DgosuInit.supportDiscretePackages=true
```

d. **Main class**: com.guidewire.tools.rest.RestEndpointGenerator

e. **Program arguments**: (blank)

f. **Working directory**: (use the default value)

g. **Environment variables**: (blank)

4. Click **OK**.

Use the run configuration to run the REST endpoint generator

Procedure

1. Start the application.
2. In Studio, in the upper right-hand corner, there is a drop-down list. (By default, it is set to "Drop DB".) Set the value to "RESTEndpointGenerator".
3. Click the green play button to the right of the drop-down list.

Results

The REST endpoint generator loads and then displays the first prompt in the console window at the bottom of the Studio interface. You can enter responses in this window. For a list of the prompts, see “The REST endpoint generator prompts”.

Be aware that the REST endpoint generator loads the entire type system. Therefore, there may be several minutes between starting the generator and the appearance of the first prompt.

Running the REST endpoint generator from the command prompt

From the installation directory, open a command prompt and enter the following:

```
gwb restEndpointGenerator
```

The REST endpoint generator loads and then displays the first prompt. For a list of the prompts, see “The REST endpoint generator prompts” on page 130.

Be aware that the REST endpoint generator loads the entire type system. Therefore, there may be several minutes between executing the command and the appearance of the first prompt.

The REST endpoint generator prompts

The following section lists the prompts for the general case of generating endpoints for a generic child resource. All responses are case-sensitive.

Which entity?

```
Which data model entity would you like to generate endpoints for?
```

Specify the entity name.

Requiring the Ext suffix

If the specified entity has a name that does not start or end with "Ext", then the following prompt appears. It identifies that "Ext" will be added to the resource name.

```
Guidewire maintains the entity naming standard of Standard-IS-FNC-1321-DataModelNaming.
<entityName> does not start or end with 'Ext', therefore the REST endpoint
generator will add the suffix 'ext' to all REST generated code. Answer 'yes' to
acknowledge. y/n
```

You must acknowledge this by entering "y". If you do not enter "y", the generator throws an illegal state exception and stops running.

The default element resource name

```
The default element resource name for this entity is '<defaultElementResourceName>'. If you
want to use a different name, specify it and press Enter. To accept the default, just
press Enter.
```

The REST endpoint generator identifies the default name it will use for the element resource. You can specify a different name is desired.

The default collection resource name

```
The default plural for this entity is <defaultCollectionResourceName>. If you want to use a different plural,
specify it and press Enter. To accept the default, just press Enter.
```

Specify a custom name for the collection resource, or press Enter to accept the default name.

Requiring the Ext suffix (for the collection resource)

If the specified collection resource has a name that does not start end with "Ext", then the following prompt appears. It identifies that "Ext" will be added to the resource name.

```
Guidewire maintains the entity naming standard of Standard-IS-FNC-1321-DataModelNaming.
<customCollectionResourceName> does not start or end with 'Ext', therefore the REST endpoint
generator will add the suffix 'ext' to it in all REST generated code. Answer 'yes' to
acknowledge. y/n
```

You must acknowledge this by entering "y". If you do not enter "y", the generator throws an illegal state exception and stops running.

Default hyphenation

```
The url string for <entityName> will use the following hyphenation: <defaultValue>.
If you would like to use a different hyphenation, please enter it here (leave blank to use
the default, all capitalized letters will be converted to lowercase):
```

If the name of the entity has multiple capital letters and/or numbers in a row, then the REST endpoint generator shows the default hyphenation for the endpoint paths (such as `/bop-line-ext` for an entity named `BOPLine_Ext`). You can specify a custom hyphenation approach (such as `/bopline-ext`) or press Enter to accept the default.

Which API?

```
Which API should the endpoints be added to?
```

Specify the API name. The technical name is specified in lower case, such as "common" for the Common API or "admin" for the Admin API.

Each API has restrictions and recommendations around what can be added to it:

- Endpoints for effective-dated entities can be added only to:
  - Job API (the generated GETs are also added to the Policy API)
- Endpoints for non-effective-dated entities can be added only to:
  - Account API
  - Admin API
  - Common API
- Endpoints cannot be added to:
  - Composite API
  - Product Definition API (except for reference data)
  - System Tools API
  - Test Util API

(Technically speaking, non-effective-dated entities can be added to the Job API. But Guidewire does not recommend doing this.)

For more information, see “The API for the new endpoints” on page 126.

Root endpoint or child endpoint?

```
Is the <entityName> endpoint the (r)oot of the endpoint path (e.g. GET /activities)
or the (c)hild of another resource (e.g. GET /activities/{activityId}/notes)? r/c
```

Enter "c" to make the custom resource a child of some existing parent resource. (This prompt is not presented for effective-dated entities. Effective-dated entities cannot be root resources.)

For more information, see “The parent of the custom resource” on page 127.

```
What is the resource name of the parent?
```

Enter the resource name of the parent.

- This is the value of the `resourceType` from the parent resource's schema.
- This must be an element resource (such as `Activity`), not a collection (such as `Activities`).
- In order to generate child endpoints, there must already be a set of CRUD endpoints for the parent resource.

Be aware that, for most endpoints, the resource name is the same as what appears in the endpoint path. For example, the resource name for GET `/activities/{ActivityId}` is `Activity`. But sometimes, the endpoint path differs from the resource name. For example, the resource name for GET `/contacts/{contactId}` is not `Contact`, but rather `AccountContact`. To verify you are using the correct name, check the `resourceType` from the parent resource's schema.

For more information, see “The parent of the custom resource” on page 127.

Stream-back collection or query-back collection?

```
Is the collection resource backed by a (s)tream or (q)uery?
```

Select whether your collection is loaded using a stream or a Gosu query. (This prompt is not presented for effective-dated entities. Effective-dated entities are always backed by streams.) For more information, see “Populating collections” on page 127.

Endpoint access

```
Which roles can access the GET collection and GET element endpoint?
Here are the options [<list-of-roles>]
Enter the values separated by comma. If you do not want to specify roles, just press Enter.
```

The REST endpoint generator lists the available roles. Enter a comma-separated list of roles that will have GET access.

Any roles that are given GET access to endpoints in the Job API will also have access to the GET endpoints in the Policy API.

Note that you do not need to answer any of the authorization prompts. You can press Enter for each prompt. However, this only bypasses the coding done by the REST endpoint generator. This does not bypass the need to configure authorization for the endpoints.

```
Which roles can access the POST collection endpoint?
Here are the options [<list-of-roles>]
Enter the values separated by comma. If you do not want to specify roles, just press Enter.
```

Enter a comma-separated list of roles that will have POST access. Note that GET access is required for POST access. Thus, any role which was not given GET access cannot be given POST access.

```
Which roles can access the PATCH element endpoint?
Here are the options [<list-of-roles>]
Enter the values separated by comma. If you do not want to specify roles, just press Enter.
```

Enter a comma-separated list of roles that will have PATCH access. Note that GET access is required for PATCH access. Thus, any role which was not given GET access cannot be given PATCH access.

```
Which roles can access the DELETE element endpoint?
Here are the options [<list-of-roles>]
Enter the values separated by comma. If you do not want to specify roles, just press Enter.
```

Enter a comma-separated list of roles that will have DELETE access. Note that GET access is required for DELETE access. Thus, any role which was not given GET access cannot be given DELETE access.

Integration graph

```
Should <CustomEntity> be added to an integration graph?
```

Enter "y" or "n". For more information on adding custom resources to integration graphs, see “Additional considerations for generated endpoints” on page 163.

Completion of the script

After the REST endpoint generator script successfully completes, it lists information about the files it created or modified similar to the following example.

```
2022-03-18 16:42:42,817 INFO Beginning endpoint generation
2022-03-18 16:42:42,818 INFO Generating resource classes
2022-03-18 16:42:42,832 INFO Modifying configuration file <api_collection>_ext-1.0.swagger.yaml
2022-03-18 16:42:42,868 INFO Modifying configuration file <api_collection>_ext-1.0.schema.json
2022-03-18 16:42:42,877 INFO Modifying configuration file <api_collection>_ext-1.0.mapping.json
2022-03-18 16:42:42,880 INFO Modifying configuration file <api_collection>_ext-1.0.updater.json
2022-03-18 16:42:42,885 INFO Modifying configuration file shared_ext-1.0.apiconfig.yaml
2022-03-18 16:42:42,903 INFO Modifying configuration file default_ext-1.0.access.yaml
2022-03-18 16:42:42,907 INFO Modifying configuration file internal_ext-1.0.access.yaml
2022-03-18 16:42:42,909 INFO Modifying configuration file accountnumbers_ext-1.0.access.yaml
2022-03-18 16:42:42,917 INFO Modifying configuration file <role>.role.yaml
2022-03-18 16:42:42,939 INFO Finished endpoint generation
```

Completing the configuration

Some files created or modified by the REST endpoint generator require further configuration. For other files, there may be optional configurations you wish to add. For all of these files, the tool adds one or more "`TODO RestEndpointGenerator`" comments to help identify the locations of additional configuration work.

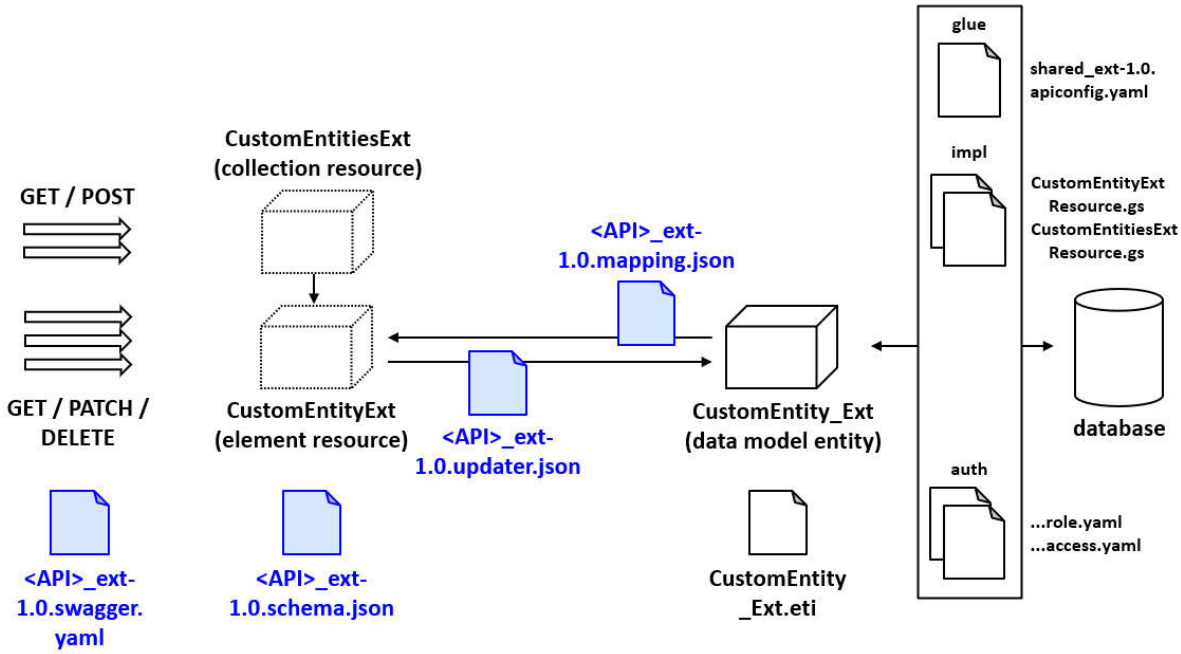
- For more information on configuring the resource definition files, see “Configuring the resource definition files”.
- For more information on configuring the glue and impl files, see “Configuring glue and impl files”.
- For more information on configuring authorization, see “Configuring authorization for generated endpoints” on page 155.

chapter 27

Configuring the resource definition files

The REST endpoint generator is a tool that creates a set of generated endpoints for custom data model entities. Most of the files generated by the REST Endpoint Generator are incomplete and require further configuration.

This topic discusses how to configure the resource definition files. They are the files highlighted in the following architecture diagram.



Configuration of the other files are discussed in other parts of the documentation.

- For an overview of the REST endpoint generator, see “The REST endpoint generator” on page 121.
- For more information on how to run the REST endpoint generator, see “Running the REST endpoint generator” on page 125.

- For more information on configuring the glue and impl classes, see “Configuring glue and impl classes for generated endpoints” on page 147.
- For more information on configuring authorization, see “Configuring authorization for generated endpoints” on page 155.

The resource definition files

The schema, mapping, and updater files

For a given Cloud API resource, there are three files that define the structure of the resource and how it connects to the corresponding data model entity:

- The **schema file**, which defines the schema used by the element and collection resources.
- The **mapping file**, which defines how information is mapped from the data model entity to the element resource. This information is used for GETs, and for the responses of POSTs and PATCHes.
- The **updater file**, which defines how information is mapped from the element resource to the data model entity. This information is used for POSTs and PATCHes.

For some types of data model fields, the modifications made by the REST endpoint generator are not complete. The places where additional coding may be needed are flagged with the following comment:

```
TODO RestEndpointGenerator
```

You can search for these "TODO RestEndpointGenerator" comments in Studio to complete the configuration.

Common patterns used in the schema, mapping, and updater files

Generally speaking, fields in a data model entity follow one of the behaviors listed in the following table. For each behavior, there is a pattern for how that field is referenced in the schema definition files.

Field behavior	Example	Schema file	Mapping file	Updater file
Field is not exposed to Cloud API	Application internal information, such as `<entity>.CreateUser`	Omitted	Omitted	Omitted
Field is read-only	The `<entity>.id` field	Included; declared as `readOnly`	Included	Omitted
Field must be set during creation and cannot be changed thereafter	`Claim.LossDate`	Included; declared as `requiredForCreate`, `x-gw-nullable: false`, and `createOnly`	Included	Included
Field must be set during creation, but can be changed later	`Document.Status`	Included; declared as `requiredForCreate`, and `x-gw-nullable: false`	Included	Included
Field can be set or omitted during creation, and can later be changed	`Activity.Description`	Included	Included	Included

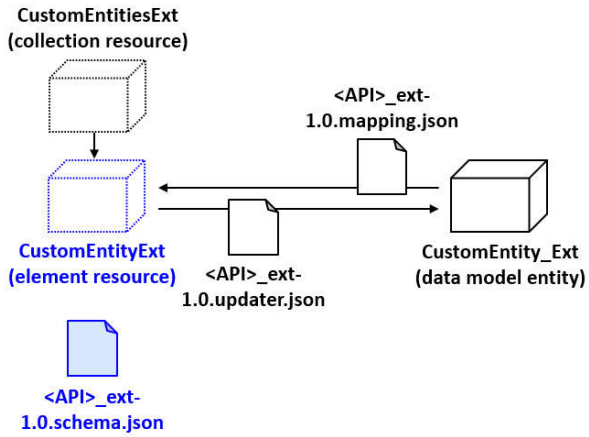
The swagger file

There is also a **swagger file**, which defines the endpoints themselves (the paths, operations, and associated resources). The REST endpoint generator adds code to this file. There is no requirement to modify this file, and there are no "TODO RestEndpointGenerator" comments in this file. However, a developer may wish to modify the file for the following reasons:

- To change the API documentation text, such as the title and description of the endpoints
- To remove an operation, such as DELETE

Configuring the schema file for generated endpoints

Within the context of Cloud API, a *schema file* defines the structure of one or more resources. This information is used for GETs, POSTs, and PATCHes.



The following sections provide an overview of configuring schema files for generated endpoints. For a complete description of how to configure schema files, see “Endpoint architecture” on page 15.

Overview of schema file syntax

A schema file contains a `definitions` section that lists one or more schemas. Each schema is used to structure an API resource. For each schema, the following attributes are specified:

- A `title` and `description`, which is used for API definition documentation.
- The resource `type`, which is typically set to `object`.
- A list of `properties`. Each property includes:
  - A `title` and `description`, which is used for API definition documentation.
  - A `type`, which defines the JSON datatype of the property (for scalar values).
  - A `ref`, which defines the URI reference for the property (for compound values such as typekeys).
  - Optional attributes as needed for the business functionality of the property.

For example, the following is a portion of the schema for the base configuration `Activity` resource:

```
"Activity": {
  "title": "Activity",
  "description": "An `Activity` is an assignable item that represents a task to be done, a decision to be made, or
information to be aware of",
  "type": "object",
  "properties": {
    "activityPattern": {
      "title": "Activity pattern",
      "description": "The code of the `ActivityPattern` used to create this activity and set its initial values",
      "type": "string"
      ...
      }
    },
```

Modifications made to the schema file

The base configuration includes a set of API-specific extension schema files. The purpose of these files is to define schema information for custom resources for a given API. These files are named using the pattern

`<API>_ext-1.0.schema.json`, where <API> is the internal name of the API. For example, the `common_ext-1.0.schema.json` file is used to define schema information for custom resources in the Common API. You can access extension schema files in Studio through the **integration -> schemas -> ext -> <API>.v1** node.

(There is an exception to the previous statement. When you add endpoints to the Job API, schema modifications are not made to a `job_ext-1.0.schema.json` file, but rather to the `policyperiod_ext-1.0.schema.json` file.)

Resource-level information

The REST endpoint generator adds the following resource-level information to the schema:

```
"<resourceName>": {
  "title": "<custom entity name>",
  "description": "<custom entity name>",
  "type": "object",
  "properties": {
      ...
  }
}
```

The id field

The REST endpoint generator adds an `id` field to the `properties` section of the schema.

```
"properties": {
  "id": {
    "title": "ID",
    "description": "The unique identifier of this element",
    "type": "string",
    "readOnly": true
  },
  ...
}
```

Data model fields added by the REST endpoint generator

The REST endpoint generator also adds the following types of fields to the `properties` section of the schema:

- Scalar fields whose datatype resolves to a String, Boolean, Integer, BigDecimal, or DateTime
- Compound data fields whose datatype resolves to a MonetaryAmount, CurrencyAmount or Typekey

For each field:

- A `title`, `description`, and `type` property are added
  - If the data model entity has values for `title` and `description`, the REST endpoint generator uses those for the `title` and `description` properties. Otherwise, it generates a default value based on the entity's name.
- If the data model field has `nullok` set to true, the REST endpoint generator also specifies

  ```
  "x-gw-nullable":
         true
  ```

  for the schema field.

For example, suppose you generated endpoints for a `CustomEntity_Ext` entity with the following fields. None of the fields have `title` or `description` information. All of which have `nullok` set to true:

- `Description` (a varchar field)
- `IsActive` (a Boolean field)
- `ActiveDate` (a datetime field)
- `ActiveCount` (an integer field)

The REST endpoint generator adds the following to the schema:

```
"properties": {
  "activeCount": {
    "title": "Active count",
```

```
        "description": "Active count",
        "type": "integer",
        "x-gw-nullable": true
      },
      "activeDate": {
        "title": "Active date",
        "description": "Active date",
        "type": "string",
        "format": "date-time",
        "x-gw-nullable": true
      },
      "description": {
        "title": "Description",
        "description": "Description",
        "type": "string",
        "x-gw-nullable": true
      },
      "isActive": {
        "title": "Is active",
        "description": "Is active",
        "type": "boolean",
        "x-gw-nullable": true
      },
      ...
    }
  }
```

How other data model settings affect schema field generation

For any given column in the data model:

- If the `getterScriptability` field is set to anything other than `all`, the REST endpoint generator does not create a field in the schema for the column.
- If the `setterScriptability` field setting is set to anything other than `all`, the REST endpoint generator adds `"readOnly": true` to the schema field.
- If `nullok` is set to false, the REST endpoint generator adds `"requiredForCreate": true` to the schema field.

For example, suppose you generated endpoints for a CustomEntity_Ext entity with the following fields:

- `GetterHidden` - A Boolean whose `getterScriptability` field is set to `hidden`
- `SetterHidden` - A Boolean whose `setterScriptability` field is set to `hidden`
- `NullOkFalse` - A Boolean whose `nullOk` field is set to `false`

The REST endpoint generator adds the following to the schema. Note that there is no schema field for the `getterHidden` field:

```
  "properties": {
    "nullOkFalse": {
      "title": "Null ok false",
      "description": "Null ok false",
      "type": "boolean",
      "x-gw-extensions": {
        "requiredForCreate": true
      }
    },
    "setterHidden": {
      "title": "Setter hidden",
      "description": "Setter hidden",
      "type": "boolean",
      "readOnly": true
    },
    ...
  }
}
```

Additional configuration work to do

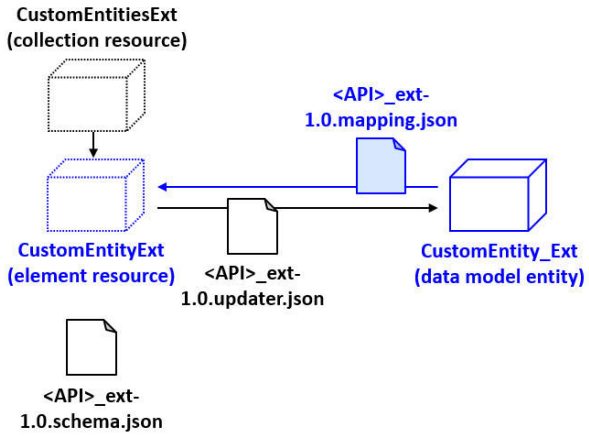
You are responsible for the following configuration work:

- Removing any schema fields that were created by the REST endpoint generator but which correspond to fields that are not to be exposed to Cloud API
- Adding definitions for any fields that must be exposed to Cloud API but that were not created by the REST endpoint generator

For more information on how to configure scalar and compound datatype properties in a schema, see “Adding scalars” on page 33 and “Adding compound datatypes” on page 39.

Configuring the mapping file for generated endpoints

Within the context of Cloud API, a *mapping file* contains mappers that define how information is mapped from a data model entity to the corresponding element resource. This information is used for GETs, and for the responses of POSTs and PATCHes.



The following sections provide an overview of configuring mapping files for generated endpoints. For a complete description of how to configure mapping files, see “Endpoint architecture” on page 15.

Overview of mapping file syntax

A mapping file contains a `mappers` section that lists one or more API resources. For each resource, the following attributes are specified:

- The `schemaDefinition` that defines the structure of the resource
  - This references a schema declared in a schema.json file.
- The data model entity that serves as the `root` for mapping information for this resource.
- A list of `properties`
  - Each property includes a `path` attribute. This is a Gosu expression that is used to populate the value of the property. Typically, this expression returns a value from the entity defined in the `root` attribute.
  - Depending on the nature of the property, there may be additional attributes.

For example, the following is a portion of the mapper for the base configuration `Activity` resource:

```
"Activity": {
  "schemaDefinition": "Activity",
  "root": "entity.Activity",
  "properties": {
    "closeDate": {
      "path": "Activity.CloseDate"
    },
    "description": {
      "path": "Activity.Description"
    },
    "mandatory": {
      "path": "Activity.Mandatory"
    },
    ...
```

Note the following:

- This resource is defined in the schema who name is `Activity` (This schema is defined in some other schema.json file.)

- The root for the resource mapping is `entity.Activity`.
- For each instance of the resource:
  - The `closeDate` property is set to the `Activity` entity's `CloseDate` field.
  - The `description` property is set to the `Activity` entity's `Description` field.
  - The `mandatory` property is set to the `Activity` entity's `Mandatory` field.

Modifications made to the mapping file

The base configuration includes a set of API-specific extension mapping files. The purpose of these files is to define mapper information for custom resources for a given API. These files are named using the pattern `<API>_ext-1.0.mapping.json`, where <API> is the internal name of the API. For example, the `common_ext-1.0.mapping.json` file is used to define mapper information for custom resources in the Common API. You can access extension mapping files in Studio through the **integration -> mappers -> ext -> <API>.v1** node.

(There is an exception to the previous statement. When you add endpoints to the Job API, mapper modifications are not made to a `job_ext-1.0.mapping.json` file, but rather to the `policyperiod_ext-1.0.mapping.json` file.)

Resource-level information

The REST endpoint generator adds the following resource-level information to the mapping file:

```
"<resourceName>": {
  "schemaDefinition": "<schemaNameForResource>",
  "root": "entity.<customEntity>",
  "properties": {
    ...
  }
}
}
```

Mappers added by the REST endpoint generator

The REST endpoint generator also adds mappers for any of the fields it added to the schema, including the `id` field.

For example, suppose you generated endpoints for a `CustomEntity_Ext` entity with the following fields:

- `Description` (a varchar field)
- `IsActive` (a Boolean field)
- `ActiveDate` (a datetime field)
- `ActiveCount` (an integer field)

The REST endpoint generator adds the following to the mapping file:

```
  "properties": {
    // TODO RestEndpointGenerator : Add mapper properties here
    "activeCount": {
      "path": "CustomEntity_Ext.ActiveCount"
    },
    "activeDate": {
      "path": "CustomEntity_Ext.ActiveDate"
    },
    "description": {
      "path": "CustomEntity_Ext.Description"
    },
    "id": {
      "path": "CustomEntity_Ext.RestId"
    },
    "isActive": {
      "path": "CustomEntity_Ext.IsActive"
    }
    ...
  }
}
```

Additional configuration work to do

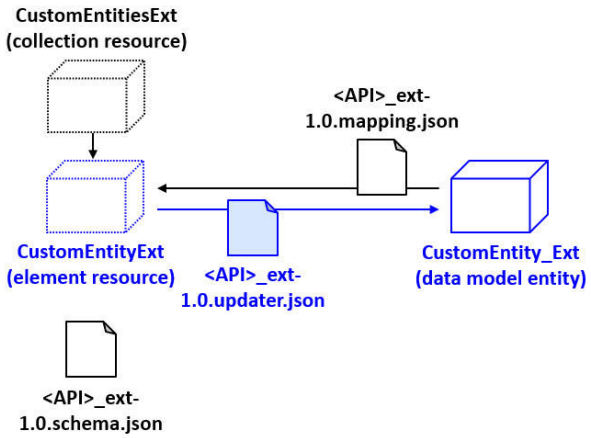
You are responsible for the following configuration work:

- Removing any mappers that were created by the REST endpoint generator but which correspond to fields that are not to be exposed to Cloud API
- Adding mappers for any fields that must be exposed to Cloud API but that were not created by the REST endpoint generator

For more information on how to configure scalar and compound datatype properties in a mapping file, see “Adding scalars” on page 33 and “Adding compound datatypes” on page 39.

Configuring the updater file for generated endpoints

Within the context of Cloud API, an *updater file* defines how information is mapped from an element resource to the corresponding data model entity. This information is used for POSTs and PATCHes.



The following sections provide an overview of configuring updater files for generated endpoints. For a complete description of how to configure mapping files, see “Endpoint architecture” on page 15.

Overview of updater file syntax

An updater file contains an `updaters` section that lists one or more API resources. For each resource, the following attributes are specified:

- The `schemaDefinition` that defines the structure of the resource
  - This references a schema declared in a schema.json file.
- The data model entity that serves as the `root` for mapping information for this resource.
- A list of `properties`
  - Each property includes a `path` attribute. This defines, for each resource property, the data model entity property into which the resource field property is to be written.
  - Depending on the nature of the property, there may be additional attributes.

For example, the following is a portion of the updater for the base configuration `Activity` resource:

```
"Activity": {
  "schemaDefinition": "Activity",
  "root": "entity.Activity",
  "properties": {
    "description": {
      "path": "Activity.Description"
    },
    "mandatory": {
      "path": "Activity.Mandatory"
    },
    ...
```

Note the following:

- This resource is defined in the schema who name is `Activity` (This schema is defined in some other schema.json file.)
- The root for the resource mapping is `entity.Activity`.
- For each instance of the resource:
  - The value of the `description` property is written to the Activity entity's `Description` field.
  - The value of the `mandatory` property is written to the Activity entity's `Mandatory` field.

Note that there may be properties that appear in the mapping file but not the updater file. This typically occurs with properties that are read-only. For example, the `Activity` entity has a `closeDate` property, which the application sets when the activity is closed. This property appears in the mapping file, as it can be read. But it does not appear it the updater file because it cannot be written to.

Modifications made to the updater file

The base configuration includes a set of API-specific extension updater files. The purpose of these files is to define updater information for custom resources for a given API. These files are named using the pattern `<API>_ext-1.0.updater.json`, where <API> is the internal name of the API. For example, the `common_ext-1.0.updater.json` file is used to define updater information for custom resources in the Common API. You can access extension mapping files in Studio through the **integration -> mappers -> ext -> <API>.v1** node.

(There is an exception to the previous statement. When you add endpoints to the Job API, updater modifications are not made to a `job_ext-1.0.updater.json` file, but rather to the `policyperiod_ext-1.0.updater.json` file.)

Resource-level information

The REST endpoint generator adds the following resource-level information to the updater file:

```
"<resourceName>": {
  "schemaDefinition": "<schemaNameForResource>",
  "root": "entity.<customEntity>",
  // TODO RestEndpointGenerator : Add updater properties here
  ...
}
```

Updaters added by the REST endpoint generator

The REST endpoint generator also adds updaters for any of the fields in the data model entity whose `setterScriptability` is set to `all`. It does not create an updater for the `id` field that it added to the schema. This is because `id` values cannot be updated.

For example, suppose you generated endpoints for a `CustomEntity_Ext` entity with the following fields:

- `Description` (a varchar field)
- `IsActive` (a Boolean field)
- `ActiveDate` (a datetime field)
- `ActiveCount` (an integer field)

The REST endpoint generator adds the following to the updater file:

```
"properties": {
  "activeCount": {
    "path": "CustomEntity_Ext.ActiveCount"
  },
  "activeDate": {
    "path": "CustomEntity_Ext.ActiveDate"
  },
  "description": {
    "path": "CustomEntity_Ext.Description"
  },
  "isActive": {
    "path": "CustomEntity_Ext.IsActive"
  },
  ...
}
```

Additional configuration work to do

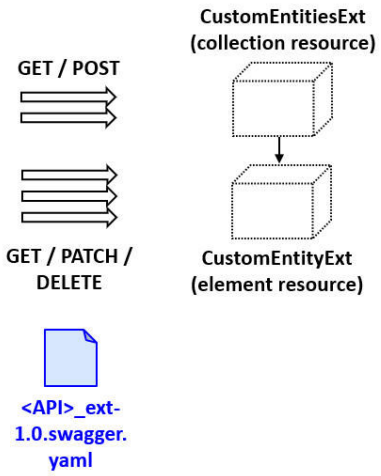
You are responsible for the following configuration work:

- Removing any updaters that were created by the REST endpoint generator but which correspond to fields that are not to be writeable from Cloud API
- Adding updaters for any fields that must be writeable to Cloud API but that do not have updaters

For more information on how to configure scalar and compound datatype properties in an updater file, see "Adding scalars" on page 33 and "Adding compound datatypes" on page 39.

Configuring the swagger file for generated endpoints

Within the context of Cloud API, a *swagger file* defines the endpoints and operations for a given API.



The following sections provide an overview of configuring swagger files for generated endpoints. For a complete description of how to configure swagger files, see "Swagger and apiconfig files" on page 25.

Overview of swagger file syntax

A swagger file contains a `paths` section that lists one or more endpoint paths. For each path, the following information is specified:

- Any path or query parameters
- Each operation supported for the path, and information about the operation

Modifications made to the swagger file

The base configuration includes a set of API-specific extension swagger files. The purpose of these files is to define swagger information for custom resources for a given API. These files are named using the pattern `<API>_ext-1.0.swagger.json`, where <API> is the internal name of the API. For example, the `common_ext-1.0.swagger.json` file is used to define swagger information for custom resources in the Common API.

(There is an exception to the previous statement. When you add endpoints to the Job API, swagger modifications are not made to a `job_ext-1.0.swagger.json` file, but rather to the `policyperiod_ext-1.0.swagger.json` file.)

When you generate endpoints for a custom entity, the REST endpoint generator adds code to the corresponding swagger extension file.

For example, suppose you generate endpoints for a `CustomEntity_Ext` custom entity. The parent of this entity is Account, and the endpoints are placed in the Account API. The REST endpoint generator add the following code to the `account_ext-1.0.swagger.json` file:

```
paths:
  "/account/{accountId}/custom-entities-ext":
    parameters:
    - $ref: "#/parameters/accountId"
    get:
      summary: "Retrieve a collection of custom entities ext"
      description: "Retrieve a collection of custom entities ext"
      operationId: getCustomEntitiesExt
      x-gw-extensions:
        childResourceType: CustomEntityExt
        operationType: get-collection
        resourceType: CustomEntitiesExt
      x-gw-parameter-sets: get-collection
      responses:
        "200":
          description: "Successful response"
          schema:
            $ref: "#/definitions/CustomEntityExtList"
    post:
      summary: "Create a new custom entity ext"
      description: "Create a new custom entity ext"
      operationId: createCustomEntityExt
      x-gw-extensions:
        childResourceType: CustomEntityExt
        operationType: post-collection
        resourceType: CustomEntitiesExt
      parameters:
      - name: body
        in: body
        required: true
        schema:
          $ref: "#/definitions/CustomEntityExtRequest"
      x-gw-parameter-sets: post-collection
      responses:
        "201":
          description: "The details of the newly-created CustomEntityExt"
          schema:
            $ref: "#/definitions/CustomEntityExtResponse"
  "/accounts/{accountId}/custom-entities-ext/{customEntityExtId}":
    parameters:
    - $ref: "#/parameters/accountId"
    - $ref: "#/parameters/customEntityExtId"
    get:
      summary: "Retrieve details of a custom entity ext"
      description: "Retrieve details of a custom entity ext"
      operationId: getCustomEntityExt
      x-gw-extensions:
        operationType: get-element
        resourceType: CustomEntityExt
      x-gw-parameter-sets: get-element
      responses:
        "200":
          description: "Successful response"
          schema:
            $ref: "#/definitions/CustomEntityExtResponse"
    patch:
      summary: "Update a custom entity ext"
      description: "Update a custom entity ext"
      operationId: updateCustomEntityExt
      x-gw-extensions:
        operationType: patch-element
        resourceType: CustomEntityExt
      parameters:
      - name: body
        in: body
        required: true
        schema:
          $ref: "#/definitions/CustomEntityExtRequest"
      x-gw-parameter-sets: patch-element
      responses:
        "200":
          description: "Successful response"
          schema:
            $ref: "#/definitions/CustomEntityExtResponse"
    delete:
      summary: "Delete a custom entity ext"
      description: "Delete a custom entity ext"
      operationId: deleteCustomEntityExt
      x-gw-extensions:
        operationType: delete-element
        resourceType: CustomEntityExt
      x-gw-parameter-sets: delete-element
      responses:
```

```
        "204":
          description: "Successful deletion"
```

Note that this text is functionally complete and there are no "TODO RestEndpointGenerator" comments in the file. However, a developer may wish to optionally modify the file.

Modifying API documentation text

Every path and response has `summary` and `description` information. This text is used by API definition display tools, such as Swagger UI. Developers may wish to modify the `summary` and `description` values created by the REST endpoint generator to improve the user experience for other developers who are consuming these endpoints.

Removing operations

The REST endpoint generator always generates a collection endpoint with a GET and POST operation, and an element endpoint with a GET, PATCH, and DELETE operation. If you do not want any of these operations, you can remove them from the swagger file.

For example, suppose an insurer wants endpoints to support `CustomEntity_Ext`, but they do not want to expose the ability to delete `CustomEntity_Ext` instances. In this case, the developer could remove the `"delete"` declaration from the swagger file.

Completing the configuration

Several of the files that are created or modified by the REST endpoint generator require further configuration. For each file requiring configuration, the tool adds one or more `"TODO RestEndpointGenerator"` comments to help identify where configuration is needed.

In addition to the resource definition files, you must also configure the glue and impl files and the authorization files.

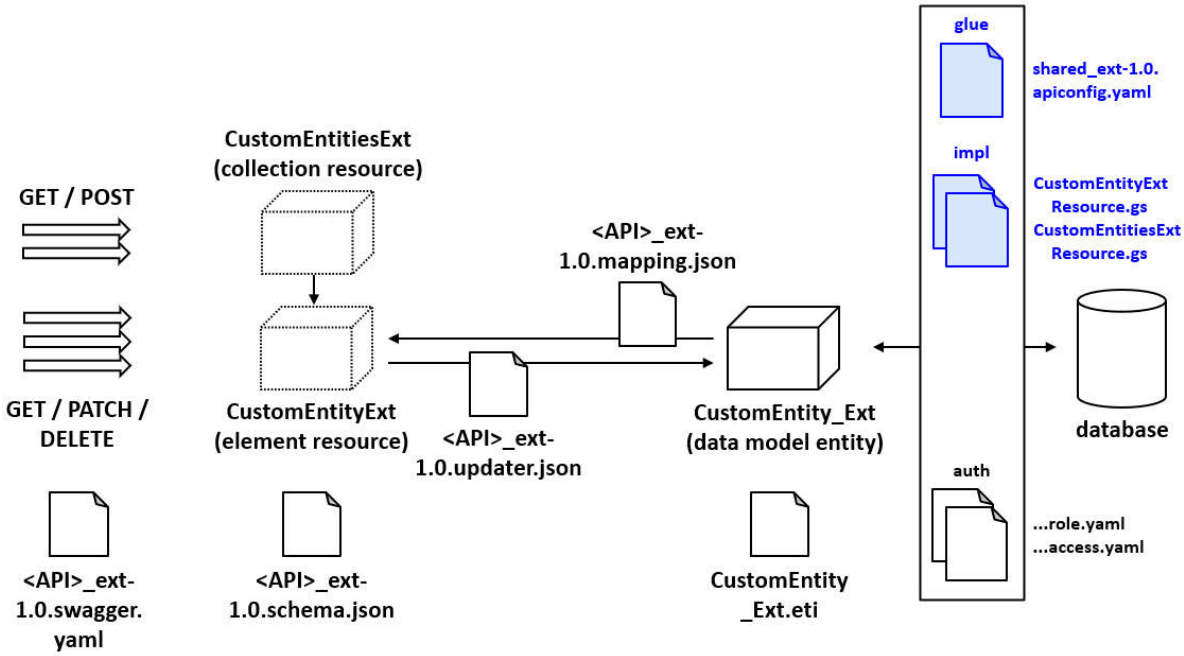
- For more information on configuring the glue and impl files, see “Configuring glue and impl files”.
- For more information on configuring authorization, see “Configuring authorization for generated endpoints”.

chapter 28

Configuring glue and impl classes for generated endpoints

The REST endpoint generator is a tool that creates a set of generated endpoints for custom data model entities. Most of the files generated by the REST Endpoint Generator are incomplete and require further configuration.

This topic discusses how to configure the glue and impl class files. They are the files highlighted in the following architecture diagram.



Configuration of the other files are discussed in other parts of the documentation.

- For an overview of the REST endpoint generator, see “The REST endpoint generator” on page 121.
- For more information on how to run the REST endpoint generator, see “Running the REST endpoint generator” on page 125.

- For more information on configuring the resource definition files, see “Configuring the resource definition files” on page 135.
- For more information on configuring authorization, see “Configuring authorization for generated endpoints” on page 155.

The glue and impl classes for generated endpoints

The REST endpoint generator creates and modifies a series of files that define logic for how the endpoints interact with the application.

- The **apiconfig file** is a "glue" file. It maps both the element resource and collection resource to a Gosu "Resource" file. For collection resources, this optionally provides a default sort order.
- There are two "impl" files that define implementation details.
  - The **<collection>Resource.gs file** is a Gosu file that defines required behaviors for working with collections. This includes behaviors such as how to retrieve the collection from the database and how to create a minimal child element.
  - The **<element>Resource.gs file** is a Gosu file that defines required behaviors for working with elements. This includes behaviors such as how to initialize a new element and how to delete one.

In most cases, the files and modifications made by the REST endpoint generator are not complete. Developers must provide additional code. The places where additional coding is needed are flagged with the following comment:

```
// TODO RestEndpointGenerator : <context>
```

You can search for these "TODO comments" in Studio and complete the configuration. The remainder of this topic explains the configuration needed in every glue and impl file.

Configuring the apiconfig file

Cloud API has multiple files whose name ends in `apiconfig.yaml`. The *shared_ext-1.0.apiconfig.yaml file* is the one most relevant to generated endpoints. This file maps both element resource and collection resources to a Gosu Resource file. The shared ext apiconfig file also defines any default sort order for collections.

The REST endpoint generator automatically modifies this shared ext apiconfig file to map element and collection resources to respective Gosu Resource files, but it does not add the default sort order. Guidewire recommends adding default sorts for all custom collection resources, as this provides a deterministic response payload.

Glue code added by the REST endpoint generator

the REST endpoint generator adds glue code for every element and collection, which maps the resource to its Gosu Resource file. For example if you generate endpoints for `CustomEntity_Ext`, the following is added to the shared apiconfig file:

```
CustomEntitiesExt:
  resource: gw.rest.ext.cc.claim.v1.claims.customentityext.CustomEntitiesExtResource
CustomEntityExt:
  resource: gw.rest.ext.cc.claim.v1.claims.customqentityext.CustomEntityExtResource
```

You can add default sort orders to each resource declaration. These are declared using a `defaultSort:` property under the `resource:` property.

Determining which fields can be used for sort order

To use a given field in a default sort order, the field must be declared a sortable in the schema definition.

For example, the following is a portion of the schema description for the `CustomEntity_Ext` resource.

```
"definitions":
  "CustomEntity_Ext": {
    "properties": {
```

```
      "customDescription": {
        ...
        "sortable": true
        },
      "customDueDate": {
        ...
        }
      },
```

Note that the `customDescription` field includes the `"sortable": true` expression, but the `customDueDate` field does not. This means that you can sort on `customDescription`, but not `customDueDate`.

Ascending order

To define ascending sort order, add `- <attributeName>` to the `defaultSort:` property.

For example, the following defines a default ascending sort order based on `customDescription`:

```
CustomEntitiesExt:
  resource: gw.rest.ext.cc.claim.v1.claims.customentityext.CustomEntitiesExtResource
  defaultSort:
    - customDescription
```

Descending order

To define descending sort order, add `– "-<attributeName>"` to the `defaultSort:` property.

For example, the following defines a default descending sort order based on `customDescription`:

```
CustomEntitiesExt:
  resource: gw.rest.ext.cc.claim.v1.claims.customentityext.CustomEntitiesExtResource
  defaultSort:
    - "-customDescription"
```

Multiple sort criteria

To define multiple sort criteria, list each attribute on separate lines. Members are sorted by the first criteria. Then, for any members with the same value for the first criteria, those members are sorted based on the second criteria, and so on.

For example, the following defines a default sort order based on `entryType` ascending and then by `typeField` descending:

```
CustomEntitiesExt:
  resource: gw.rest.ext.cc.claim.v1.claims.customentityext.CustomEntitiesExtResource
  defaultSort:
    - entryType
    - "-typeField"
```

Configuring the element resource file

The element resource file contains implementation code used to manipulate an element resource.

The name of this file is `<elementResource>Resource.gs`. For most APIs, the file is in the `gw.rest.ext.pc.<api>.<ancestor>.<resource>` package. For example, when generating endpoints for a `CustomEntity_Ext` data model entity in the Account API whose ancestor is Account:

- The element file is named `CustomEntityExtResource.gs`.
- It is in the `gw.rest.ext.pc.account.v1.accounts.customentityext` package.

However, if the endpoints have been added to the Job API, then the resource files are created in the policyperiod package. For example, when generating endpoints for a `CustomEntity_Ext` data model entity in the Job API whose ancestor is Activity:

- The element file is named `CustomEntityExtResource.gs`.
- It is in the `gw.rest.ext.pc.policyperiod.v1.activities.customentityext` package.

You must modify the following getters and methods in the class in the following ways.

The init method

The `init` method initializes a new instance of this resource. It also ensures that the element of this resource is a child of the element of its parent resource.

In most cases, the example code provided is sufficient. It requires only uncommenting the method code and defining the parent foreign key.

For example, suppose your child element uses `account` as its parent resource. The uncommented method code looks like this:

```
override function init(parent : CustomEntityExtResource, elementId : String) {
  super.init(parent, elementId)
  validateParentChildConsistency(this.Parent.Parent.Element, this.Element.Account)
}
```

If the endpoint is a root resource endpoint, the REST endpoint generator does not generate an `init` method. For more information, see “Configuring root resource endpoints” on page 174.

The delete method

The `delete` method deletes the resource, and it can provide an exception to the delete method from a business standpoint.

In most cases, the example code provided is sufficient. It requires only uncommenting the method code. For example, the uncommented method code looks like this:

```
override function delete() {
  this.Element.remove()
}
```

The CanEditException getter

The `CanEditException` getter determines whether the resource is editable from a business standpoint. If the resource is editable, the getter returns null. If the resource is uneditable, the getter returns a `CanEditException`.

For example, suppose the resource has an `ExpirationDate` field and edits can be made only before the expiration date.

- If today's date is on or before the `ExpirationDate`, the getter returns null.
- If today's date is after the `ExpirationDate`, the getter returns a `LocalizedExceptionUtil.operationNotCurrentlyAllowedException`

The following code illustrates this:

```
override property get CanEditException() : RestRequestException {
  var today = Calendar.getInstance().getTime();
  if (this.CustomEntity_Ext.ExpirationDate > today)
    return null
  else
    return LocalizedExceptionUtil.operationNotCurrentlyAllowedException
}
```

In most cases, you do not need constraints on editing the resource based on its business state, and the getter always returns null. For example:

```
override property get CanEditException() : RestRequestException  {
  return null
}
```

Note: The purpose of this getter is to allow or prevent edits based solely on the business state of the resource. It is not intended to control authorization. For more information on controlling authorization, see “Configuring authorization for generated endpoints”.

The CanDeleteException getter

The `CanDeleteException` getter determines whether the resource can be deleted from a business standpoint. If the resource can be deleted, the getter returns null. If the resource cannot be deleted, the getter returns a `CanEditException`.

For example, suppose the resource has an `ExpirationDate` field and can be deleted only before the expiration date.

- If today's date is on or before the `ExpirationDate`, the getter returns null.
- If today's date is after the `ExpirationDate`, the getter returns a `LocalizedExceptionUtil.operationNotCurrentlyAllowedException`.

The following code illustrates this:

```
override property get CanDeleteException() : RestRequestException {
  var today = Calendar.getInstance().getTime();
  if (this.CustomEntity_Ext.ExpirationDate > today)
    return null
  else
    return LocalizedExceptionUtil.operationNotCurrentlyAllowedException()
}
```

In most cases, you do not need constraints on viewing the resource based on its business state, and the getter always returns null. For example:

```
override property get CanViewException() : RestRequestException {
    return null
}
```

Note: The purpose of this getter is to allow or prevent edits based solely on the business state of the resource. It is not intended to control authorization. For more information, see “Configuring authorization for generated endpoints”.

The CanViewException getter

The `CanViewException` getter determines whether the resource is viewable from a business standpoint. If the resource is viewable, the getter returns null. If the resource is not viewable, the getter returns a `CanViewException`.

For example, suppose the resource has an `ExpirationDate` field and can be viewed only before the expiration date.

- If today's date is on or before the `ExpirationDate`, the getter returns null.
- If today's date is after the `ExpirationDate`, the getter returns a `LocalizedExceptionUtil.operationNotCurrentlyAllowedException`.

The following code illustrates this:

```
override property get CanViewException() : RestRequestException {

  var today = Calendar.getInstance().getTime();
  if (this.CustomEntity_Ext.ExpirationDate > today)
    return null
  else
    return LocalizedExceptionUtil.operationNotCurrentlyAllowedException()
}
```

In most cases, you do not need constraints on viewing the resource based on its business state, and the getter always returns null. For example:

```
override property get CanViewException() : RestRequestException {
    return null
}
```

Note:

The purpose of this getter is to allow or prevent views based solely on the business state of the resource. It is not intended to control authorization. For more information, see “Configuring authorization for generated endpoints” on page 155.

Configuring the collection resource file

The collection resource file contains implementation code that manipulates the collection resource.

The name of this file is `<collectionResource>Resource.gs`. The file is located in the `gw.rest.ext.pc.<api>.<ancestor>.<resource>` package. For example, when generating endpoints for a `CustomEntities_Ext` data model entity in the Account API whose ancestor is Account:

- The element file is named `CustomEntitiesExtResource.gs`.
- It is in the `gw.rest.ext.pc.account.v1.accounts.customentityext` package.

You must modify the following getters and methods in the class in the following ways.

> **Note:** The authorization layer is responsible for determining which resources the caller can access based on who the caller is. The following methods and getters are not intended to control authorization. For more information on controlling authorization, see “Configuring authorization for generated endpoints” on page 155.

The loadValues method (stream-backed collections only)

The `loadValues` method converts the collection resource array into a stream object. It also populates the response payload with the stream-backed collection.

For example, suppose the parent of `CustomEntitities_Ext` has an array of `CustomEntitities_Ext`. This method returns that array converted into a stream. The following code illustrates this:

```
protected override function loadValues() : Stream<Object> {
  return this.Parent.Element.getCustomEntities_Ext().stream()
```

If the parent does not have an array of the custom entity, then you must write Gosu code to construct an array or list manually and then convert it into a stream.

The buildBaseQuery method (query-backed collections only)

The `buildBaseQuery` method creates a Gosu query that returns the collection resource related to the parent resource.

For example, suppose the parent of `CustomEntity_Ext` is Account. This method needs to return all `CustomEntity_Ext` instance associated with the relevant account. following code illustrates this:

```
protected override function buildBaseQuery() : IQueryBeanResult<CustomEntity_Ext> {
  return Query.make(entity.CustomEntity_Ext)
      .compare(CustomEntity_Ext#Account, Relop.Equals, this.Parent.Element)
      .select()
```

The canViewException getter

The `canViewException` getter determines whether the collection resource is viewable from a business standpoint. If the collection resource is viewable, the getter returns null. If the collection resource is not viewable, the getter returns a `CanViewException`.

In most cases, you do not need constraints on viewing the resource based on its business state. In this case, the getter always returns null. For example:

```
override property get CanViewException() : RestRequestException {
    return null
```

If the endpoint is a root resource endpoint, the REST endpoint generator does not generate a `canViewException` getter. For more information, see “Configuring root resource endpoints” on page 174.

The canCreateException getter

The `canCreateException` getter determines whether the collection resource can be created from a business standpoint. If the collection resource is createable, the getter returns null. If the collection resource is not createable, the getter returns a `CanCreateException`.

If there are no constraints on creating the resource based on any business state, then the getter always returns null.

If the endpoint is a root resource endpoint, the REST endpoint generator does not generate a `canCreateException` getter. For more information, see “Configuring root resource endpoints” on page 174.

The createMinimalChildElement method

The `createMinimalChildElement` method is used for POSTs on the collection resource. It creates a new instance of the entity and attaches it to its parent.

In most cases, the example code as shown below is sufficient. However, if there is additional initialization logic, it can be defined here.

```
override function createMinimalChildElement(attributes : DataAttributes) : CustomEntity_Ext {
  var customEntity_Ext = new CustomEntity_Ext()
  customEntity_Ext.Account = this.Parent.Element
  return customEntity_Ext
```

Completing the configuration

In addition to the glue and impl files, you must also configure the resource definition files and the authorization files.

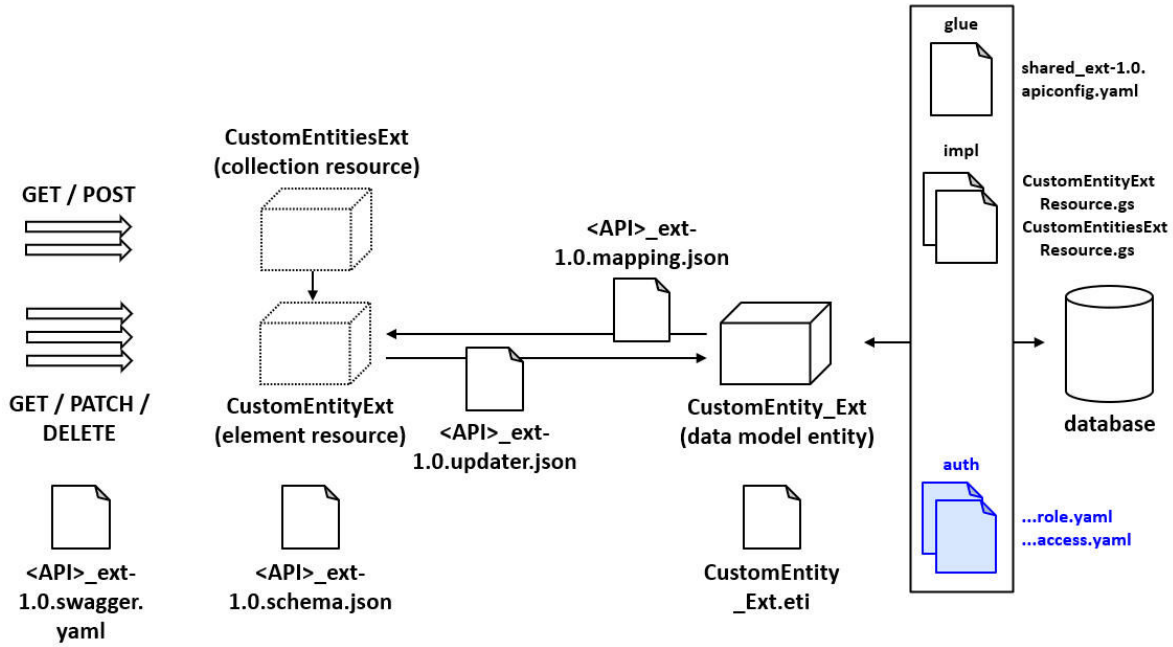
- For more information on configuring the resource definition files, see “Configuring the resource definition files”.
- For more information on configuring authorization, see “Configuring authorization for generated endpoints” on page 155.

chapter 29

Configuring authorization for generated endpoints

The REST endpoint generator is a tool that creates a set of generated endpoints for custom data model entities. Most of the files generated by the REST Endpoint Generator are incomplete and require further configuration.

This topic discusses how to configure the authorization files. They are the files highlighted in the following architecture diagram.



Configuration of the other files are discussed in other parts of the documentation.

- For an overview of the REST endpoint generator, see “The REST endpoint generator” on page 121.
- For more information on how to run the Rest endpoint generator, see “Running the REST endpoint generator” on page 125.

- For more information on configuring the resource definition files, see “Configuring the resource definition files” on page 135.
- For more information on configuring the glue and impl classes, see “Configuring glue and impl classes for generated endpoints” on page 147.

Configuring endpoint access for generated endpoints

Endpoint access defines the aspects of an endpoint's behaviors that are available to a caller. This includes:

- What endpoints are available to the caller?
- What operations can a caller call on an available endpoint?
- What fields can the caller specify in a request payload or get in a response payload?

For more information on the general behavior of endpoint access, see “Endpoint access” on page 373.

When running the REST Endpoint Generator, several prompts pertain to endpoint access. This includes:

- Which roles can access the GET collection and GET element endpoint?
- Which roles can access the POST collection endpoint?
- Which roles can access the PATCH element endpoint?
- Which roles can access the DELETE element endpoint?

Based on the responses, the REST endpoint generator adds code to the corresponding role.yaml files. Places where additional coding may be needed are flagged with the following comment:

```
TODO RestEndpointGenerator
```

You can search for these "TODO RestEndpointGenerator" comments in Studio to complete the configuration.

Code generated in role.yaml files

Providing access to all operations

If you specify that a given role has GET, POST, PATCH, and DELETE access to a generated endpoint, the REST endpoint generator adds the following code to the associated role.yaml file:

```
- endpoint: /<path>/<customEndpointCollection>
  # TODO RestEndpointGenerator : Adjust the permissions appropriately
  methods:
  - GET
  - POST
- endpoint: "<path>/<customEndpointCollection>/*"
  # TODO RestEndpointGenerator : Adjust the permissions appropriately
  methods:
  - GET
  - DELETE
  - PATCH
```

The first block grants the ability to retrieve a collection and the ability to create a resource.

The second block grants the ability to retrieve an element, to modify an element, and to delete an element. (A single * wildcard indicates access is provided for anything one level below the current endpoint level.)

For example, suppose you generated endpoints for the `CustomEntity_Ext` data model entity and added GET, POST, PATCH, and DELETE access to these endpoints to the Manager role. The REST endpoint generator would add the following lines to the `Manager.role.yaml` file:

```
- endpoint: /account/v1/accounts/custom-entities-ext
  # TODO RestEndpointGenerator : Adjust the permissions appropriately
  methods:
  - GET
  - POST
- endpoint: "/account/v1/accounts/custom-entities-ext/*"
```

```
  # TODO RestEndpointGenerator : Adjust the permissions appropriately
  methods:
  - GET
  - DELETE
  - PATCH
```

The first block grants the ability to retrieve a collection of `CustomEntities_Ext` and the ability to create a `CustomEntity_Ext`.

The second block grants the ability to retrieve a specific `CustomEntity_Ext`, to modify a `CustomEntity_Ext`, and to delete a `CustomEntity_Ext`.

Limiting access to only some operations

During the REST endpoint generator prompts:

- If you specified GET access for a given role, but not POST access, the POST operation is omitted from the first block.
- If you specified GET access for a given role, but not PATCH (or DELETE) access, the PATCH (or DELETE) operation is omitted from the second block.
- If you did not specify GET access for a given role, then both blocks are omitted. (In order to POST, PATCH, or DELETE, the role must also have the ability to GET.)

Configuring code in role.yaml files

If you specified roles in response to the REST endpoint generator prompts, then the endpoint access code generated by the REST endpoint generator is likely to be usable as is. However, there may be cases where additional configuration is required. For example:

- Guidewire recommends removing the `TODO RestEndpointGenerator` comments.
- An insurer may want to add additional field-level access in the `accessibleFields` section.

For more information on how to configure `role.yaml` files, see “Endpoint access” on page 373.

Configuring resource access for generated endpoints

Resource access defines, for a given type of resource, which instances of that resource type the caller can access. For example, for a given caller, endpoint access might grant access to a GET `/policies` endpoint. But this does not necessarily mean the caller can access every policy in the database. Resource access can limit which specific policies that caller can view.

For more information on the general behavior of resource access, see “Resource access” on page 385.

When running the REST endpoint generator, there are no prompts that pertain to resource access. The REST endpoint generator generates resource access code in the same way for every set of generated endpoints. At the very least, the code requires review. In some cases, the code may require additional configuration. The places where coding requires review and additional configuration is flagged with the following comment:

```
TODO RestEndpointGenerator
```

You can search for these "TODO RestEndpointGenerator" comments in Studio to complete the configuration.

Code generated in access.yaml files

Every resource access strategy is defined in multiple files. In the base configuration, all of the files for a given resource access strategy start with the same name.

Broadly speaking, there are two types of access files: core and extension.

A *core access file* is an access file that defines one or more base configuration resource access strategies. Core access files either have `core` in the name, or are located in a package with `core` in the path.

WARNING: Do not modify any core access files. Modifying these files can result in certain sets of callers being unable to execute API calls.

An *extension access file* is an access file that provides a location for extensions to base configuration resource access strategy behavior. Extension access files either have `ext` in the name, or are located in a package with `ext` in the path.

Code generated by the REST endpoint generator is placed in extension access files. The specific code added to each file depends on the associated type of caller.

Generated resource access code for internal users

The extension access file for internal users is named `internal_ext-1.0.access.yaml`.

The REST endpoint generator assumes that an internal user can access a given instance of the custom resource if the internal user can also access the resource's parent. Therefore, the filters for all operations are set to `__inherit`.

Stream-based collections

If you generate endpoints for a `CustomEntity_Ext` entity, and the entity is backed by a stream, the following is added to the internal user resource access extension file:

```
resources:
  CustomEntitiesExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: __inherit
      create: __inherit
  CustomEntityExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: __inherit
      edit: __inherit
      delete: __inherit
```

Query-based collections

If you generate endpoints for a `CustomEntity_Ext` entity, and the entity is backed by a query, the following is added to the internal user resource access extension file:

```
resources:
  CustomEntitiesExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: __inherit
      create: __inherit
    filter: __noFilter
  CustomEntityExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: __inherit
      edit: __inherit
      delete: __inherit
```

Note that, for query-backed collections, there is an additional `filter: __noFilter` line of code in the collections section. This line is needed to determine appropriate view permissions for query-backed collections.

- This line of code does not exist for stream-backed collections because stream-backed collections load the entire collection contents into memory at one time. Cloud API can simply iterate over the entire collection using the specified `view` permission to determine which items in the stream can be viewed.
- This line of code does exist for query-backed collections because query-backed collections do not load the entire collection into memory at one time. Instead, the collection is loaded one page at a time. Additional logic is required to identify the items in the collection that can be viewed. This is specified by the additional filter.

In most business circumstances, the filter can be set to `__noFilter`, which specifies no additional filter logic is needed. This is acceptable because, in most cases, a child object can be viewed if the parent object can be viewed.

Generated resource access code for external users

The extension access file for external users is named `accountholder_ext-1.0.access.yaml`.

The REST endpoint generator assumes that an external user can access a given instance of the custom resource if the external user can also access the resource's parent. Therefore, the filters for all operations are set to `__inherit`.

Stream-based collections

If you generate endpoints for a `CustomEntity_Ext` entity, and the entity is backed by a stream, the following is added to the external user resource access extension file:

```
resources:
  CustomEntitiesExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: "__inherit"
      create: "__inherit"
  CustomEntityExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: "__inherit"
      edit: "__inherit"
      delete: "__inherit"
```

Query-based collections

If you generate endpoints for a `CustomEntity_Ext` entity, and the entity is backed by a query, the following is added to the internal user resource access extension file:

```
resources:
  CustomEntitiesExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: "__inherit"
      create: "__inherit"
    filter: "__inherit"
  CustomEntityExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: "__inherit"
      edit: "__inherit"
      delete: "__inherit"
```

Note that, for query-backed collections, there is an additional `filter: __inherit` line of code in the collections section. This line is needed to determine appropriate view permissions for query-backed collections.

- This line of code does not exist for stream-backed collections because stream-backed collections load the entire collection contents into memory at one time. Cloud API can simply iterate over the entire collection using the specified `view` permission to determine which items in the stream can be viewed.
- This line of code does exist for query-backed collections because query-backed collections do not load the entire collection into memory at one time. Instead, the collection is loaded one page at a time. Additional logic is required to identify the items in the collection that can be viewed. This is specified by the additional filter.

Generated resource access code for services

The extension access file for internal users is named `service_ext-1.0.access.yaml`.

The REST Endpoint Generator assumes that services are not restricted by resource access. Therefore, the filters for all operations are set to `true`.

Stream-based collections

If you generate endpoints for a `CustomEntity_Ext` entity, and the entity is backed by a stream, the following is added to the service resource access extension file:

```
resources:
  CustomEntitiesExt:
```

```
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: true
      create: true
  CustomEntityExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: true
      edit: true
      delete: true
```

Query-based collections

If you generate endpoints for a `CustomEntity_Ext` entity, and the entity is backed by a query, the following is added to the internal user resource access extension file:

```
resources:
  CustomEntitiesExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: true
      create: true
    filter: __noFilter
  CustomEntityExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: true
      edit: true
      delete: true
```

Note that, for query-backed collections, there is an additional `filter: __noFilter` line of code in the collections section. This line is needed to determine appropriate view permissions for query-backed collections.

- This line of code does not exist for stream-backed collections because stream-backed collections load the entire collection contents into memory at one time. Cloud API can simply iterate over the entire collection using the specified `view` permission to determine which items in the stream can be viewed.
- This line of code does exist for query-backed collections because query-backed collections do not load the entire collection into memory at one time. Instead, the collection is loaded one page at a time. Additional logic is required to identify the items in the collection that can be viewed. This is specified by the additional filter.

If services are not restricted by resource access, then no additional filter logic is needed. The `filter: __noFilter` line of code specifies this.

Generated resource access code for special use cases

There are two resource access strategies used for special cases.

- Unauthenticated user resource access is used for unauthenticated users. This access is defined in `unauthenticated_ext-1.0.access.yaml`.
- Default resource access is used for callers who have been authenticated but specify no resource access strategy with the call. This access is defined in `default_ext-1.0.access.yaml`.

The REST endpoint generator assumes that no resource access is provided to unauthenticated or default users. Therefore, the filters for all operations are set to false.

Stream-based collections

If you generate endpoints for a `CustomEntity_Ext` entity, and the entity is backed by a stream, the following is added to the unauthorized access extension file:

```
resources:
  CustomEntitiesExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: "false"
      create: "false"
  CustomEntityExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: "false"
```

```
      edit: "false"
      delete: "false"
```

Query-based collections

If you generate endpoints for a `CustomEntity_Ext` entity, and the entity is backed by a query, the following is added to the unauthorized access extension file:

```
resources:
  CustomEntitiesExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: "false"
      create: "false"
    filter: __noFilter
  CustomEntityExt:
    # TODO RestEndpointGenerator : Update the default generated access here
    permissions:
      view: "false"
      edit: "false"
      delete: "false"
```

Note that, as is the case for the other caller types, query-backed collections have an additional "`filter:`" line of code in the collections section.

Configuring generated resource access code

Depending on the business needed, the following resource access configuration may be required:

- Guidewire recommends removing the `TODO RestEndpointGenerator` comments.
- The default resource access for internal users is "you can access the custom entity if you can access its parent." If this is not appropriate or sufficient, you must configure the internal resource strategy extension file.
- The default resource access for services is "you can access all custom entities." If this is not appropriate or sufficient, you must configure the service resource strategy extension file.
- There is no default resource access for external users, unauthenticated users, or callers with default access. If your endpoints support any of these types of callers, you must configure the corresponding resource strategy extension file.

For more information on how to configure resource access files, see “Resource access” on page 385.

Completing the configuration

Once you have run the REST endpoint generator and configured the resource definition files, the glue and impl Gosu files, and the authorization files, the configuration of the new CRUD endpoints is complete.

chapter 30

Additional considerations for generated endpoints

This topic covers special use cases of the REST endpoint generator and consideration for those use cases.

Integration graphs

An *integration graph* is a special base configuration schema and mapper that certain base configuration outbound integrations use to send information about a parent object and its children objects. Integration graphs are not used by Cloud API, but the files that define them are a part of Cloud API.

Cloud API has the following integration graphs:

- The ClaimCenter `claim_graph`, whose parent is Claim
- The PolicyCenter `policyperiod_graph`, whose parent is PolicyPeriod
- The ContactManager `contact_graph`, whose parent is Contact

There are several outbound integrations that make use of integration graphs. For example, integration graphs reflect the structure of the event payload that App Events generates for downstream systems.

Guidewire recommends against insurers configuring integration graphs directly. However, when you generate endpoints for a custom entity, you can also add the custom resource to an integration graph if the resource has a parent resource and that parent resource belongs to the integration graph. To add the custom resource to the graph, answer the final prompt ("Should <CustomEntity> be added to an integration graph?") with "y". There may be an additional prompt asking for the name of the graph.

- If the entity is effective-dated, the information is added to the policy period graph.
- If the entity is not effective-dated, an additional prompt asks for the name of the graph to add the entity to.

When a custom entity is added to an integration graph, the REST endpoint generator modifies the following files:

- The `<graphName>_graph_ext-1.0.schema.json` file
- The `<graphName>_graph_ext-1.0.mapping.json` file
- The `shared_ext-1.0.apiconfig.yaml` file

You do not need to configure the graph schema file. But, configurations are required in the graph mapper file and the `apiconfig.yaml` file.

Note: Adding a custom entity or custom resource to the integration graph does not ensure that the event payload that App Events generates includes this data. For more information, see *App Events*.

The graph schema file

The graph schema file defines the structure of the integration graph. When you specify that you want to add a custom resource to a given graph, the REST endpoint generator adds code to the corresponding `<graphName>_graph_ext-1.0.schema.json` file.

For example, suppose you generated endpoints for the `CustomEntity_Ext` data model entity and added the corresponding resource to the PolicyPeriod graph. The `policyperiod_graph_ext-1.0.schema.json` file has the following code added (shown in bold):

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "x-gw-combine": [
    "ext.account.v1.account_ext-1.0",
    "ext.common.v1.common_ext-1.0",
    "ext.policyperiod.v1.policyperiod_ext-1.0",
    "gw.content.pc.policyperiod.v1.policyperiod_graph_content-1.0"
  ],
  "definitions": {
    "PolicyPeriod": {
      "properties": {
        "customEntitiesExt": {
          "title": "Custom entities ext",
          "description": "The collection of custom entities ext on this policy period",
          "type": "array",
          "items": {
            "$ref": "#/definitions/CustomEntityExt"
          }
        }
      }
    }
  }
}
```

Typically, the graph schema file does not require configuration.

The graph mapper file

The graph mapper file defines how information is written to the properties in an integration graph. When you specify that you want to add a custom resource to a given graph, the REST endpoint generator adds code to the corresponding `<graphName>_graph_ext-1.0.mapping.json` file.

For example, suppose you generated endpoints for the `CustomEntity_Ext` data model entity and added the corresponding resource to the PolicyPeriod graph. The `policyperiod_graph_ext-1.0.mapping.json` file has the following code added (shown in bold):

```
{
  "schemaName": "ext.policyperiod.v1.policyperiod_graph_ext-1.0",
  "combine": [
    "ext.account.v1.account_ext-1.0",
    "ext.policyperiod.v1.policyperiod_ext-1.0",
    "gw.content.pc.policyperiod.v1.policyperiod_graph_content-1.0"
  ],
  "mappers": {
    "PolicyPeriod": {
      "schemaDefinition": "PolicyPeriod",
      "root": "entity.PolicyPeriod",
      "properties": {
        "customEntitiesExt": {
          "path": "TODO RestEndpointGenerator provide a collection
                   of CustomEntities_Ext",
          "mapper": "#/mappers/CustomEntityExt"
        }
      }
    }
  }
}
```

You must configure the file by specifying the `path` for custom entity collection properties. This is typically set to the corresponding array on the parent data model entity.

Following on from the previous example, the path attribute in `policyperiod_graph_ext-1.0.mapping.json` file would need to be set as follows:

```
{
  "schemaName": "ext.policyperiod.v1.policyperiod_graph_ext-1.0",
  "combine": [
    "ext.account.v1.account_ext-1.0",
    "ext.policyperiod.v1.policyperiod_ext-1.0",
    "gw.content.pc.policyperiod.v1.policyperiod_graph_content-1.0"
  ],
  "mappers": {
    "PolicyPeriod": {
      "schemaDefinition": "PolicyPeriod",
      "root": "entity.PolicyPeriod",
      "properties": {
        "customEntitiesExt": {
          "path": "PolicyPeriod.CustomEntities_Ext",
          "mapper": "#/mappers/CustomEntityExt"
        }
      }
    }
  }
}
```

Mapping in the apiconfig file

The `shared_ext-1.0.apiconfig.yaml` maps shared element resource and collection resources to Gosu Resource files. It also specifies URI mappings for integration graphs. When you specify that you want to add a custom resource to a graph, the REST endpoint generator adds code to the corresponding `shared_ext-1.0.apiconfig.yaml` file.

For example, suppose you generated endpoints for the `CustomEntity_Ext` data model entity and added the corresponding resource to the PolicyPeriod graph. The `shared_ext-1.0.apiconfig.yaml` file has the following code added (shown in bold):

```
entityURIMappings:
 CustomEntity_Ext:
    uri: "${parentUri}/custom-entities-ext/${CustomEntity_Ext.RestId}"
    parent: "TODO RestEndpointGenerator link to the parent of CustomEntity_Ext"
```

You must configure the file by specifying the `parent` for custom entities. This is typically set to the corresponding foreign key on the custom data model entity.

Following on from the previous example, the path attribute in the `shared_ext-1.0.apiconfig.yaml` file would need to be set as follows:

```
entityURIMappings:
  CustomEntity_Ext:
    uri: "${parentUri}/custom-entities-ext/${CustomEntity_Ext.RestId}"
    parent: "CustomEntity_Ext.PolicyPeriod"
```

Marking graph properties as eventSafe

The REST endpoint generator also checks if the root entity for the endpoint generates events. If it does, it sets the `eventSafe` property on the graph mapping property appropriately.

Base configuration entities

You can generate endpoints for base configuration entities that do not have endpoints. For example, suppose that the base configuration of PolicyCenter had a data model entity named `TextMessage` that was used to store text messages sent to an associated contact. Also, suppose that there were no endpoints for this entity in Cloud API. In this case, you could use the REST endpoint generator to generate endpoints for the `TextMessage` entity.

The REST endpoint generator prompts

The prompts associated with generating endpoints for base configuration entities are the same as the prompts for a custom entity.

The generated resource name

The REST endpoint generator automatically adds an "Ext" suffix to the resource name. For example, if you generate endpoints for the theoretical `TextMessage` entity, the resource name would be `TextMessageExt`. This suffix is added to prevent conflicts that could arise in the future if Guidewire provides base configuration endpoints for the entity.

> **Note:** When the REST endpoint generator prompts you for the entity name, provide the entity name as it appears in the Data Dictionary. Do not add the "Ext" suffix. The suffix becomes a part of the resource name, but the prompt is asking for the data model entity name.

The generated endpoint paths

As is the case with custom entities, the paths of generated endpoints are based on the resource name.

- The path contains an initial lower-case letter.
- Underscores are replaced by hyphens.
- In general, capital letters are converted to lower case letters with a hyphen in front of them.

The REST endpoint generator adds an "Ext" to the resource name, and therefore the paths contain `-ext` and `Ext`. For example, if you generated endpoints for the theoretical `TextMessage` entity, the endpoints would be:

- GET `/text-messages-ext`
- POST `/text-messages-ext`
- GET `/text-messages-ext/{textMessageExtId}`
- PATCH `/text-messages-ext/{textMessageExtId}`
- DELETE `/text-messages-ext/{textMessageExtId}`

What if some future release adds endpoints for the base configuration entity?

Suppose you generate endpoints for a base configuration entity that has no endpoints, such as the theoretical `TextMessage` entity. Then, you upgrade to a release in which Guidewire does provide endpoints for that entity. In this case, there would be two resources for the `TextMessage` entity:

- A `TextMessageExt` resource created by the REST endpoint generator that is used by the generated endpoints.
- A `TextMessage` resource created by Guidewire that is used by the base configuration endpoints.

There would also be two sets of endpoints:

- The paths for the generated set would use an "-ext"/"Ext" suffix, such as:
  - GET `/text-messages-ext`
  - POST `/text-messages-ext`
  - GET `/text-messages-ext/{textMessageExtId}`
  - PATCH `/text-messages-ext/{textMessageExtId}`
  - DELETE `/text-messages-ext/{textMessageExtId}`
- The paths for the Guidewire set would not use an "-ext"/"Ext" suffix, such as:
  - GET `/text-messages`
  - POST `/text-messages`
  - GET `/text-messages/{textMessageId}`
  - PATCH `/text-messages/{textMessageId}`
  - DELETE `/text-messages/{textMessageId}`

Thus, in this future release, you would be able to continue using the endpoints you generated with the functionality you configured. You could also use the endpoints that Guidewire generated.

Endpoints may correspond to unsupported operations

The REST endpoint generator always generates POST, PATCH, and DELETE endpoints for the target entity. However, some base configuration entities may not support all of these operations. For example, the theoretical `TextMessage` entity may not support DELETEs. In these situations, the endpoints for the unsupported operations will not work.

When an endpoint corresponds to an unsupported operation, you may want to consider removing it from the `<api>_ext-1.0.swagger.yaml` file to prevent caller applications from attempting to use it.

Base configuration subtype entities

In some cases, a supertype entity may have some subtypes which have base configuration endpoints and other subtypes which do not. In this case, you can generate endpoints for any subtype that does not have base configuration endpoints, even if the supertype or some of the other subtypes do have generated endpoints.

Prohibited entities

You cannot generate endpoints for base configuration entities that already have endpoints. This is true for both entities with endpoints created by Guidewire and entities with endpoints generated by you.

Some InsuranceSuite applications also have base configuration entities that do not have endpoints and that you cannot generate endpoints for. If you name one of these entities at the first prompt, the REST endpoint generator responds with an error message similar to the following:

```
You cannot generate endpoints for the '<namedEntity>' entity. The REST endpoint generator
does not allow endpoint generation for the following entities and their subtypes
[<list_of_prohibited_entitied>]
```

Supertype entities

A supertype entity is a data model entity which has one or more other entities that act as subtypes. In a supertype/subtype entity structure:

- The top-level entity is referred to as the *parent entity* or the *supertype entity*.
- Each lower-level entity is referred to as a *child entity* or a *subtype entity*.

For example, suppose there is an `Interaction_Ext` entity that captures information about an interaction (a phone call, email, or in-person conversation) that the insurer has with someone else. The `Interaction_Ext` entity has two subtypes: `InteractionWithInsured_Ext` and `InteractionWithVendor_Ext`.

- The `Interaction_Ext` supertype has an `InteractionDate` datetime field.
- `InteractionWithInsured_Ext` has an `isComplaint` Boolean field.
- `InteractionWithVendor_Ext` has an `isBillable` Boolean field.

When you use the REST endpoint generator to generate endpoints for a supertype entity, the following additional prompt is asked:

```
The entity '<CustomEntity>' is a concrete type that has subtypes. Do you wish for subtypes
to (sh)are this endpoint or have their own (se)parate endpoints? sh/se
```

You can choose either shared handling or separate handling. With *shared handling*, the REST endpoint generator CRUD endpoints that are intended to work with information at both the supertype and subtype level simultaneously. When you choose *separate handling*, it creates CRUD endpoints that are intended to work with information at the supertype level only, and if you want to work with information at the subtype level, you must generate an additional set of endpoints for each subtype.

Shared handling

Shared handling is designed for situations where the bulk of the information is at the supertype level and you want to manage the information with a single set of endpoints.

In this approach, the tool generates a single element resource and a single set of CRUD endpoints. After configuration, the element resource would typically include fields declared at the supertype level and at each subtype level. A single GET can potentially return objects of different subtypes.

The subtype field

The REST endpoint generator automatically adds a `subtype` field. This field tracks the subtype of a given object. The field must be specified when an object is created, as it is used to determine how to create the object. Once created, an object's `subtype` value cannot be changed.

Data mapping requirements

The only properties that are automatically added to the resource are the `id` and `subtype` properties. All other properties, from both the supertype and subtype levels, must be configured manually.

In the schema file, Guidewire recommends configuring subtype fields so that they can be populated only when the object is of that subtype. This prevents invalid data from being added to the database.

The syntax for specifying these behaviors on subtype fields is as follows:

```
"definitions": {
  "<supertype>": {
    "properties": {
      "<subtypeFieldName>": {
        "x-gw-nullable": true,
        "x-gw-extensions": {
          "entitySubtype": "<subtypeThatOwnsThisField>"
        }
      },
```

In the mapping file, the mapper for a subtype field must cast the supertype entity as the specific subtype entity. It must also include a `predicate` attribute that specifies the correct subtype.

The syntax for specifying these behaviors is as follows:

```
"mappers": {
  "<supertype>": {
    "properties": {
      "<subtypeFieldName>": {
        "path": "(<Supertype> as <Subtype>).<subtypeFieldName>",
        "predicate": "<Supertype> typeis <Subtype>"
      },
```

In the updater file, the updater for a subtype field must cast the supertype entity as the specific subtype entity.

The syntax for specifying these behaviors is as follows:

```
"updaters": {
  "<supertype>": {
    "properties": {
      "<subtypeFieldName>": {
        "path": "(<Supertype> as <Subtype>).<subtypeFieldName>"
      },
```

Shared handling example

For example, suppose there is an `Interaction_Ext` entity that captures information about an interaction (a phone call, email, or in-person conversation) that the insurer has with someone else. The `Interaction_Ext` entity has two subtypes: `InteractionWithInsured_Ext` and `InteractionWithVendor_Ext`.

- The `Interaction_Ext` supertype has an `InteractionDate` datetime field.
- `InteractionWithInsured_Ext` has an `isComplaint` Boolean field.
- `InteractionWithVendor_Ext` has an `isBillable` Boolean field.

The REST endpoint generator creates CRUD endpoints for the `Interaction_Ext` entity as root resource endpoints using shared subtype handling.

The following endpoints are generated:

- GET `/interaction-ext`
- POST `/interaction-ext`
- GET `/interaction-ext/{interactionExtId}`
- PATCH `/interaction-ext/{interactionExtId}`
- DELETE `/interaction-ext/{interactionExtId}`

After configuration, the schema definition for `Interaction_Ext` would include the following:

```
"definitions": {
  "Interaction_Ext": {
    "properties": {
      "id": {
        "title": "ID",
        "description": "The unique identifier of this element",
        "type": "string",
        "readOnly": true
      },
      "subtype": {
        "title": "Subtype",
        "description": "The specific type of...",
        "type": "string",
        "x-gw-type": "typekey.Interaction_Ext",
        "x-gw-extensions": {
          "createOnly": true,
          "filterable": true,
          "requiredForCreate": true,
          "sortable": true
        }
      },
      "interactionDate": {
        "type": "string",
        "format": "date-time"
      },
      "isComplaint": {
        "type": "boolean",
        "x-gw-nullable": true,
        "x-gw-extensions": {
          "entitySubtype": "InteractionWithInsured_Ext"
        }
      },
      "isBillable": {
        "type": "boolean",
        "x-gw-nullable": true,
        "x-gw-extensions": {
          "entitySubtype": "InteractionWithVendor_Ext"
        }
      },
      ...
```

After configuration, the mapping definition for `Interaction_Ext` would include the following:

```
"mappers": {
  "Interaction_Ext": {
    "properties": {
      "id": {
        "path": "Interaction_Ext.RestId"
      },
      "subtype": {
        "path": "Interaction_Ext.Subtype"
      },
      "interactionDate": {
        "path": "Interaction_Ext.InteractionDate"
      },
      "isComplaint": {
        "path": "(Interaction_Ext as InteractionWithInsured_Ext).IsComplaint",
        "predicate": "Interaction_Ext typeis InteractionWithInsured_Ext"
        }
      },
      "isBillable": {
        "path": "(Interaction_Ext as InteractionWithVendor_Ext).IsBillable",
        "predicate": "Interaction_Ext typeis InteractionWithVendor_Ext"
        }
      },
      ...
```

After configuration, the updater definition for `Interaction_Ext` would include the following:

```
"updaters": {
  "Interaction_Ext": {
    "properties": {
```

```
      "subtype": {
        "path": "Interaction_Ext.Subtype"
      },
      "interactionDate": {
        "path": "Interaction_Ext.InteractionDate"
      },
      "isComplaint": {
        "path": "(Interaction_Ext as InteractionWithInsured_Ext).IsComplaint"
        }
      },
      "isBillable": {
        "path": "(Interaction_Ext as InteractionWithVendor_Ext).IsBillable"
        }
      },
      ...
```

Separate handling

Separate handling is designed for situations where there is so much information at the subtype level that you want to manage each subtype as a distinct entity.

In this approach, you must run the REST endpoint generator for the supertype and each subtype. This means that you will have a resource and set of CRUD endpoints for the supertype and an additional resource and set of CRUD endpoints for each subtype. It also means that the supertype endpoints can be used to interact with objects declared at the supertype level, and only objects declared at the supertype level. The supertype endpoints will not interact with objects declared at the subtype level.

When you run the REST endpoint generator for the supertype and choose separate handling, there is no `subtype` field automatically added to the resource.

Data mapping requirements

With separate handling, each entity in the data model is treated as a distinct entity. Therefore, there are no special requirements around how to specify schema definitions and mappers.

Keep in mind that if there are any supertype fields that you want to have access to when using the subtype endpoints, you must configure these fields in each subtype resource.

Separate handling example

For example, suppose there is an `Interaction_Ext` entity that captures information about an interaction (a phone call, email, or in-person conversation) that the insurer has with someone else. The `Interaction_Ext` entity has two subtypes: `InteractionWithInsured_Ext` and `InteractionWithVendor_Ext`.

- The `Interaction_Ext` supertype has an `InteractionDate` datetime field.
- `InteractionWithInsured_Ext` has an `isComplaint` Boolean field.
- `InteractionWithVendor_Ext` has an `isBillable` Boolean field.

The REST endpoint generator creates the following. In each case, the custom entity is the root resource for the endpoint:

- CRUD endpoints for the `Interaction_Ext` entity using separate subtype handling.
- CRUD endpoints for the `InteractionWithInsured_Ext` entity
- CRUD endpoints for the `InteractionWithVendor_Ext` entity

The following endpoints are generated:

- Endpoints for working with `Interaction_Ext` entities
  - GET `/interaction-ext`
  - POST `/interaction-ext`
  - GET `/interaction-ext/{interactionExtId}`
  - PATCH `/interaction-ext/{interactionExtId}`

    - DELETE `/interaction-ext/{interactionExtId}`
- Endpoints for working with `InteractionWithInsured_Ext` entities
    - GET `/interaction-with-insured-ext`
    - POST `/interaction-with-insured-ext`
    - GET `/interaction-with-insured-ext/{interactionWithInsuredExtId}`
    - PATCH `/interaction-with-insured-ext/{interactionWithInsuredExtId}`
    - DELETE `/interaction-with-insured-ext/{interactionWithInsuredExtId}`
- Endpoints for working with `InteractionWithVendor_Ext` entities
    - GET `/interaction-with-vendor-ext`
    - POST `/interaction-with-vendor-ext`
    - GET `/interaction-with-vendor-ext/{interactionWithVendorExtId}`
    - PATCH `/interaction-with-vendor-ext/{interactionWithVendorExtId}`
    - DELETE `/interaction-with-vendor-ext/{interactionWithVendorExtId}`

Schema definitions

After configuration, the schema definition for `Interaction_Ext` would include the following:

```
"definitions": {
  "Interaction_Ext": {
    "properties": {
      "id": {
        "title": "ID",
        "description": "The unique identifier of this element",
        "type": "string",
        "readOnly": true
      },
      "interactionDate": {
        "type": "string",
        "format": "date-time"
      },
      ...
```

The schema definition for `InteractionWithInsured_Ext` would include the following:

```
"definitions": {
  "InteractionWithInsured_Ext": {
    "properties": {
      "id": {
        "title": "ID",
        "description": "The unique identifier of this element",
        "type": "string",
        "readOnly": true
      },
      "interactionDate": {
        "type": "string",
        "format": "date-time"
      },
      "isComplaint": {
        "type": "boolean",
      },
      ...
```

The schema definition for `InteractionWithVendor_Ext` would include the following:

```
"definitions": {
  "InteractionWithVendor_Ext": {
    "properties": {
      "id": {
        "title": "ID",
        "description": "The unique identifier of this element",
        "type": "string",
        "readOnly": true
      },
      "interactionDate": {
        "type": "string",
        "format": "date-time"
      },
      "isBillable": {
```

```
        "type": "boolean",
      },
      ...
```

Mapping definitions

After configuration, the mapping definition for `Interaction_Ext` would include the following:

```
"mappers": {
  "Interaction_Ext": {
    "properties": {
      "id": {
        "path": "Interaction_Ext.RestId",
      },
      "interactionDate": {
        "path": "Interaction_Ext.InteractionDate",
      },
      ...
```

The mapping definition for `InteractionWithInsured_Ext` would include the following:

```
"mappers": {
  "InteractionWithInsured_Ext": {
    "properties": {
      "id": {
        "path": "InteractionWithInsured_Ext.RestId"
      },
      "interactionDate": {
        "path": "InteractionWithInsured_Ext.InteractionDate"
      },
      "isComplaint": {
        "path": "InteractionWithInsured_Ext.IsComplaint"
        }
      },
      ...
```

The mapping definition for `InteractionWithVendor_Ext` would include the following:

```
"mappers": {
  "InteractionWithVendor_Ext": {
    "properties": {
      "id": {
        "path": "InteractionWithVendor_Ext.RestId"
      },
      "interactionDate": {
        "path": "InteractionWithVendor_Ext.InteractionDate",
      },
      "isBillable": {
        "path": "InteractionWithVendor_Ext.IsBillable"
        }
      },
      ...
```

Updater definitions

After configuration, the updater definition for `Interaction_Ext` would include the following:

```
"updaters": {
  "Interaction_Ext": {
    "properties": {
      "interactionDate": {
        "path": "Interaction_Ext.InteractionDate",
      },
      ...
```

The updater definition for `InteractionWithInsured_Ext` would include the following:

```
"updaters": {
  "InteractionWithInsured_Ext": {
    "properties": {
      "interactionDate": {
        "path": "InteractionWithInsured_Ext.InteractionDate"
      },
      "isComplaint": {
        "path": "InteractionWithInsured_Ext.IsComplaint"
        }
      },
      ...
```

The updater definition for `InteractionWithVendor_Ext` would include the following:

```
"updaters": {
  "InteractionWithVendor_Ext": {
    "properties": {
      "interactionDate": {
        "path": "InteractionWithVendor_Ext.InteractionDate",
      },
      "isBillable": {
        "path": "InteractionWithVendor_Ext.IsBillable"
        }
      },
      ...
```

Root resource endpoints

Root resource endpoints in Cloud API

Root resources

An endpoint's path consists of an API and a path to a resource element or collection.

For example, given the endpoint path `/account/v1/accounts/{accountId}/activities`:

- `/account/v1` is the API
- `accounts/{accountId}/activities` is the path.

The first resource listed in the path is known as the *root resource* for the endpoint. A root resource is a resource that is identified without any reference to any other resource. When retrieving root resources, you must query the entire database (with or without a query filter).

In the example above, the root resource is `account`.

For a given endpoint, all resources after the root resource are child resources. A child resource is a resource that is identified with reference to some other resource. When retrieving child resources, you only need to query for child instances related to the appropriate parent.

Generating root resource endpoints for custom entities

In most cases, when you generate endpoints for a custom entity, you want the custom resource to be a child resource. This is because most custom entities are created to extend information about some existing base configuration entity, such as a `Claim`, `Policy`, or `Account`.

For example, suppose an insurer creates a `SecurityIssue` custom entity that tracks security issues about an account. When generating endpoints for this entity, you would typically have the `SecurityIssue` resource be a child to the Account resource. The GET endpoints would look like this:

- `GET /account/v1/accounts/{accountId}/security-issues`
- `GET /account/v1/accounts/{accountId}/security-issues/{security-issue-id}`

However, there may be cases where you want to generate endpoints for a custom entity and you want the custom resource to be a root resource. The REST Endpoint Generator supports this use case. If you were to generate endpoints for the `SecurityIssue` custom entity as a root resource, the GET endpoints would look like this:

- `GET /account/v1/security-issues`
- `GET /account/v1/security-issues/{security-issue-id}`

Endpoints and foreign keys

For child endpoints for custom entities, the data model entity must have a foreign key to its parent.

For root resource endpoints for custom entities, there are no foreign key requirements. A custom entity can have root resource endpoints, regardless of the presence or absence of foreign keys in the corresponding data model entity.

For example, suppose you had two custom entities: `CustomEntity1` has a foreign key to `Activity`, `CustomEntity2` has no foreign keys to any other entities.

- For `CustomEntity1`, you could generate either child endpoints (with `Activity` as the parent) or root resource endpoints.
- For `CustomEntity2`, you could generate root resource endpoints. But you cannot generate child endpoints.

Advantages to root resource endpoints

You can search for all instances of the resource or retrieve a given instance of the resource without having to first identify the parent.

Disadvantages to root resource endpoints

If the custom entity has a parent in the database, then whenever you create a new instance of the custom entity, you must specify the parent in the payload.

If you want to retrieve all instances of a given resource that are related to a given logical parent, you must add filtering logic to your call to restrict it to only the resource for that parent.

Root resource endpoints are not inherently limited to working with resources for a given parent. Therefore, there is a greater chance that a root resource endpoint can compromise performance by retrieving a large number of resources at one time.

Generating root resource endpoints

One of the first prompts in the REST Endpoint Generator is:

```
Is the <CustomEntity> entity at the root of the endpoint
path (GET /activities)? (y = yes; n = no, it is a child
of some other entity (GET /activities/{activityId}/notes))
```

To make root resource endpoints for a custom entity, answer this question with `y`.

For any set of custom endpoints, you have the ability to have the collection backed by a Java stream or a Gosu query. This is determined by the answer to the following question.

```
Is the collection resource backed by a (s)tream or (q)uery?
```

Guidewire does not recommend using Java streams with root resource endpoints. Java streams load the entire collection into memory, and root resource endpoints do not limit the collection to those objects associated with a single parent. Therefore, root resource endpoints backed by Java streams may lead to compromised performance.

Configuring root resource endpoints

The REST Endpoint Generator creates and modifies a set of files which require further configuration. Configuration for child endpoints is covered in the following topics:

- “Configuring the resource definition files” on page 135
- “Configuring glue and impl classes for generated endpoints” on page 147
- “Configuring authorization for generated endpoints” on page 155

Configuration of root resource endpoints differs from child resource endpoint configuration in the following ways.

Configuring the collection resource file

The buildBaseQuery method (query-backed collections)

The `buildBaseQuery` method creates a Gosu query that returns the collection resource related to the parent resource. Root resource entities have no parents. Therefore, in most cases, the query must return all instances of the corresponding data model entity.

The sample code for root resources does this. In most cases, all you will need to do is uncomment the query. For example:

```
protected override function buildBaseQuery() : IQueryBeanResult<CustomEntity_Ext> {
                    return Query.make(entity.CustomEntity_Ext).select()
```

The loadValues method (stream-backed collections)

The `loadValues` method is used for collections that are backed by a Java stream (and not a Gosu query). Guidewire does not recommend using Java streams with root resource endpoints. Java streams load the entire collection into memory, and root resource endpoints do not limit the collection to those objects associated with a single parent. Therefore, root resource endpoints backed by Java streams may lead to compromised performance.

The createMinimalChild method

The `createMinimalChild` method is used for POSTs on the collection resource. It creates a new instance of the entity. If the new instance is a child instance, it also attaches it to its parent. Root resource entities have no parents. Therefore, in most cases, this method only needs to return a new instance of the appropriate entity.

The sample code for root resources does this. In most cases, all you will need to do is uncomment the query. For example:

```
override function createMinimalChildElement(attributes : DataAttributes) : CustomEntity_Ext {
                    return new CustomEntity_Ext()
```

The canViewException and canCreateException getters

For root resource endpoints, the REST endpoint generator does not generate either a `canViewException` getter or a `canCreateException` getter. The purpose of these getters is to determine if an instance of the resource can be viewed or created based on the business state of the parent. A root resource has no parent, and this these getters are not needed.

Configuring the element resource file

The init method

For root resource endpoints, the REST endpoint generator does not generate an `init` method. The primary purpose of this method is to validate consistency between the new instance and its parent. For a root resource, there is no parent, so there is no consistency to validate.

Configuring resource access for generated files

Resource access determines which instances of a given resource a caller can access. The REST Endpoint generator defaults resource access for root resources in the following ways. It is up to the developer to replace the defaults with more appropriate code as needed.

Note that root resources have no parent, and therefore the `__inherit` keyword cannot be used for collections.

The following sections detail the default settings for the extension files for some of the base configuration resource access strategies. These settings assume that the custom entity is named `CustomEntity_Ext`:

internal_ext-1.0.access.yaml

```
CustomEntitiesExt:
    view: "TODO RestEndpointGenerator"
    create: "TODO RestEndpointGenerator"
  filter: "TODO RestEndpointGenerator"
CustomEntityExt:
  permissions:
    view: __inherit
    edit: "TODO RestEndpointGenerator"
    delete: __inherit
```

accountholder_ext-1.0.access.yaml

```
CustomEntitiesExt:
  permissions:
    view: "TODO RestEndpointGenerator"
    create: "TODO RestEndpointGenerator"
  filter: "TODO RestEndpointGenerator"
CustomEntityExt:
  permissions:
    view: "TODO RestEndpointGenerator"
    edit: "TODO RestEndpointGenerator"
    delete: "TODO RestEndpointGenerator"
```

unauthenticatedUser_ext-1.0.access.yaml

```
CustomEntitiesExt:
  permissions:
    view: false
    create: false
CustomEntityExt:
  permissions:
    view: false
    edit: false
    delete: false
```

Root resource endpoint restrictions

There are some entities in the base configuration which are a root resource in one API and a child resource in another. For example:

- In ClaimCenter, `Activity` is a root resource in the Admin API but a child resource in the Claim API.
- In PolicyCenter, `Activity` is a root resource in the Admin API but a child resource in the Account API.

However, for custom resources, the REST Endpoint Generator will generate endpoints only as a root resource or only as a child resource. You cannot run the REST Endpoint Generator twice for the same custom entity to create both root and child endpoints.

There are some entity types, such as custom risk units, that must be a direct or indirect child of Policy. For these types of entities, the REST Endpoint Generator will not allow you to create root resource endpoints. The root resource endpoint prompt is suppressed.

There are some entity types, such as effective-dated entities, that must have a parent. For these types of entities, the REST Endpoint Generator will not allow you to create root resource endpoints. The root resource endpoint prompt is suppressed.

part 6

Generating LOB-specific endpoints

Cloud API has endpoints that caller applications can use to interact with contents of a policy. In this context, the term "policy" refers to bound policies, unbound policies associated with jobs such as policy changes, and future policies associated with submissions. The Cloud API endpoints can retrieve information about the contents of these policies. Within the context of a job, they can also create and modify the contents of a policy.

The contents of a policy can be divided into two categories:

- **LOB-generic**
  - These contents have a structure that remains the same, regardless of the policy's line(s) of business.
  - Examples: policy contacts, policy locations
- **LOB-specific**
  - These contents have a structure that varies for each line of business.
  - Examples: coverables, coverages, modifiers

The base configuration contains endpoints for LOB-generic policy contents. But, it does not initially contain endpoints for LOB-specific policy contents because insurers structure each product to suit the needs of the business.

In order to work with LOB-specific contents through Cloud API, you must first generate LOB-specific endpoints for each line.

> **Note:** PolicyCenter provides a mechanism for generating endpoints that let you work with the LOB-specific aspects of policies and jobs. However, this mechanism does not generate or modify the endpoints in the Product Definition API.

Related Information

- “Generating LOB-specific endpoints for APD-native products” on page 183
- “Generating LOB-specific endpoints for non-APD-native products” on page 185
- “Regenerating LOB-specific endpoints” on page 211

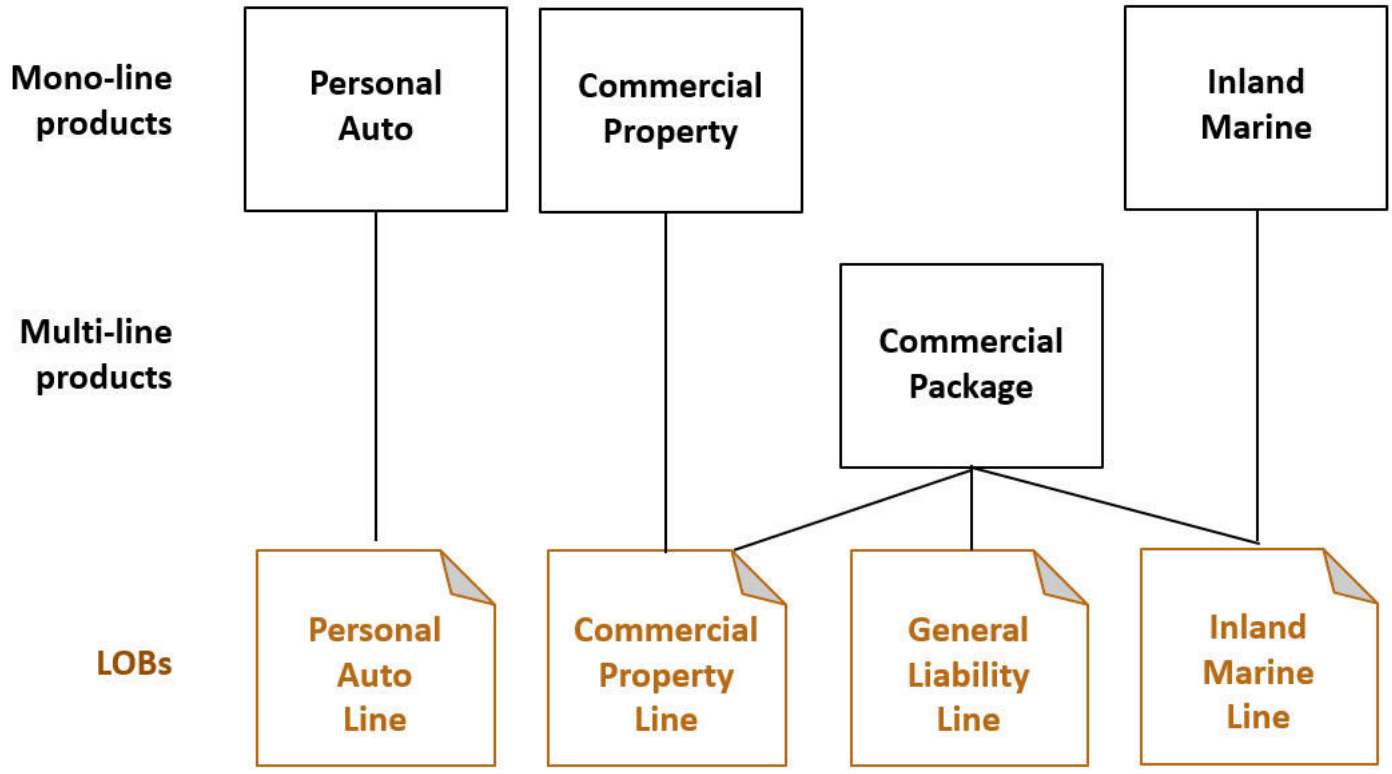
chapter 31

Products and lines of business

A *product* is a type of policy offered by an insurer. For example, an insurer may have a "Personal Auto" product that is used to create personal auto policies.

Every product consists of one or more lines of business. A *line of business* (LOB) is a collection of coverables, coverages, and other policy contents that are defined as a unit. For example, an insurer may have a "Personal Auto Line" that defines vehicle and driver coverables, vehicle and driver coverages, and so on.

The possible relationships between products and LOBs are depicted in the following diagram.



Most products have a single LOB. They are referred to as *mono-line products*. These are depicted in the first row of the diagram. For example, the Personal Auto, Commercial Property, and Inland Marine products all contain a single LOB. For these types of products, the product itself and the underlying LOB are often discussed interchangeably.

Some products have multiple lines of business. They are referred to as *multi-line products* or *package products*. This is depicted in the second row of the diagram. For example, an insurer could offer a Commercial Package product that consists of three lines of business, the "Commercial Property Line", the "General Liability Line", and the "Inland Marine" line.

Furthermore, an LOB is not limited to being used by one product.

- An LOB could be used by only a single product. For example, a Personal Auto Line LOB is typically used by only one product, the Personal Auto product.
- An LOB could be used by multiple products. For example, a Commercial Property LOB could be used by a Commercial Property product (a mono-line product) and a Commercial Package product (a multi-line product).

The endpoints that you generate act upon aspects of the LOB, not the product. If two products make use of the same LOB, and both products are exposed to Cloud API, then the endpoints for that LOB ought to behave the same in each product. Thus, the endpoints are referred to as *LOB-specific* endpoints, not product-specific endpoints.

Product sources

A PolicyCenter product can be created from any of the following sources:

- **Advanced Product Designer (APD) App** - APD App is a business tool that helps you design, simulate, and deploy an insurance product. Guidewire recommends creating all new products through APD App.
- **Standards Based Template (SBT)** - An SBT is a set of files you can add to an instance of PolicyCenter to implement a product that is based on content licensed from a standards bureau such as ISO or NCCI.
- **Product Designer** - Product Designer is a web-based tool for examining and editing products in the PolicyCenter product model. Product Designer was one of the first tools Guidewire offered for product design. Guidewire no longer recommends creating products with Product Designer. But, insurers may have existing products that were created with Product Designer before other tools were available.
- **Base configuration products** - This is a product that is provided with the base configuration of PolicyCenter.
  - The base configuration products are not installed in the base configuration itself. To implement them, you must download and install the appropriate extension pack business template. For more information, see the *Application Guide*.

LOB artifacts

Within the context of product development, an *LOB artifact* is a PolicyCenter resource that is used to manage and present the LOB-specific portions of a policy. LOB artifacts include the following:

- Database tables that store LOB-specific information
  - These are required for all products.
- LOB-specific reference tables
  - Some LOBs require one or more reference tables. For example, a Workers' Compensation LOB usually requires a reference table for storing class codes.
- LOB-specific PCFs (Page Configuration Files)
  - These files define the user interface used when one logs on to PolicyCenter. They are required when the product is available to the PolicyCenter user interface.
- LOB-specific endpoints
  - These are required when the policy is exposed through Cloud API.

When LOB-specific endpoints are generated for a product, they are added to two APIs in Cloud API.

- In the **Job API**, PolicyCenter generates several sets of GET, POST, PATCH, and DELETE endpoints. There is one set for each type of policy object, such as coverables, coverages, and exposures. These endpoints can be used to work with policies attached to a job.

- In the **Policy API**, PolicyCenter generates several GET endpoints. There is one for each type of policy object, such as coverables, coverages, and exposures. These can be used to retrieve information about bound policies.

The general pattern is that for any given LOB-specific object type (such as a Personal Auto product's Vehicle), there is a full set of CRUD endpoints in the Job API and element and collection GETs in the Policy API.

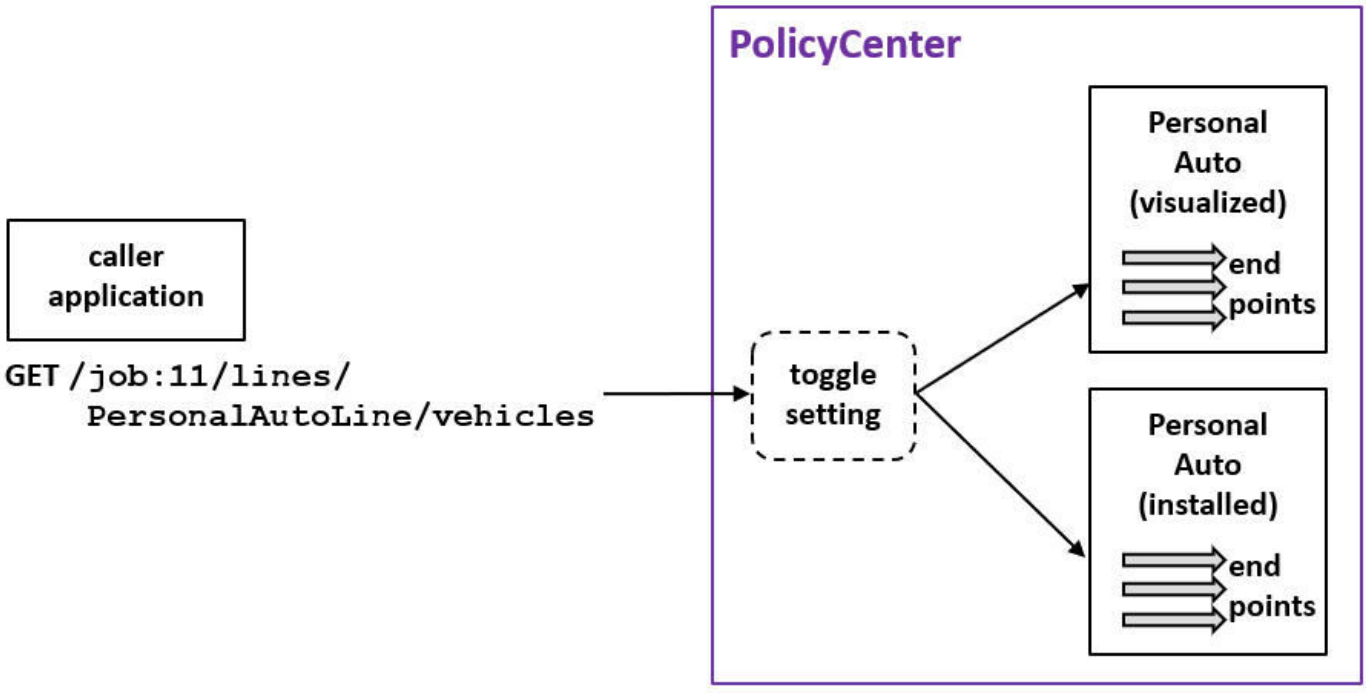
Visualized and installed products

Within the context of product development, a product can be in one of two states:

- **Visualized** - The product exists in a "draft" state.
  - Some product-specific artifacts (such as the product-specific `coverages` database table) have not yet been created.
- **Installed** - The product exists in a "finalized" state.
  - All product-specific artifacts (including the product-specific `coverages` database table) have been created.

Every visualized product has a set of Cloud API endpoints. These endpoints are created automatically as you create and edit a product. You can also manually generate a set of endpoints for an installed product. Thus, there can be two versions of a product (a visualized version and an installed version), and each version can have its own set of endpoints.

For a given product, there can only be one set of active endpoints at a given time. The active endpoints are the ones used for incoming calls from a third-party application. For the purposes of testing and development,PolicyCenter gives you the ability to toggle which set is active. This is depicted in the following diagram.



For more information on how to set the toggle, see “Toggling between visualized and installed endpoints” on page 213.

chapter 32

Generating LOB-specific endpoints for APD-native products

An APD-native product is a product that you create in the APD App with a mindmap, from scratch, or from a GO product template from the Guidewire Marketplace.

For visualized APD-native products that are enabled for REST API, PolicyCenter generates API endpoints when you make a REST API request after updating the product. Changes that trigger API regeneration include uploading a modified template in PolicyCenter or synchronizing PolicyCenter with the Cloud.

PolicyCenter generates API endpoints for APD-native products when you install the product and generate the product code. To install an APD product and generate endpoints, use the **Generate Product Code** option on the **Product Management screen** and choose to generate **System APIs - Code**. For more information, see “Regenerating LOB-specific endpoints” on page 211.

chapter 33

Generating LOB-specific endpoints for non-APD-native products

For all installed products that are not APD-native, PolicyCenter supports *bootstrapped generation* of LOB-specific endpoints through a process known as Cloud Retrofit. With Cloud Retrofit, the majority of the endpoint generation is automatic. However, a small amount of manual intervention is required.

For a given product, the generation process both requires and creates several files.

Cloud Retrofit with the APD App

If you have the APD App and you have products that were not created in the APD App, the process of generating LOB-specific endpoints is performed with Guidewire Studio, PolicyCenter **Product Management** screen, and the APD App.

The process involves the following steps:

1. In Guidewire Studio, define the `template_gen_config.<lob_prefix>.yaml` file for the line of business.
2. In PolicyCenter, export the product and resolve any errors.
3. In APD App, create a new product.
4. In Guidewire Studio, create or modify the codegen file as needed.
5. In PolicyCenter, synchronize PolicyCenter with APD and then generate the code for the product.
6. In the APD App, delete the product.

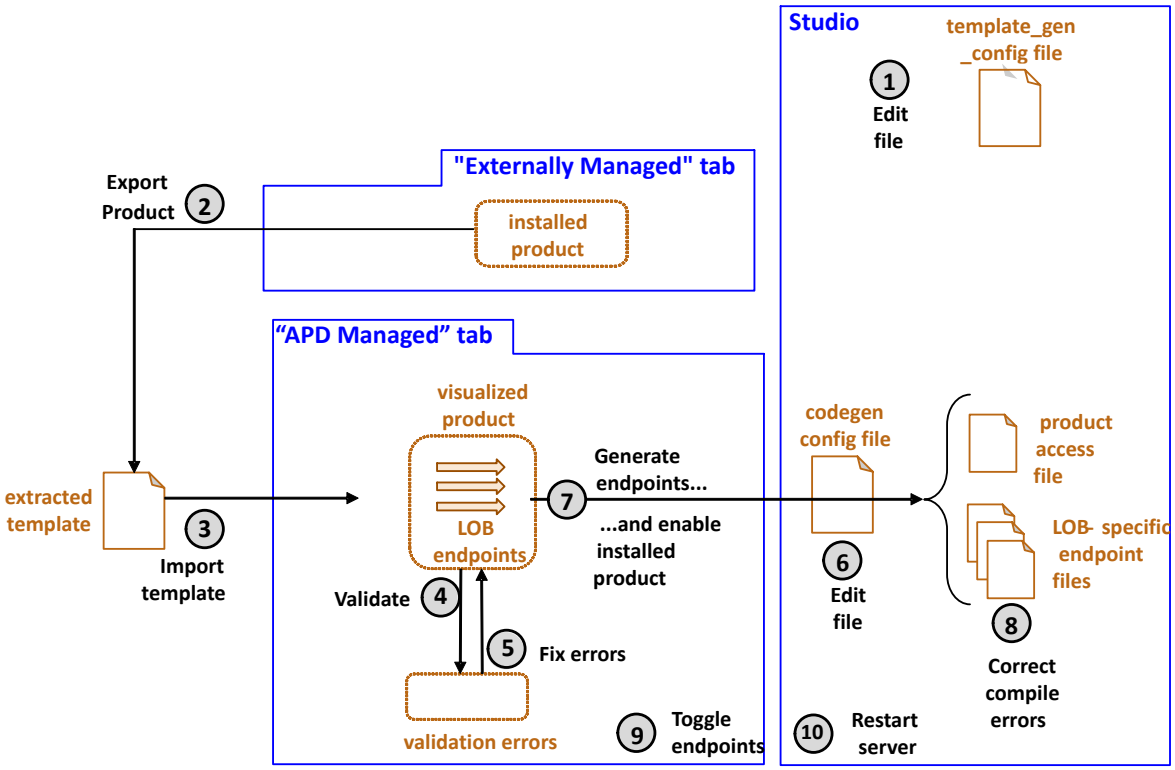
For a more detailed description of the process, see “Cloud Retrofit with APD App” on page 197

Cloud Retrofit without the APD App

When you do not use the APD App, the process of generating LOB-specific endpoints is performed with the PolicyCenter **Product Management** screen and with Guidewire Studio.

> **Note:** This is *not* the recommended Cloud Retrofit process. Consider provisioning the APD App and then following the process for “Cloud Retrofit with APD App” on page 197.

The process involves the following steps:



1. In **Studio**, update the `template_gen_config.ml` file for the line of business.
2. From the **Externally Managed** tab, extract a product template from the installed product.
3. From the **APD Managed** tab, import the product template. This creates a visualized version of the product. It also creates a set of LOB endpoints you can use to interact with the visualized product.
4. From the **APD Managed** tab, validate the visualized product. This potentially generates a list of errors.
5. If there are any validation errors, you must fix them before you can generate endpoints.
   - When fixing errors, Guidewire recommends that you modify the visualized product directly and then export a new version of the product template from the visualized product. This ensures your product template file matches the visualized product.
6. In **Studio**, create or modify the codegen config file, if necessary.
   a. Identify mismatches between type and field names in the APD representation and the corresponding names in the installed product.
   b. Identify other constraints that you want to declare or override.
7. From the **APD Managed** tab, generate endpoints for the installed product.
   a. This creates the product access file.
   b. This also creates a set of LOB-specific endpoint files for the installed product. These files reflect any overrides specified in the codegen config file.
8. Check the generated endpoint files for compile errors and correct them.
   - This typically involves updating the product template and/or the codegen config file and then regenerating the endpoints.
9. From the **APD Managed** tab, toggle the endpoints so that the installed product's endpoints are active.
10. Restart the server.

For a more detailed description of the process, see “Cloud Retrofit without APD App” on page 199.

Related Information

- “Regenerating LOB-specific endpoints” on page 211

Files required for Cloud Retrofit process

Each Cloud Retrofit process uses a different set of configuration files to provide PolicyCenter with information about how to generate LOB-specific endpoints for a given non-APD native product:

Cloud Retrofit Process	Files used by the process
Cloud Retrofit without APD App	• <lob_prefix>_codegen_config_ext-1.0.yaml • template_gen_config.xml
Cloud Retrofit with APD App	• <lob_prefix>_codegen_config_ext-1.0.yaml • template_gen_config.xml • template_gen_config.all.yaml • template_gen_config.<lob_prefix>.yaml

<lob_prefix>_codegen_config_ext-1.0.yaml
Defines aspects of how the REST APIs are generated. For example, you can define overrides for certain behaviors such as auto-numbering and available endpoints for a particular type. These files can be used anytime REST API endpoints are generated, including Cloud Retrofit, APD native, and APD conversion (EA).

template_gen_config.xml
Defines the APD model structure needed to generate LOB-specific endpoints for the *Cloud Retrofit without APD App* process. For the *Cloud Retrofit with APD App* process, this file needs to be edited to remove LOB-specific information for LOBs that you create in the APD app. For more information, see “Template Gen Config XML File” on page 194.

template_gen_config.all.yaml
Defines aspects of how the REST APIs are generated for *Cloud Retrofit with the APD App* and APD conversion. This file is available in the base configuration and it applies to all LOBs. If required, you can modify the file to change default values and define configuration details that can't be automatically inferred from the data model when generating the REST APIs.

template_gen_config.<lob_prefix>.yaml
Template gen config YAML files are used for *Cloud Retrofit with the APD App* and APD conversion. This LOB-specific file allows you to change default values and define configuration details that can't be automatically inferred from the data model when generating the REST APIs. You have the option to create this file from scratch or with a generated file. For more information, see “Define a Template Gen Config file for a LOB from scratch” on page 193 and “Define a Template Gen Config file for a LOB from a generated file ” on page 194.

Template Gen Config YAML Files

When you perform *Cloud Retrofit with APD App*, you must ensure that the configuration files are set up correctly in order for the lines to be generated successfully.

The following are examples of configurations you might need to add to a template gen config file because the information can't automatically be inferred from the data model:

- Define entities to treat as coverables in the APD model in cases where the entity doesn't already implement the Coverable interface, but has child coverables or exposures

- Define entities to treat as exposures in the APD model
- Specify which columns correspond to auto-number sequences, and the auto-number columns themselves
- Identify fields as splittable

Examples of defaults you might want to change include:

- Mark fields as ignored that you don't want to appear in the APD model (such as internally-managed values)
- Ignore some subtypes in cases where a root type is shared across LOBs
- Override field names in cases where the name is illegal in APD (such as "Type" or "Use") or you want a clearer name in the APD model (for example, the name contains cryptic abbreviations).

File names and locations

Template gen config files are located in the `modules\configuration\config\apd\model\generate\json` folder, and are typically named using the following format:

template_gen_config.<*lob-prefix*>.yaml

The lob-prefix is the code for the line that is found in the `InstalledPolicyLine` typelist, lowercased. For example, the template config file for PersonalAuto would be named template_gen_config.pa.yaml.

Each template gen config file can have up to three top-level properties: **lineCode**, **languages**, and **types**.

The languages property

This property specifies an array of language codes to include languages for, from the `LanguageType` typelist. By default, all languages configured in the `LanguageType` typelist are included, with the exception of "EDG" languages. Therefore, you need to specify this property only in one of the following two scenarios:

- You want to exclude languages that exist in `LanguageType` that haven't actually been localized.
- You want to include the EDG languages.

Example

```
languages:
  - en_US
  - ja
  - fr_CA
```

The lineCode property

This property is the only required property in the YAML file. It specifies the LOB that the configuration applies to. This value can be one of the following:

- All: Indicates that the template gen config file contains a cross-line configuration.
- <code-identifier>: The `PolicyLinePattern` code identifier of the LOB.

This example is for configurations that apply to multiple lines:

```
lineCode: All
```

Here is an example for a template config file that applies to the PersonalAutoLine:

```
lineCode: PersonalAutoLine
```

Note that if an LOB name is overridden in the codegen config file, the `lineCode` is the override name. For example, the preceding LOB could have an entry in the codegen config file as follows:

```
PALine:
 nameOverride: PersonalAutoLine
```

The monoLineProductCode property

Extracting the APD product ZIP file for a package (multi-line) product produces a file that can be uploaded to APD App, provided that the lines included in that package have been uploaded to APD as mono-line products.

In a JSON template, lines are referenced by the `line_links` property. The `line_links` property uses the appropriate mono-line product code, which is determined as follows:

- Look for a `monoLineProductCode` property in the line's `template_gen_config` file.
- If the `monoLineProductCode` property is not found, look for a mono-line product already configured with that line:
  - If there is exactly one product that includes the line, use its product code.
  - If there are no products that include the line, a default algorithm is used to determine which product code to use.
  - If there are several products that include the line, use the first product's code (in alphabetical order) and warn the user to configure the `monoLineProductCode` property in `template_gen_config`.

If the `monoLineProductCode` is set for a line, a product containing only that line is generated as a package product if the product's code does not match the `monoLineProductCode`. For example, if the `LineC` line is shared between the `ProductA` product and the `ProductB` product, and `monoLineProductCode` is set to `ProductA`, exporting the `ProductA` product includes the `LineC` line, but exporting the `ProductB` product does not include the `LineC` line and instead references the line in the `ProductA` product.

The types property

The `types` property contains a map of type configuration. The keys can be concrete type names, supertype names, or interface/delegate names. The following properties can be applied to a type under the `types` property.

autoNumberColumn

For entities that are auto-numbered, this field specifies the auto-number column. Specifying a value will cause the entity to be marked as auto-numbered in the APD model. The auto-number column itself will not appear in the APD model (its existence is implied by the auto-number flag, so it's not explicitly modeled), and the associated schema property will be read-only in generated APIs. If no value is specified, codegen assumes a value of "SequenceNumber."

Example

```
types:
 PersonalVehicle:
   autoNumberColumn: VehicleNumber
```

coveragesArray, conditionsArray, and exclusionsArray

Set the `coveragesArray`, `conditionsArray`, or `exclusionsArray` properties on a type if the associated Coverable entity has multiple child arrays of a given Clause type. For example, when Coverables that represent blankets have arrays of their own Coverages as well as arrays of blanketed Coverages, the `coveragesArray` property must be configured using the name of the array property that represents the blanket's Coverages rather than the name of the array property that represents the blanketed Coverages.

fields

A map of field configurations for this type. The keys in the object must correspond to the property names on the associated type. For more information on field-level configurations, see "The types property fields" on page 192 below.

handling

Use this property for entities that do not implement a coverable or modifiable. Possible values are `coverable`, `exposure`, or `location`.

- `coverable`: Treat this entity as a Coverable in the APD model. Use this handling value for entities with children that need to be included in the APD model but that aren't themselves Coverables.

- `exposure`: Treat this entity as an Exposure in the APD model. Use this handling value for leaf entities that have no children and aren't Coverables.
- `location`: Treat this entity as a Location-Based Coverable in the APD model.

If handling is not specified, default logic will be used to try to automatically determine how to handle the entity. The following logic will be used:

- Entities that implement the `Coverable` delegate will be treated as coverables.
- Entities that subtype the `AddlInterestDetail` entity will be treated as exposures.
- Entities that implement the `LineSpecificLocation` delegate will be treated as locations.

If none of the preceding can be determined, handling must be specified for the type. Entities with no configured handling, and that aren't automatically detected as a coverable, exposure, or location-based coverable, will not be included in the resulting APD template.

Example

```
lineCode: PersonalAutoLine
types:
 VehicleDriver:
   handling: exposure
```

ignored

There are cases where you want to exclude an entity from the APD model. For example, you might want to exclude some subtypes from consideration when a supertype is shared across LOBs and only some subtypes apply to some LOBs. To exclude an entity, set `ignored` to `true`.

If a type is marked as ignored then:

- Child array and one-to-one properties that are exactly of the ignored type will be ignored when determining which child properties are Coverables or Exposures.
- Unhandled one-to-one or array properties of exactly the ignored type will not be reported as unknown properties.
- Any subtypes of exactly the ignored type will be excluded when flatting a coverable or exposure property into its subtypes.

The most common use cases for ignoring types are the following:

- You want to suppress warnings about unhandled array or one-to-one properties, in cases where the type really ought to be excluded entirely.
- You want to exclude specific subtypes of a base Coverable or Exposure type.

Example

```
lineCode: WorkersCompLine
types:
 WCFedCoveredEmployee:
   ignored: true
```

parentAutoNumberSequenceColumn

When an entity is auto-numbered, a foreign key to `AutoNumberSequence` is automatically added to the parent of that entity. A given entity can have multiple auto-numbered children, so it needs to know which foreign keys go with which children. The `parentAutoNumberSequenceColumn` property identifies the column on the entity parent that contains that key.

In the following example, for the PersonalVehicle entity this is set to `PersonalVehicleAutoNumberSeq`, which is the name of the foreign key property on PersonalAutoLine that is used to auto-number vehicles.

Example

```
lineCode: PersonalAutoLine
types:
```

```
  PersonalVehicle:
    autoNumberColumn: VehicleNumber
    parentAutoNumberSequenceColumn: PersonalVehicleAutoNumberSeq
```

typeName

Use this property to override the type name inferred for a coverable or exposure.

By default, if the entity name starts with the LOB prefix, the type name used in the APD model is the name of the entity without the prefix. If the entity does not start with the LOB prefix, the entire entity name is used. This property can be used to override the name in the APD model.

Example

```
lineCode: PersonalAutoLine
types:
 VehicleDriver:
   typeName: Driver
```

other type fields

If you use the default configuration for JSON template generation for Businessowners, Commercial Property, and Workers' Compensation and are converting these LOBs into Guidewire Cloud, then you must modify your existing codegen configuration file. You must add the `other` type APD field configurations so that you can generate the corresponding schema/mapper/updater properties with the correct settings. After you make the changes shown below, you must run the API codegen again.

When converting an OOTB product to APD, there are elements that APD cannot support, such as class code and risk class. In these cases, the output JSON template for APD conversion generates fields of type `other`. Fields of type `other` are displayed in the APD product model, but are not used in APD.

If the OOTB product already supports `other` field types, when generating or re-generating LOB-specific endpoints for the lines/coverables with these fields, the correct codegen configuration is required so that the code can handle these fields and produce schema/mapper/updater JSON files.

Fields of type `other` cannot be added using APD.

`bop_codegen_config_ext-1.0.yaml`

```
types:
  BOPBuilding:
    fields:
      ClassCode:
        customType: ClassCode
  ...
...
```

`cp_codegen_config_ext-1.0.yaml`

```
types:
  CPBuilding:
    fields:
      ClassCode:
        customType: ClassCode
        updaterHandling:
          valueTypeResolver: gw.rest.ext.pc.policyperiod.cp.v1.building.CPClassCodeJsonValueResolver
  ...
...
```

`wc_codegen_config_ext-1.0.yaml`

```
types:
  WCEmployee:
    fields:
      ClassCode:
        createOnly: true
        customType: ClassCode
        requiredForCreate: true
  WCLine:
    fields:
      GoverningClass:
        customType: ClassCode
```

```
        readOnly: true
  WCWaiverOfSubro:
    fields:
      ClassCode:
        customType: ClassCode
  ...
...
```

The types property fields

Possible values for field properties are `fieldName`, `ignored` and `split`.

fieldName

Use this property to override the field name in the APD model.

By default the field name is the entity property name. The primary reason for overriding the field name is for cases where the name is an illegal identifier in APD. For example, you might need to change a field named Use to VehicleUse or Type to EmployeeType. You can also use this property in cases where you want to change a cryptically-named property to something more human readable.

Example

```
WCWaiverOfSubro:
  handling: exposure
  fields:
    Type:
      fieldName: WaiverType
```

ignored

Set `ignored` to `true` to exclude a field from the APD model in the following cases:

- The field is an implementation detail or internal bookkeeping, and does not represent a meaningful business value.
- The field represents something that cannot be modeled in APD, for example an effective-dated foreign key to something other than PolicyLocation or PolicyContactRole.

If a field is not modeled in APD, it cannot be managed in APD. Furthermore:

- When you generate code, the field is untouched in the `eti` file as only the pieces managed by APD are updated.
- You must manually add the field to Cloud APIs.

Example

```
lineCode: BOPLine
types:
  BusinessOwnersLine:
    fields:
      ViewBundledCoverages:
        ignored: true
```

split

Setting `split` to `true` will mark a field as `SplitByRatingPeriods` in the generated template. This value cannot be inferred automatically, so you must set it explicitly for any splittable fields, such as fields on Workers Comp employees or General Liability exposures.

Example

```
lineCode: GLLine
types:
  GLExposure:
    fields:
      AuditedBasis:
        split: true
```

Define a Template Gen Config file for a LOB from scratch

Create a template configuration file for a line of business (LOB) to change default values and define configuration details that can't be automatically inferred from the data model when generating the REST APIs. You can create a template configuration file (`template_gen_config.lob_prefix.yaml`) for an LOB from scratch or you can generate it.

About this task

This task describes how to manually create the template configuration file for a LOB from scratch. For information on how to create the file from a generated template, see “Define a Template Gen Config file for a LOB from a generated file ” on page 194.

Procedure

1. Using any editor, create an outline of the entities associated with the line of business. Creating a diagram is optional.
   a) Start at `PolicyLine.eti` and find the subtype for the LOB.
   b) Document the arrays, foreign keys, one-to-ones, and autonumber sequences for the LOB's policy line entity.
   c) Repeat the process for any entities referenced using arrays, foreign keys, or one-to-ones that are LOB-specific and that do not represent Clauses, ScheduledItems, Modifiers, Answers, PolicyContactRoles, Costs, or Transactions.
2. Create the `template_gen_config.lob_prefix.yaml` in the `modules/configuration/config/apd/model/generate/json/` folder.

   Note: The *`lob_prefix`* is the code for the line that is found in the `InstalledPolicyLine` typelist. For consistency with YAML files in the base configuration, we recommend using the lowercase version of the line prefix. Guidewire may have a YAML file available for the LOB. Contact your Guidewire representative for support.
3. Add the following properties to the `template_gen_config.lob_prefix.yaml` file.
   a) Add the `lineCode`. For more information, see “The lineCode property” on page 188.
   b) Under the `types` property, add each entity that has an `autoNumberSequence`. Then, configure the `autoNumberColumn` and `parentAutoNumberSequenceColumn` as needed. For more information on the sequencing options, see “The types property” on page 189.
   c) Under the `types` property, add each entity that does not implement either of the Coverable or Modifiable interfaces. Then, configure its `handling` as `exposure`, `coverable`, or `location` as needed. For more information on handling options, see “The types property” on page 189.
   d) Add the `fieldName` property to override each field that has the name of a reserved keyword.

      For example, the `WCWaiverOfSubro` entity has a field with a name of `Type`, which is not a legal field name in APD. The following entry overrides this field name with `WaiverType`:

      ```
      WCWaiverOfSubro:
         handling: exposure
         fields:
           Type:
             fieldName: WaiverType
      ```

      For more information on `fieldName` property, see “The types property fields” on page 192.
   e) As needed, add additional configurations or make additional modifications to entities under the types property, such as:
      - Add `ignored` and set it to `true` to exclude entities from the APD model.
      - Add a `typeName` to override the type name inferred for a coverable or exposure.

      For information on all the configuration options, see “Template Gen Config YAML Files” on page 187

What to do next

Complete the steps in “Cloud Retrofit with APD App” on page 197 to continue the process of generating the LOB-specific endpoints for a non-APD native product.

Define a Template Gen Config file for a LOB from a generated file

You can generate a Template Gen Config file for line of business (LOB) and then use the generated file to complete the configuration based on your requirements. As part of the *Cloud Retrofit with the APD App* and APD Conversion (EA), you can use the Template Gen Config file (`template_gen_config.lob_prefix.yaml`) to change default values and define configuration details that can't be automatically inferred from the data model when generating the REST APIs.

About this task

When PolicyCenter generates the template configuration file for a LOB, it creates a `template_gen_config.lob_prefix.yaml` file in the `config/apd/model/generate/json/` folder. Then, it adds the following content to the file based on the LOB configuration:

- If a type is not configured as a coverable, exposure, or location or marked as `ignored`, it is added with a default of `exposure` handling and a `TODO` comment indicating that the handling must be set, or that the type must be marked as `ignored`.
- If a field has an illegal name, the `fieldName` configuration is added and defaulted to the field's current name appended with the `_1` suffix. A `TODO` comment indicates that it must be renamed.
- If a field has a type APD cannot map, the field is added with the `ignored: true` flag and a `TODO` comment explaining why it is excluded.
- If autonumber sequences are unassigned, no configuration is added but a `TODO` comment is added to the containing type that explains which sequences are not handled and must be added to the appropriate children.
- If there are multiple arrays of Coverables, PolicyConditions, or Exclusions on a Coverable, the `coveragesArray`, `conditionsArray`, or `exlcusionsArray` property is added with a default value, along with a `TODO` comment that indicates that the value must be set appropriately and lists the possible options.

Procedure

1. In PolicyCenter, select **Administration** > **Product Management**. Then, select the **Externally Managed** tab.
2. Select the LOB and click **Create Template Configuration**.

   Clicking **Create Template Configuration** creates a template gen config file for the LOB.
3. Review the generated `template_gen_config.lob_prefix.yaml` file and address any `TODO` comments.
4. Add new lines to the `template_gen_config.lob_prefix.yaml` file as needed. For example, set `ignored` to `true` to exclude entities from the APD model.

   For more information about the configuration options, see “Template Gen Config YAML Files” on page 187.

What to do next

Complete the steps in “Cloud Retrofit with APD App” on page 197 to continue the process of generating the LOB-specific endpoints for a non-APD native product.

Template Gen Config XML File

When you perform *Cloud Retrofit without APD App*, you can add configuration to `template_gen_config.xml` so that the templates for lines of business generate successfully. For the *Cloud Retrofit with APD App* process, this file needs to be edited to remove LOB-specific information for LOBs that you create in the APD app.

For *Cloud Retrofit without APD App*, the following are examples of configurations you might need to add to template_gen_config.xml:

- Define entities to treat as coverables in the APD model, in cases where the entity doesn't already implement the Coverable interface but has child coverables or exposures
- Define entities to treat as exposures in the APD model
- Mark fields as ignored that you don't want to appear in the APD model (such as internally-managed values)
- Override field names in cases where the name is illegal in APD (such as "Type" or "Use") or you want a clearer name in the APD model (for example, the name contains cryptic abbreviations).

AllLines, AllTypes, and PolicyLine elements

The `AllLines` element includes configuration that applies to all lines of business. The `PolicyLine` element includes configuration that is specific to a LOB. You can configure `Type` and `Field` elements as children of these elements.

Type elements

When you define a `Type` element, you provide a `name` attribute and an additional attribute that specifies a property of that `Type`.

Possible properties of the `Type` element are the `handling` attribute and the `autonumber` attribute.

`handling`

Define this attribute for entities that do not implement a coverable or modifiable. Possible values are `coverable`, `exposure`, or `location`.

- `coverable`: Treat this entity as a Coverable in the APD model. Use this handling value for entities with children that need to be included in the APD model but that aren't themselves Coverables.
- `exposure`: Treat this entity as an Exposure in the APD model. Use this handling value for leaf entities that have no children and aren't Coverables.
- `location`: Treat this entity as a Location-Based Coverable in the APD model.

If `handling` is not specified, default logic tries to automatically determine how to handle the entity with the following logic:

- Entities that implement the `Coverable` delegate will be treated as coverables.
- Entities that subtype the `AddlInterestDetail` entity will be treated as exposures.
- Entities that implement the `LineSpecificLocation` delegate will be treated as locations.
- Entities with no configured handling, and that aren't automatically detected as a coverable, exposure, or location-based coverable, will not be included in the resulting APD template.

Example:

```
<PolicyLine code="TSTLine">
    <Type name="TSTCfgCoverable" handling="coverable"/>
    <Type name="TSTConfiguredExposure" handling="exposure"/>
    ...
  </PolicyLine>
```

`autonumber`

For entities that are auto-numbered, this attribute specifies the auto-number column. Specifying a value causes the entity to be marked as auto-numbered in the APD model. The auto-number column itself will not appear in the APD model (its existence is implied by the auto-number flag, so it's not explicitly modeled), and the associated schema property will be read-only in generated APIs. If no value is specified, codegen assumes a value of "SequenceNumber."

Example:

```
<PolicyLine code="PersonalAutoLine">
    ...
    <Type name="PersonalVehicle" autonumber="true">
      <Field name="VehicleNumber" ignored="true"/>
    </Type>
```

```
    ...
  </PolicyLine>
```

Field elements

When you define a `Field` element, you add it as a child of a `Type` element. Each `Field` element has a `name` attribute and an additional attribute that specifies a property of that `Field`.

Possible properties of the `Field` elements are the `templateName` attribute, the `ignored` attribute, and the `split` attribute.

`templateName`

Use this attribute to override the field name in the template. By default the field name that is included in the template is the entity name. The primary reason for overriding the field name is for cases where the name is a reserved keyword in APD. For example, you might need to change a field named Use to VehicleUse or Type to WaiverType. You can also use this attribute in cases where you want to change a cryptically-named field to something more human readable.

Example:

```
<PolicyLine code="WorkersCompLine">
   <Type name="WCWaiverOfSubro" handling="exposure">
      <Field name="Type" templateName="WaiverType"/>
    </Type>
</PolicyLine>
```

> **Note:** For each `templateName` attribute that you define in this file, configure the `nameOverride` at the field-level in the codegen config file to map the name you define for the APD template to the name defined in the installed product. For more information on working with codegen config files, see “Codegen config files” on page 235.

`ignored`

Set `ignored` to `true` to exclude a field from the APD model in the following cases:

- The field is an implementation detail or internal bookkeeping, and does not represent a meaningful business value.
- The field represents something that cannot be modeled in APD, for example an effective-dated foreign key to something other than PolicyLocation or PolicyContactRole.

If a field is not modeled in APD, it cannot be managed in APD. Furthermore:

- When you generate code, the field is untouched in the `eti` file as only the pieces managed by APD are updated.
- You must manually add the field to Cloud APIs.

Example:

```
<PolicyLine code="IMLine">
   ...
    <Type name="IMSign" autonumber="true">
      <Field name="SignNumber" ignored="true"/>
    </Type>
  </PolicyLine>
```

`split`

Set the `split` attribute to `true` to mark a field as `SplitByRatingPeriods` in the generated template. This value cannot be inferred automatically, so you must set it explicitly for any splittable fields, such as fields on Workers Comp employees or General Liability exposures.

Example:

```
<PolicyLine code="WorkersCompLine">
   ...
    <Type name="WCCoveredEmployee" handling="coverable">
      <Field name="AuditedAmount" split="true"/>
      <Field name="BasisAmount" split="true"/>
      <Field name="NumEmployees" split="true"/>
```

```
    </Type>
   ...
  </PolicyLine>
```

Cloud Retrofit with APD App

Users that have the APD App can use the following steps to generate LOB-specific endpoints for products that were not created in APD.

After reading this topic, you'll be able to:

- Define the product with template configuration files.
- Use PolicyCenter to export the product template to a ZIP file.
- Create the product from the template in the APD app.
- Create or modify the codegen file, as needed.
- Use PolicyCenter to generate the endpoints.
- Delete the product from the APD App.

Instructions

Step 1: Define the product in the template configuration files

About this task

Template gen config files allow you to change default values and define configuration details that cannot be automatically inferred from the data model when generating the REST APIs. For more information on the files in this process, see “Files required for Cloud Retrofit process” on page 187.

Procedure

1. If you want to generate endpoints for a LOB that was included in the base configuration of a previous release, you need to remove its associated LOB-specific entries from the `template_gen_config.xml`. Guidewire provides default entries in the `template_gen_config.xml` file to facilitate Cloud Retrofit for users that do not have the APD App.
   a) In Studio, open the `template_gen_config.xml`
   b) Search for a PolicyLine XML node for the LOB. For example, you can search for `<PolicyLine code="PersonalAutoLine">`.
   c) Delete the entries for the LOB.

      Note: Do not remove existing PolicyLine entries for LOBs that you are not generating endpoints for using this particular process.
2. Choose one of the following options to define a `template_gen_config.`*`lob_prefix`*`.yaml` file.

Define the `template_gen_config.`*`lob_prefix`*`.yaml` from scratch	See “Define a Template Gen Config file for a LOB from scratch” on page 193
Define a `template_gen_config.`*`lob_prefix`*`.yaml` from a generated file	See “Define a Template Gen Config file for a LOB from a generated file ” on page 194

3. *Optional:* Edit the `template_gen_config.all.yaml` file to ignore `NumAddInsured`.
   a) In Studio, open the file `template_gen_config.all.yaml`.
   b) As part of the `PolicyLine` entity, add an entry to ignore the `NumAddInsured` field so that it is not generated as an APD attribute of the LOB. Add the last two lines of the following code snippet:

```
PolicyLine:
   fields:
```

```
      PatternCode:
         ignored: true
      NumAddInsured:
         ignored: true
```

This field was not used in the LOBs that were included in the base configuration of previous releases, so it is unlikely that there are any occurrences.

4. Edit or create the *lob_prefix*_codegen_config_ext-1.0.yaml file.

   This file is used for generating the LOB-specific endpoints.

   a) Verify the contents of the file and make sure it is compatible with any changes to the product and the template generator configuration.

   b) Make sure the file complies with the rules for codegen config file syntax.

Step 2: Export the product

Procedure

1. In PolicyCenter, select **Administration** > **Product Management**. Then, select the **Externally Managed** tab.
2. Select the LOB and select **Export Product**.

   Selecting **Export Product** creates and exports a JSON representation of the LOB. A ZIP file with the name of the LOB is saved to your Downloads folder.
3. Check the server log for errors and warnings and address them.

   Review the server log for errors and warnings, and specifically for `WARN Application.APD` messages that are output as part of generating the JSON template. These warnings include information about pieces of the product that appear to be missing appropriate configuration in `template_gen_config.`*`lob_prefix`*`.yaml`, for example children that are not configured as coverables or exposures, field names that are not compatible with APD, and other problems that may occur during code generation. For example, you might see the error message `The field name Type is not a legal APD field name. Use the template_gen_config file to add an appropriate override for the field name to something legal.`
4. Export the product again and check the server log for any remaining warnings or errors.
5. Save the LOB template ZIP file for your records.

Step 3: Create a new product in APD App

Procedure

1. On the **Insurance Products** page of the APD App, select **Create New Product**.
2. Select the **From Template** tab.
3. Select the LOB template ZIP file.
4. Select **Create**.
   A tile for the new product appears.

Step 4: Create or modify the codegen config file

A codegen config file is a file that provides PolicyCenter with information about how to generate LOB-specific endpoints for a given non-APD-native product. For more information, see “Codegen config files” on page 235.

Step 5: Synchronize PolicyCenter with APD and generate the endpoints

Before you begin

In PolicyCenter **Preferences**, set the **Product Design Mode** to **Developer.**

Procedure

1. In PolicyCenter, select **Administration** > **Product Management**. Then, select the **APD Managed** tab
2. Click **Synchronize With Cloud** or click **Import From Cloud** and type the product identifier.

   To find the product identifier:

   a) In APD, click the product's tile to display the product.

   b) In the Navigation pane, select the product node.

   c) Click **View [Product Name]**.
   Product Identifier appears in **Element Details** and also in the URL.
3. Select the product from the list.
4. Generate the product code.

   a) Click **Generate Product Code**.

   b) Review the warnings at the top of the screen and in the list of elements. Perform updates to the codegen config file,as needed.

   c) In **Generate Options**, verify that the **System APIs - Code** is set to **Default Directory**.

   d) Click **Complete Generation** and then click **OK** to confirm.
   The following message appears: `Code has been successfully generated.`

Step 6: Delete the product from APD App

About this task

While you can use the APD App to generate the endpoints, the product cannot be maintained in the APD App. Therefore, delete the product from APD App after you successfully generate the endpoints.

Procedure

1. On the **Insurance Products** page of the APD App, locate the tile associated with the product you want to delete.
2. From the **...** menu of the product you want to delete, select **Remove product**.

Results

Removing this product from the APD App is done to avoid confusions between products that can be maintained in the APD app and those that you add to the APD App to facilitate the generation of endpoints.

Next steps

- To view the generated endpoints, restart the PolicyCenter server. For information on how to access the endpoints, see *PolicyCenter Cloud API Consumer Guide*
- To regenerate the endpoints, verify that the product is no longer in the APD App and then repeat the steps in this procedure.

Cloud Retrofit without APD App

Users that do not have the APD App can use the following steps to generate LOB-specific endpoints for products that were not created in APD.

Before you begin

- In PolicyCenter **Preferences**, set the **Product Design Mode** to **Developer.**

  Note: This is *not* the recommended Cloud Retrofit process. Consider provisioning the APD App and then following the process for *Cloud Retrofit with APD App*. For more information, see “Cloud Retrofit with APD App” on page 197.

WARNING: You can extract a product template from one instance of PolicyCenter and then use that template to generate endpoints on a different instance of PolicyCenter. However, do not generate LOB-specific endpoints for a product that is not installed on that instance of PolicyCenter. PolicyCenter will generate the endpoints. But, because the product itself is not installed, the endpoints will reference other LOB artifacts that do not exist on that instance of PolicyCenter. This will cause compile errors that prevent PolicyCenter from starting. (For information on how to remove the installed product endpoints, see "Removing an installed product's endpoints" on page 219.)

Procedure

1. In Guidewire Studio, update `template_gen_config.xml` for each LOB that you want to generate endpoints for.
   a) If a `PolicyLine` element does not already exist for the LOB, add a `PolicyLine` element with a `code` attribute set to the pattern code of that line.
   b) For each entity that does not implement either a Coverable or Modifiable interface, add a `Type` element as child of the `PolicyLine`. For each `Type` element that you add, set the `name` attribute to the name of the entity type and the `handling` attribute to `exposure`, `coverable`, or `location`.

      For example:

      ```
      <PolicyLine code="TSTLine">
          <Type name="TSTCfgCoverable" handling="coverable"/>
          <Type name="TSTConfiguredExposure" handling="exposure"/>
          ...
        </PolicyLine>
      ```

   c) For each field that has the name of a reserved keyword in APD, add a `Field` element as a child of the associated `Type` element, set the `name` attribute to the name of the field in the current entity model, and set the `templateName` attribute to the field name that you want to use in APD.

      For example:

      ```
      <PolicyLine code="WorkersCompLine">
         <Type name="WCWaiverOfSubro" handling="exposure">
            <Field name="Type" templateName="WaiverType"/>
          </Type>
      </PolicyLine>
      ```

      Note: For each `templateName` attribute that you define in this file, configure the `nameOverride` at the field-level in the codegen config file to map the name you define for the APD template to the name defined in the installed product. For more information on working with codegen config files, see "Codegen config files" on page 235.

   d) As needed, add additional configurations or make additional modifications to the configuration for the product's PolicyLines. For example, to exclude a field from the APD model, add a `Field` element as a child of the associated `Type` element and set the `ignored` attribute to `true`.

   For more information on all the configuration options, see "Template Gen Config XML File" on page 194.
2. In PolicyCenter, generate a product template for the existing installed product.
   a. Navigate to the **Product Management** screen.
   b. Click the **Externally Managed** tab.
   c. In the list of **Installed Products**, click the desired product.
   d. Click **Extract APD Representation**. This generates the XML template and stores it in your `Downloads` directory.
3. In PolicyCenter, import the product template.
   a. On the **Product Management** screen, click the **APD Managed** tab.
   b. Click **Import From Template**.
   c. Click **Browse**, then navigate to the template's location and select the template.

   d. Click **Update**. It may take a few seconds for the template to load. Once it has been loaded, the product is listed on the **APD Managed** tab.

4. In PolicyCenter,validate the installed product.
   a. On the **APD Managed** tab, select the product.
   b. On the **Details** tab below the list, select **Validation Status**. The **<ProductName>** screen appears.
   c. If the product has any errors, the screen title also says **"See Errors Highlighted"**.
5. Fix all validation errors on the **<Product Name>** screen, and optionally fix any warnings.
   a. Errors appear with an "!" in a red triangle. Warnings appear with an "!" in a yellow triangle.
   b. For more information on the most common types of errors and how to fix them, see “Fixing product validation errors” on page 203.
   c. Guidewire recommends making any required changes to the visualized product through the **Product Management** screen and then extracting an updated version of the product template from the **Product Management** screen. To do this, on the **APD Managed** tab, select the product from the product list and then click **Extract Template**.
6. In Guidewire Studio, modify the codegen config file, if necessary.
   a. Navigate to **modules > configuration > config > integration > apis > installedlobs**.
   b. Open the file whose name starts with `<productCode>_codegen`. If there is no codegen config file for your LOB and you need one, you must create one. Then, edit the codegen config file as needed.
   c. For more information on working with codegen config files, see “Codegen config files” on page 235.
7. In PolicyCenter, generate endpoints for the installed product.
   - On the **APD Managed** tab, select the product.
     a. On the **Details** tab below the list, select **Generate Product Code -> System APIs -> System APIs - Code**. The **Review Product Elements** screen appears.
     b. Click **Complete Generation**. In the confirmation dialog box, click **OK**.
8. Identify and correct all compile errors in the generated endpoint files.
   - For more information on how to identify and correct compile errors, see “Correcting compile errors” on page 209.
9. For a given product, there can only be one set of active endpoints at a time. By default, the visualized product endpoints are active. If you wish to continue working with the visualized product, no additional action is needed.

   Toggle the installed product endpoints to make them active
   a) On the **APD Managed** tab and select the product.
   b) In the **Enabled for REST API** field, select *Disabled* to enable the installed product's endpoint.
10. Restart PolicyCenter.

Files generated by the Cloud Retrofit process

When you generate the endpoints, several files are created.

The first set is the *files that define the endpoints themselves*. This includes schema, mapping, updater, and swagger files for the endpoints. For more information on these files, see “Files that define LOB-specific endpoints” on page 207.

Second, there is a *product access file* (`<productCode>_ext-1.0.yaml`). This file exposes the installed product to Cloud API. For information on disabling access to installed products, see “Disable an installed product's endpoints” on page 218.

chapter 34

Fixing product validation errors

Before you can generate endpoints from a visualized product, you must validate the product. This is done using the **Product Definition** screen's **Validation Status** button. When you click this button, PolicyCenter validates the visualized product and shows the results on the **<ProductName>** screen.

If you attempt to generate endpoints for a product that has validation errors, PolicyCenter will not generate the endpoints. It also displays a screen named the **Review Product Elements** screen. This is identical to the **<ProductName>** screen.

Error information in the <ProductName> screen

The **<ProductName>** screen lists the contents of the product as a hierarchical tree. Each warnings and error is noted with a triangle icon with a "!".

- Validation warnings appear after a yellow triangle with a "!". You are not required to fix warnings in order to generate endpoints.
- Validation errors appear after a red triangle with a "!". You must fix errors in order to generate endpoints.

When a given node in the tree has an error, the red triangle appears at that node and at every parent of that node, all the way to the top-level Product node.

For example, suppose you have a Workers' Compensation product which includes the following structure:

- Product: Workers' Compensation
  - Product Line: Workers' Compensation Line
    - Exposure: InclusionPerson
    - Exposure: WCWaiverOfSubro
      - Field: Type

Also, suppose there is a warning for the "Field: Type" node. The red triangle appears at this node, as well as the three parent nodes ("Exposure: WCWaiverOfSubro", "Product Line: Worker's Compensation Line", and "Product: Workers' Compensation") This is depicted in the following list, with "(!)" representing the red triangle.

- (!) Product: Workers' Compensation
  - (!) Product Line: Workers' Compensation Line
    - Exposure: InclusionPerson
    - (!) Exposure: WCWaiverOfSubro

- (!) Field: Type

If there are no errors for the "Exposure: InclusionPerson" node, then that node will have no triangle.

Unlike errors, warnings do not cascade up the tree. They appear only at the node that the warning pertains to.

Correcting "Short Name must be a valid name" errors

Every field in a visualized product has a *Short Name*. This value acts as a code for the field and it gets used for code identifiers in PolicyCenter. Guidewire has a list of reserved words that cannot be used for a `Short Name` field. This includes Java keywords, Gosu keywords, and the SQL keywords `use` and `type`. For more information on Short Name reserved words, see *Creating Products with APD App*.

The product validation results may include a "Short Name must be a valid name" error. This occurs when a field name was valid in the tool used to create the installed product, such as Product Designer, but it is one of the previously referenced reserved word. For example, a field named "Type" is valid in Product Designer but not as a `Short Name`.

Identifying the mismatch

When this type of error occurs, the **<ProductName>** screen shows the following next to the field's node:

```
Short Name must be a valid name.
```

Resolving the mismatch (for Cloud Retrofit without APD App)

1. In the APD representation, modify the `Short Name` field to a non-reserved name.
2. In the codegen config file, add a `nameOverride` to map the new, non-reserved name in the APD representation to the original name in the installed product. For more information on working with codegen config files, see “Codegen config files” on page 235.

Guidewire recommends that you re-extract the product template whenever you modify the visualized version of the product. To do this, on the **APD Managed** tab, select the product from the product list and then click **Extract Template**.

Resolving the mismatch (for Cloud Retrofit with APD App)

1. Update the field name with the `fieldName` property in the `template_gen_config.`*`lob_prefix`*`.yaml` file.
2. Delete the product from the APD App and then repeat the steps to generate the LOB-specific endpoints. For more information, see “Cloud Retrofit with APD App” on page 197.

Correcting "naming conflict with an existing field" warnings

In some situations, the `Short Name` assigned to a field is identical to some other existing field name. This triggers a "naming conflict with an existing field" warning. To generate LOB-specific endpoints, you are not required to fix warnings. Fixing warnings is options.

The product validation results may include a

The short name "Pattern Code" has a naming conflict with an existing field in PolicyCenter. Choose an alternative short name.

Identifying the mismatch

When this type of warning occurs, the **<ProductName>** screen shows the following next to the field's node:

```
The short name "<value>" has a naming conflict with an existing field in PolicyCenter.
Choose an alternative short name.
```

Resolving the mismatch (for Cloud Retrofit without APD App)

1. In the APD representation, modify the `Short Name` field to an alternate name.

2. In the codegen config file, add a `nameOverride` to map the new, non-conflicting name in the APD representation to the original name in the installed product. For more information on working with codegen config files, see “Codegen config files” on page 235.

Guidewire recommends that you re-extract the product template whenever you modify the visualized version of the product. To do this, on the **APD Managed** tab, select the product from the product list and then click **Extract Template**.

Resolving the mismatch (for Cloud Retrofit with APD App)

1. Update the field name with the `schemaPropertyName` field in the codegen config file. For more information on working with codegen config files, see “Codegen config files” on page 235
2. Complete the steps to generate the endpoints for the first time or update the endpoints. For more information, see “Cloud Retrofit with APD App” on page 197.

chapter 35

Files that define LOB-specific endpoints

When you generate LOB-specific endpoints for an installed product, PolicyCenter creates a series of files to define the endpoints and their behavior. There are two types of files: codegen output files and LOB extension files.

Codegen output files

Codegen output files define the LOB-specific endpoints. They include schema files, mapping files, updater files, swagger files, and Gosu resource files.

- These files have `<productCode>_gen` in their name
- These files are overwritten any time you regenerate the endpoints.

For example, if you generate endpoints for a Workers' Compensation product whose product code is "`wc`", then the codegen output files would include:

- `policyperiod_wc_gen-1.0.schema.json`
- `policyperiod_wc_gen-1.0.mapping.json`
- `policyperiod_wc_gen-1.0.updater.json`

Do not modify codegen output files directly. Any changes you make to these files will be overwritten the next time you generate endpoints for the product.

LOB extension files

LOB extension files define extensions to the endpoints. They include schema files, mapping files, updater files, swagger files, and Gosu resource files. These files have `<productCode>_ext` in their name.

If these files do not exist (such as when you have never generated installed endpoints for the given product), then these files are created when you generate the endpoints. But, if these files already exist, the codegen process does not modify them.

For example, if you generate endpoints for a Workers' Compensation product whose product code is "`wc`", then the LOB extension files would include:

- `policyperiod_wc_ext-1.0.schema.json`
- `policyperiod_wc_ext-1.0.mapping.json`
- `policyperiod_wc_ext-1.0.updater.json`

File locations

The codegen output files and LOB extension files are placed in the following directories in the `modules\configuration` directory:

- `config\integration\apis\ext\job\v1`
- `config\integration\apis\ext\policy\v1`
- `config\integration\apis\ext\policyperiod\v1`
- `config\integration\mappings\ext\policyperiod\v1`
- `config\integration\schemas\ext\policyperiod\v1`
- `config\integration\updaters\ext\policyperiod\v1`
- `gsrc\gw\rest\ext\pc\policyperiod\<productCode>\v1`

PolicyCenter also creates a product access file that exposes the installed product to Cloud API. This file is named `<productCode>_ext-1.0.yaml`. It is placed in the `modules\configuration\config\integration\apis\installedlobs` directory along with the codegen config files.

chapter 36

Correcting compile errors

The first time that you generate LOB-specific endpoints for an installed product, the generated files may have compile errors. The primary cause of these compile errors are naming conflicts caused by field name truncation related to Advanced Product Designer.

Field truncation in APD App

Advanced Product Designer limits type and field names to a certain number of characters. The specific number varies based on the type or field itself. For example, type names are limited to 24 characters. For more information on APD name length limits, see *Creating Products with APD App*.

If a name in the product template exceeds the corresponding APD character limit, then when you import the template, Advanced Product Designer truncates the field name in the visualized product. When you generate endpoints from the visualized product, field names in the endpoints do not match the corresponding field names in the installed product. This results in compile errors in one or more generated LOB endpoint files.

Identifying the mismatch

You can identify files with compile errors using your IDE.

If you enable the installed product endpoints, you can also identify the compile errors in the PolicyCenter console and logs when you attempt to use the endpoints.

For example, suppose you have a Workers' Compensation product with a type named `WCRetroRatingLetterOfCredit`. When the product template is imported, the type name is truncated to `WCRetroRatingLetterOfCre`. This results in the following errors:

- In the mapping file, there is an error stating `Cannot resolve symbol: WCRetroRatingLetterOfCre`.
- If you attempt to use the endpoints, you will see errors such as: `No property descriptor found for property, RetrospectiveRatingPlan, on class, entity.WCRetroRatingLetterOfCre`.

Resolving the mismatch

Mismatches can happen at the type level and at the field level.

For each mismatch, you must first identify the correct name. This is the name that appears in the entity definition (the `.eti` or `.ext` file) for the corresponding data model entity.

Then, in the product's codegen config file, you must add a `nameOverride` property that maps the truncated visualized product name with the correct installed product name.

For example, to fix the error in the previous example, the Workers' Compensation codegen config file would need the following entry:

```
types:
  WCRetroRatingLetterOfCre:
    nameOverride: WCRetroRatingLetterOfCredit
```

For more information on working with codegen config files, see “Codegen config files” on page 235.

Finally, you must regenerate the endpoints. This recreates the LOB-specific endpoints files with the correct type and field names. For more information, see “Regenerating LOB-specific endpoints” on page 211.

chapter 37

Regenerating LOB-specific endpoints

Regenerate LOB-specific endpoints after you resolve compilation errors in the files generated for the LOB-specific endpoints.

PolicyCenter will not automatically regenerate endpoints for the installed product if the only change you have made is to the codegen config file. Also, LOB-specific endpoints that are no longer needed after an update are not removed until you remove the previously generated files.

Use one of the following methods to regenerate LOB-specific endpoints:

- “Regenerate endpoints by first removing the template” on page 211
- “Regenerating endpoints by first removing the generated files” on page 212

  Note: For products that were imported into APD app as part of Cloud Retrofit, see “Cloud Retrofit with APD App” on page 197 for the steps to regenerate the endpoints.

Regenerate endpoints by first removing the template

You can trigger endpoint regeneration by removing and re-importing the product template, and then generating the endpoints. You can only use this method when PolicyCenter is running.

Before you begin

Verify that the PolicyCenter **Product Design Mode** is set to **Developer**. To view or set the product design mode, go to **Options** > **Preferences**.

Procedure

1. Navigate to the **Product Management** screen.
2. Click the **APD Managed** tab.
3. In the list of products, click the desired product. Then click **Remove Product**. The product is removed from the **APD Managed** tab.
4. Reimport the template.
   a. Click **Import Template**.
   b. Click **Browse**, then navigate to the template's location and select the template.
   c. Click **Update**. It may take a few seconds for the template to load. Once it has been loaded, the product reappears on the **APD Managed** tab.

5. Regenerate the endpoints.
   a. Click **Generate Product Code**.
   b. In the **Generate Options** section of the **Review Product Elements** screen, choose to generate **System APIs - Code**.
   c. Click **Complete Generation**.

Regenerating endpoints by first removing the generated files

You can trigger endpoint regeneration by removing the generated LOB-specific endpoint files, updating the template, and then generating the endpoints. This method does not require PolicyCenter to be running.

If you generate LOB-specific endpoints with compile errors, and then you stop PolicyCenter, PolicyCenter will not restart until the compile errors are fixed or removed. At this point, the easiest option is usually to remove the generated files.

Complete the following steps to remove generated files and then regenerate endpoints:

1. Search for and remove all files with the following in their names:
   - `<productCode>_gen`
   - `<productCode>_ext`

   These files exist in the following directories under the `modules\configuration` directory:
   - `config\integration\apis\ext\job\v1`
   - `config\integration\apis\ext\policy\v1`
   - `config\integration\apis\ext\policyperiod\v1`
   - `config\integration\mappings\ext\policyperiod\v1`
   - `config\integration\schemas\ext\policyperiod\v1`
   - `config\integration\updaters\ext\policyperiod\v1`
2. Remove the following directory and its contents:
   - `gsrc\gw\rest\ext\pc\policyperiod\<productCode>\v1`
3. Restart PolicyCenter.
4. Verify that the PolicyCenter **Product Design Mode** is set to **Developer**. To view or set the product design mode, go to **Options** > **Preferences**.
5. On the **Product Management** screen, select the product.
6. Select **Update Product** > **From Template**.
7. On the **Load Product** screen, select **Browse** and select the template file. Then, select **Update**.
8. On the **Product Management** screen, select **Generate Product Code**.
9. In the **Generate Options** section of the **Review Product Elements** screen, choose to generate **System APIs - Code**.
10. Select **Complete Generation**.

chapter 38

Toggling between visualized and installed endpoints

Once you generate endpoints for an installed product, you have two sets of endpoints:

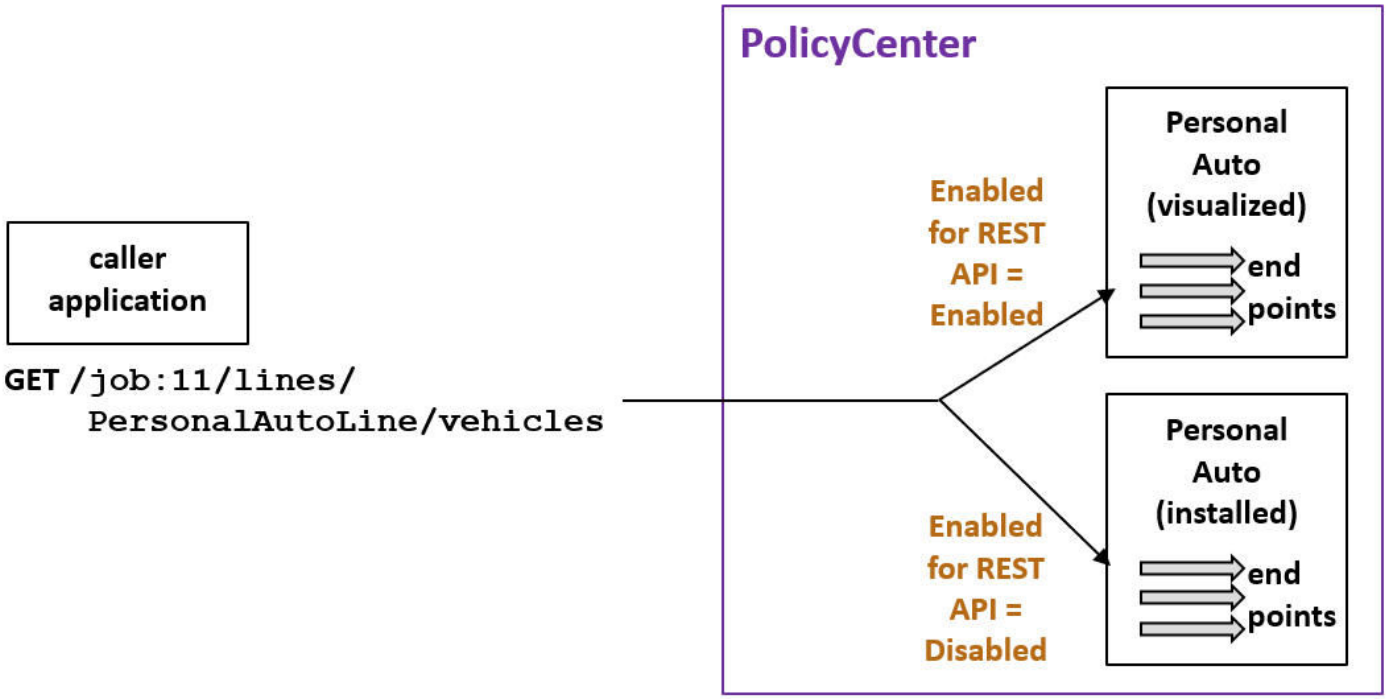
- One set corresponds to the visualized version of the product. These endpoints were created automatically when the product template is imported.
- The other set corresponds to the installed version of the product. These are the endpoints you generated manually.

For a given product, there can only be one set of active endpoints at a given time. The active endpoints are the ones used for incoming calls from a third-party application. For the purposes of testing and development, PolicyCenter gives you the ability to toggle which set is active. This toggle is controlled by the **Enabled for REST API** field on the **APD Managed** tab of the **Product Management** screen.

When there are two sets of endpoints, then:

- Setting **Enabled for REST API** to *Enabled* enables the visualized product's endpoints. This is the default for every product.
- Setting **Enabled for REST API** to *Disabled* enables the installed product's endpoints.

This is depicted in the following diagram:



Which set of endpoints are active?

If a given product exists only as a visualized product or only as an installed product, Cloud API uses the endpoints for that one version of the product. (However, if the only version of the product has been disabled, Cloud API throws an error. For more information on disabling products, see “Disable a visualized product's endpoints” on page 218 and “Disable an installed product's endpoints” on page 218.)

If a given product exists as both a visualized product and an installed product, and both versions have endpoints, then Cloud API checks the product's **Enabled for REST API** flag.

- If the flag is set to *Enabled*, the visualized product is used.
- If the flag is set to *Disabled* (and there is a product access file for the product in the `/installedlobs` directory), the installed product is used.

PolicyCenter also supports the following actions related to visualized and installed products.

Action	More Information
Toggle the active endpoints through the PolicyCenter user interface	“Toggling between visualized and installed endpoints” on page 213
Toggle the active endpoints through Cloud API	“Toggle a product's endpoints” on page 252
Remove a visualized product	“Remove a visualized product” on page 218
Disable an installed product's endpoints	“Disable an installed product's endpoints” on page 218

Toggle a product's active endpoints through the user interface

About this task

You can also toggle a product's endpoints through Cloud API. For more information, see “Toggle a product's endpoints” on page 252.

Procedure

1. On the **Product Management** screen's **APD Managed** tab, select the product.
2. Click **Edit Product**.
3. In the **Enabled for REST API** field, select the appropriate option:
   a. *Enabled* toggles on the visualized product's endpoints.
   b. *Disabled* toggles on the installed product's endpoints.
4. Click **Save**. The change becomes effective immediately. You do not need to restart PolicyCenter.

Results

For more information on toggling a product's endpoints through Cloud API, see “Toggle a product's endpoints” on page 252.

Determining which set of endpoints is active

You can determine which set of endpoints are active by checking the **Enabled for REST API** flag in PolicyCenter. You can also use the following endpoint:

- GET `/productdefinition/v1/products/{productId}`

The response object has a Boolean `visualized` field. When set to true, the visualized endpoints are active. When set to false, the installed endpoints are active. For example, the following request determines which set of endpoints are active for the WorkersComp product.

Command

```
GET /productdefinition/v1/products/WorkersComp
```

Response body

```
{
    "data": {
        "attributes": {
            "abbreviation": "WC",
            "description": "Workers' Compensation",
            "id": "WorkersComp",
            "name": "Workers' Compensation",
            "productType": {
                "code": "Commercial",
                "name": "Commercial"
            },
            "visualized": false
        },
```

The `visualized` field is set to false. Therefore, the installed product's endpoints are active.

chapter 39

Working with products and product templates

The following topic summarizes different actions you can take on a product or product template.

Export a product template from an installed product

About this task

You can export a product template from an installed product. This template can then be imported into PolicyCenter to create a visualized version of the product, which you can use to generate endpoints for the installed product.

Procedure

1. Navigate to the **Administration** tab's **Product Management** screen.
2. Click the **Externally Managed** tab.
3. In the list of **Installed Products**, click the desired product.
4. Click **Extract APD Representation**.

Results

PolicyCenter generates the template and stores it in your `Downloads` directory.

Export a product template from a visualized product

About this task

During the endpoint generation process, you may need to make changes to the visualized version of the product. After doing this, Guidewire recommends that you export an updated version of the product template. This ensures that your product template file has all updates made to the visualized product itself.

Procedure

1. Navigate to the **Administration** tab's **Product Management** screen.
2. Click the **APD Managed** tab.

3. In the list of products, click the desired product.
4. Click **Export Template**.

Results

PolicyCenter generates the template and stores it in your `Downloads` directory.

Disable a visualized product's endpoints

Procedure

1. On the **Product Management** screen's **APD Managed** tab, select the product.
2. Click **Edit Product**.
3. In the **Enabled for REST API** field, select *Disabled*.
4. Click **Save**.

Results

The change becomes effective immediately. You do not need to restart PolicyCenter.

If there is an installed version of the product with installed endpoints, the installed endpoints are used for incoming Cloud API calls for the product. If there is not an installed version of the product with installed endpoints, incoming Cloud API calls for the product will throw an error.

Remove a visualized product

About this task

You may want to remove a visualized product because it is no longer needed, or because you want to reimport an updated version of the product template.

Procedure

1. Navigate to the **Administration** tab's **Product Management** screen.
2. Click the **APD Managed** tab.
3. In the list of products, click the existing visualized product.
4. Click **Remove Product**.

Results

The product is removed from the list of visualized products.

Disable an installed product's endpoints

About this task

You can disable an installed product's endpoints, which prevents the product from being available through Cloud API.

Procedure

1. In the file system, navigate to the `integration/apis/installedlob` directory.
2. Either delete or rename the product access file.
   - This file is named `<productCode>_ext-1.0.yaml`.

3. Restart PolicyCenter.

Removing an installed product's endpoints

You can remove an installed product's endpoints. This may be necessary when you need to regenerate the endpoints but the endpoint files have compilation errors that prevent PolicyCenter from starting.

To do this, search for and remove all files with the following in their names:

- `<productCode>_gen`
- `<productCode>_ext`

These files are place in the following directories in the `modules\configuration` directory:

- `config\integration\apis\ext\job\v1`
- `config\integration\apis\ext\policy\v1`
- `config\integration\apis\ext\policyperiod\v1`
- `config\integration\mappings\ext\policyperiod\v1`
- `config\integration\schemas\ext\policyperiod\v1`
- `config\integration\updaters\ext\policyperiod\v1`

Also, remove the following directory and its contents:

- `gsrc\gw\rest\ext\pc\policyperiod\<productCode>\v1`

chapter 40

Special use cases

Endpoints for pre-Hakuba products

The manner in which LOB-specific endpoints are generated changed in the Hakuba (2023.06.0) release. In prior releases, LOB-specific endpoints were generated using the product code as a prefix, not the line code. This made it difficult for PolicyCenter to determine when multiple products were using the same line.

As of the Hakuba release, LOB-specific endpoints are generated using the line code as a prefix. Guidewire has the following recommendations for insurers with products whose LOB-specific endpoints were generated in a pre-Hakuba release.

Controlling the level at which the endpoints are generated

Starting in Hakuba, there is an application configuration parameter named `RestAPIsGeneratingByPolicyLine`. It is a Boolean parameter that controls the level at which endpoints are generated.

- `true` - PolicyCenter generates endpoints at the line level.
  - Endpoints are prefixed by the line code.
- `false` - PolicyCenter generates endpoints at the product level.
  - Endpoints are prefixed at the product code.

The default for this parameter is `true`. However, when an insurer moves from a pre-Hakuba release to Hakuba (or later), this parameter is set to `false` to preserve the original product-prefix-style generation.

Guidewire recommends all insurers eventually set this parameter to `true`, as this setting supports multi-line products. Insurers may need to modify existing products to ensure they continue to behave as expected.

WARNING: Before you change the `RestAPIsGeneratingByPolicyLine` parameter's value from `false` to `true`, Guidewire strongly recommends that you ensure that for every mono-line products whose product abbreviation is different than the line prefix, you have updated the product as specified in “Products whose line prefix is not identical to the product prefix” on page 222. If you do not, then any future regeneration of the LOB-specific endpoints will create a new set of endpoints, as opposed to replacing the existing endpoints. This is likely to lead to unexpected behaviors.

Products whose line prefix is identical to the product prefix

If you have a mono-line product whose line has a prefix that is identical to the product prefix, no additional work is needed. The new behavior will be extracting the prefix from a different location (the line instead of the product). But if the prefix itself is the same, the LOB-specific endpoints generated for the line will also be the same.

Products whose line prefix is not identical to the product prefix

If you have a mono-line product whose line has a prefix that is not identical to the product prefix, then you must take additional steps to move the product to the new line-prefix-style generation.

The following instructions use the following abbreviations:

- `<productcode>` = the product abbreviation, all in lowercase letters
- `<linecode>` = the line prefix, all in lowercase letters

To modify the product, execute the following steps in a development environment with the current product.

1. Set the `RestAPIsGeneratingByPolicyLine` application configuration parameter to `true`.
2. In `config/integration`:
   - Rename all of the files that contain the product code so that `_<productcode>_` is changed to `_<linecode>_`.
   - Within all of the file, change any reference of `_<productcode>_` to `_<linecode>_`.
3. In `config/integration/apis/installedlobs`:
   - Rename `<productcode>_ext-1.0.yaml` to `<linecode>_ext-1.0.yaml`,
4. In `gsrc/gw/rest/ext/pc/job/<productcode>/v1`:
   a. Change the name of the directory path so that it is `gsrc/gw/rest/ext/pc/job/<linecode>/v1`.
   b. Rename `Job<product-abbreviation>ExtApiHandler.gs` to `Job<linecode>ExtApiHandler.gs`.
5. In `gsrc/gw/rest/ext/pc/policy/<productcode>/v1`:
   a. Change the name of the directory path so that it is `gsrc/gw/rest/ext/pc/policy/<linecode>/v1`.
   b. Rename `Policy<product-abbreviation>ExtApiHandler.gs` to `Policy<linecode>ExtApiHandler.gs`.
6. Drop the PolicyCenter database.
7. Restart PolicyCenter
8. Regenerate the LOB-specific endpoints for the product.
   - For more information, see “Generating LOB-specific endpoints”.

You can then include the product along with the other updated resources as part of the regular update process.

Endpoints for scheduled items in SBT products

Sometimes, an account holder wants a policy to cover a specific item that has unusual value, such as a piece of fine art or a diamond necklace. There is a generic coverage on the policy that could apply to the item. But because of the item's unusual value, it ought to be covered by its own coverage. The insurer may require additional information, such as the item's value and how this value was determined. The item's coverage may also require a deductible or limit that is outside the range of what is allowed by the more generic coverage. These types of "unusual value" items appear on the policy in a list that is called a *schedule*. Thus, the items themselves are called *scheduled items*.

The original Product Designer, Advanced Product Designer, and Schedule-Based Templates (SBTs) all have the ability to define products with scheduled items. However, the underlying mechanism for defining scheduled items in SBTs is significantly different than that of Product Designer and Advanced Product Designer. Therefore, when you generate LOB-specific endpoints for SBT products with schedule items, PolicyCenter needs additional code to determine how to generate the endpoints. This code is defined in the `ComplexSchedulePlugin` plugin.

Implementing the ComplexSchedulePlugin plugin

The `ComplexSchedulePlugin` plugin is a part of the base configuration as of the Garmisch release (2023.02.0). The manner in which this plugin is implemented varies based on the insurer's original version of PolicyCenter and the original version of the SBT.

Development starts on Garmisch (or later)

If the development of the product started on a Garmisch (or later) instance of PolicyCenter using a Garmisch (or later) instance of the SBT, then there is an implementation of the `ComplexSchedulePlugin` plugin included with the base configuration. No additional work is required.

Updating both a pre-Garmisch PolicyCenter and pre-Garmisch SBT to Garmisch (or later)

If the development of the product started on a pre-Garmisch instance of PolicyCenter using a pre-Garmisch instance of the SBT, and both PolicyCenter and the SBT are updated to Garmisch (or later), then no additional work is required. An implementation of the `ComplexSchedulePlugin` plugin will be added to PolicyCenter during the update process.

Updating a pre-Garmisch PolicyCenter to Garmisch (or later) while using a pre-Garmisch SBT

If the development of the product started on a pre-Garmisch instance of PolicyCenter using a pre-Garmisch instance of the SBT, and only PolicyCenter is updated to Garmisch (or later), then some additional work is required to make the pre-Garmisch SBT compatible with Garmisch PolicyCenter. This additional work consists of the following:

1. Download a Garmisch (or later) version of the SBT.
2. Copy the following files from the Garmisch (or later) version of the SBT to the existing pre-Garmisch SBT

File location	File name
`/config/plugin/registry`	`ComplexSchedulePlugin.gwp`
`/gsrc/gw/lob/common/rest/productdefinition/v1/coverage`	`SbtScheduledItemPropertyWrapper.gs`
`/gsrc/gw/plugin/rest/v1/impl/`	`SbtComplexSchedulePlugin.gs`

3. In the Garmisch (or later) SBT, find the file from the list below that matches the SBT you are implementing. Replace the pre-Garmisch version of this file in your SBT with the Garmisch (or later) version from the downloaded SBT.

SBT	File location	File name
SBT-BP7	`/gsrc/gw/lob/bp7`	`BP7BusinessOwnersLineScheduleEnhancement.gsx`
SBT-CA7	`/gsrc/gw/lob/ca7`	`CA7CommAutoLineScheduleEnhancement.gsx`
SBT-CP7	`/gsrc/gw/lob/cp7`	`CP7PropertyLineScheduleEnhancement.gsx`
SBT-CR7	`/gsrc/gw/lob/cr7`	`CR7CrimeLineScheduleEnhancement.gsx`
SBT-GL7	`/gsrc/gw/lob/gl7`	`GL7GeneralLiabilityLineScheduleEnhancement.gsx`
SBT-WCM	`/gsrc/gw/lob/wcm`	`WCMWorkersCompLineScheduleEnhancement.gsx`

Note: The structure of SBTs has changed significantly over time. If you are working with an SBT that was released in 2015 or earlier, Guidewire recommends contacting your Guidewire representative to discuss the best way to implement your SBT on a Garmisch or later release.

Differences between "product designer" scheduled items and SBT scheduled items

Once there is a suitable plugin implementation, the endpoints for SBT scheduled items work almost entirely the same as scheduled items in Product Designer/Advanced Product Designer products. There are two differences with endpoints for scheduled items in SBT products.

- They have two new scheduled item property types: `additionalInsured` and `option`. These properties have corresponding value properties on the `ScheduledItemProperty` schema definition.
- They can have a set of child `/coverage` endpoints. This is because scheduled items in an SBT product can have coverages attached to them. When the `ComplexSchedulePlugin` plugin identifies that a given coverable has at least one coverage that has scheduled items with child coverages, it generates the child /coverage endpoints. For

example, for an SBT product, there could be an endpoint such as `.../buildings/{buildingId}/coverages/{coverageId}/scheduled-items/{scheduledItemId}/coverages/{scheduledItemCoverageId}`.

Endpoints for multi-line products

A *multi-line product* is a product that has multiple lines of business. If the product is exposed to Cloud API, then each line has its own set of LOB-specific endpoints. Every line in a multi-line product could be used by only one multi-line product, or it could be used by multiple products, each of which could be mono-line or multi-line.

This topic discusses the way in which PolicyCenter generates LOB-specific endpoints for multi-line products.

Products that started development prior to Hakuba (2023.06.0)

The manner in which LOB-specific endpoints are generated changed in the Hakuba (2023.06.0) release. If you developed or installed any products on a release prior to Hakuba, there are additional considerations for the pre-Hakuba products. For more information, see “Products whose line prefix is not identical to the product prefix” on page 222..

LOB-endpoint generation is initiated at the product level, but it is executed at the line level

When you generate LOB-specific endpoints, you initiate the generation at the product level.

However, by default, PolicyCenter generates LOB-specific endpoints at the line level. When the endpoints are generated, the endpoints are prefixed using the line code. If you generate LOB-specific endpoints for a multi-line product, the endpoints for each line will have their own prefix.

> **Note:** Guidewire recommends that all insurers who entered production on or after the Hakuba release (2023.06.0) generate endpoints at the line level. Insurers who entered production prior to the Hakuba release may want to consider generating endpoints at the product level. For more information, see “Products whose line prefix is not identical to the product prefix” on page 222..

Lines shared across products

Lines that are used by multiple products must have the same line code, such as `ba` for a business auto line. This ensures that the same set of endpoints are used by products that share a given line.

If a given line is used by multiple products and you make a change to that line, you only need to regenerate the endpoints for any one of the products using this line. For example, suppose you have an Inland Marine line that is used by an Inland Marine product and a Commercial Package product. After generating endpoints for both products, you make a change to the Inland Marine line. You then regenerate the endpoints for the Inland Marine product. You do not also need to regenerate the endpoints for the Commercial Package product. The Commercial Package product will automatically have access to the regenerated Inland Marine line endpoints.

If a single line of business is used by multiple products, then either all of those products must be in visualized mode or all of them must be in installed mode. You cannot have a single line of business used by two or more products where some are in visualized mode and some are in installed mode.

Cloud API does not require a mono-line product for every LOB

Cloud API supports the ability to generate LOB-specific endpoints for a line of business, even when that line is used only by multi-line products. For example, suppose you have a Commercial Property Line that is used only by a multi-line Commercial Package product. When you generate LOB-specific endpoints for the Commercial Package product, Cloud API generates endpoints for the Commercial Property Line, even though this line is used only by multi-line products.

Other services or environments may require that you create a mono-line product for every line. For example, Guidewire APD Service may require you to create a mono-line Commercial Property package for the Commercial Property Line before you can add that line to a multi-line Commercial Package product. But this requirement is not enforced by Cloud API. Cloud API can generate endpoints for the Commercial Property Line using only the Commercial Property package.

The RestAPIsGeneratingByPolicyLine config parameter

There is an application configuration parameter named `RestAPIsGeneratingByPolicyLine`. It is a Boolean parameter that controls the level at which endpoints are generated.

- `true` - PolicyCenter generates endpoints at the line level.
  - Endpoints are prefixed by the line code.
  - This is the default.
- `false` - PolicyCenter generates endpoints at the product level.
  - Endpoints are prefixed at the product code.

Guidewire recommends against setting this parameter to `true` unless you are working with one or more products whose LOB-specific endpoints were generated in an instance of PolicyCenter prior to the Hakuba release (2023.06.0). For more information, see “Products whose line prefix is not identical to the product prefix” on page 222.

chapter 41

Generating endpoints for the Personal Auto product

The following topic describes how to generate LOB-specific endpoints for the Personal Auto product found in the base configuration. It also describes how to test the generated endpoints.

For information on how to generate LOB-specific endpoints for products in general, see “Generating LOB-specific endpoints”.

The base configuration Personal Auto product

To generate endpoints for the Personal Auto line of business, you must:

1. Generate an APD template from the existing product.
2. Modify the template to align naming in the original Cloud API version of the product.
3. From APD, install the system API code.
4. Set the product configuration to use the installed resources.
5. Modify the `VehicleDriverExtResource` class.
6. Restart PolicyCenter.

Step 1: Generate an APD template from the existing product

First, you must generate an APD template from the existing Personal Auto product. This is done through the PolicyCenter user interface.

1. If necessary, set **Options** > **Preferences** > **Product Design Mode** to *Developer*, and then click **Update**.
2. Select **Administration** > **Product Management**.
3. Click the **Enternally Managed** tab. This pane contains a list of products that have been installed by means other than APD.
4. In the **Installed Products** pane, select "Personal Auto".
5. In the **Details** pane, click **Extract APD Representation**. PolicyCenter generates the APD template and stores it in the `<USER_HOME>/Downloads` directory.

Step 2: Modify the template

The Personal Auto product predates Cloud API. Therefore, there are a small number of names in the product that do not align with Cloud API naming. You must modify the template to bring these values into alignment.

1. Open the template generated in the previous step.
2. Find the following tag and change it to the following value. (There is only one instance of this tag.)
   a. Original value: `<TypeName>VehicleDriver</TypeName>`
   b. Modified value: `<TypeName>Driver</TypeName>`
3. Find the following tag and change it to the following value. (There is only one instance of this tag.)
   a. Original value: `<TypeName>PersonalVehicle</TypeName>`
   b. Modified value: `<TypeName>Vehicle</TypeName>`
4. Save the template.

Step 3: Install the system API code

Next, you must import the modified template into Advanced Product Designer and then install the "system API code".

Note: In this step, you are installing the system API code only. Do not install the entire product. Installing the entire product could overwrite existing logic, which would result in the product behaving in unexpected ways.

1. On the **Product Management** screen, click the **APD Managed** tab.
2. Click **Import from Template**.
3. Click **Browse**. Select the template to import.
4. Click **Update**. APD imports the product template. Once the import is complete, the product is listed on the **APD Managed** tab.
5. Select the imported template from the list.
6. On the **Details** tab below the list, select **Generate Product Code**. APD shows the **Review Product Elements** screen.
   - You may see warnings about naming conflicts for the NumAddInsured and PatternCode fields. You can ignore these warnings.
7. Under **Generate Options**, ensure **Default Directory** is selected for **System APIS - Code**. Set all other options to **Do Not Generate**.
8. Click **Complete Generation**. In the confirmation dialog box, click **OK**.

Step 4: Set the product configuration to use the installed resources

A given product can have two sets of artifacts:

- *Visualized artifacts* are created when a product is imported from a template or mind map.
- *Finalized artifacts* are created when a product is installed.

Calls from Cloud API can use only one set of artifacts. Every product has an **Enabled for REST API** flag. When a single product has both sets of artifacts, this setting determines which set to use for incoming Cloud API calls.

- If the flag is set to *Enabled*, the visualized set is used.
- If the flag is set to *Disabled*, the finalized set is used.

By default, this flag is set to Enabled. This means that, by default, the Personal Auto line will use the artifacts generated when the template was imported in the previous step. To make incoming calls use the original installed artifacts, you must disable this flag.

1. If you have not done so already, set **Options** > **Preferences** > **Product Design Mode** to *Developer*, and then click **Update**.

2. Navigate to the **Product Management** screen.
3. Select the product. In the **Details** tab below the list of products, click **Edit Product**. PolicyCenter shows the **Product Definition** screen for that product.
4. Set **Enabled for REST API** to *Disabled*.
5. Click **Save**.

This setting does not take effect until PolicyCenter is restarted, which is done in the final step.

Step 5: Modify the VehicleDriverExtResource class

During system API code installation, APD generates several artifacts for the new product. For the base configuration Personal Auto product, one of these artifacts is the `VehicleDriverExtResource` class, located in the `gw.rest.ext.pc.policyperiod.pa.v1.driver` package.

The initial version of this class is empty. Guidewire recommends replacing the class with the code provided below for the following reasons.

- The override of the `finishCreate` function ensures that the vehicle driver is linked to the parent `PALine` object. This is necessary for certain behaviors, such as some of the base configuration Personal Auto screens in the user interface.
- The override of the `applyPatchForCreate` function is not required, but it is recommended to ensure there are no duplicate driver values.

```
package gw.rest.ext.pc.policyperiod.pa.v1.driver

uses gw.api.modules.rest.framework.v1.batch.BatchUpdateMap
uses gw.api.modules.rest.framework.v1.exceptions.LocalizedExceptionUtil
uses gw.api.modules.rest.framework.v1.json.DataEnvelope
uses gw.rest.core.pc.policyperiod.v1.JsonConstants#DATA_ATTRIBUTES
uses gw.rest.core.pc.policyperiod.v1.JsonConstants.VehicleDriver#POLICY_DRIVER

@Export
class VehicleDriverExtResource extends VehicleDriverGenResource {
  override function finishCreate(data : DataEnvelope, batchUpdateMap : BatchUpdateMap) {
    super.finishCreate(data, batchUpdateMap)

    var driver = this.Driver.PolicyDriver
    driver.PersonalAutoLine = this.Parent.Vehicle.PALine
  }

  override function applyPatchForCreate(data : DataEnvelope, batchUpdateMap : BatchUpdateMap) {
    super.applyPatchForCreate(data, batchUpdateMap)

    // Ensure there are no duplicate PolicyDriver values
    var pcr = Driver.PolicyDriver
    var duplicate = Parent.Vehicle.Drivers.firstWhere(\d -> d.PolicyDriver == pcr && d != Driver)
    if (duplicate != null) {
      throw LocalizedExceptionUtil.badInputInBody({DATA_ATTRIBUTES, POLICY_DRIVER}.join("."),
          "Rest.PolicyPeriod.V1.PolicyContactJsonValueResolver.Duplicate",
          {duplicate.PolicyDriver.AccountContactRole.AccountContact.Contact.RestId, "driver", "vehicle"})
    }
  }
}
```

Step 6: Restart PolicyCenter

To finish the deployment of all new resources, you must restart PolicyCenter.

Generic endpoints in Personal Auto submissions

The Personal Auto product, as designed in the base configuration, stores some auto-specific information in resources that are not LOB-specific.

For example, the following auto-specific fields exist on the `PolicyContact` resource and are accessed using the generic `/jobs/{jobId}/contacts` endpoints:

- `applicableGoodDriverDiscount`
- `dateCompletedTrainingClass`

- `goodDriverDiscount`
- `licenseNumber`
- `licenseState`
- `numberOfAccidents`
- `numberOfViolations`
- `policyNumberOfAccidents`
- `policyNumberOfViolations`
- `trainingClassType`
- `yearLicensed`

Therefore, when modifying a base configuration Personal Auto submission, you need to use a mix of LOB-specific endpoints and generic endpoints.

Composite request submission example for Personal Auto

You can test your work by executing the following composite request. This is an end-to-end test that does the following:

1. Creates a new account.
2. Create a Personal Auto submission for the account.
3. Answers the question "Is the applicant currently insured" with "No - New Driver" (`newdriver`).
4. Adds a line-level optional coverage (`PALossOfUseCov`).
5. Adds a vehicle.
6. Adds an optional coverage (`PARentalCov`) to the vehicle, and specifies the 20 dollars/day for 60 days (`60/20`) coverage term.
7. Specifies contact information that is required for quoting (such as date of birth and number of accidents).
8. Specifies vehicle driver information (percent of time the contact driver the vehicle).
9. Sets the Anti-Lock Brakes Discount modifier to true.
10. Quotes the job.

```
{
  "requests": [
    {
      "method": "post",
      "uri": "/account/v1/accounts",
      "body": {
        "data": {
          "attributes": {
            "initialAccountHolder": {
              "contactSubtype": "Person",
              "firstName": "Tamsin",
              "lastName": "Tester",
              "primaryAddress": {
                "addressLine1": "2850 S. Delaware St.",
                "city": "San Mateo",
                "postalCode": "94403",
                "state": {
                  "code": "CA"
                }
              }
            },
            "initialPrimaryLocation": {
              "addressLine1": "2850 S. Delaware St.",
              "city": "San Mateo",
              "postalCode": "94403",
              "state": {
                "code": "CA"
              }
            },
            "producerCodes": [
              {
```

```
              "id": "pc:16"
            }
          ],
          "organizationType": {
            "code": "other"
          }
        }
      }
    },
    "vars": [
      {
        "name": "accountId",
        "path": "$.data.attributes.id"
      },
      {
        "name": "driverId",
        "path": "$.data.attributes.accountHolder.id"
      }
    ]
  },
  {
    "method": "post",
    "uri": "/job/v1/submissions",
    "body": {
      "data": {
        "attributes": {
          "account": {
            "id": "${accountId}"
          },
          "baseState": {
            "code": "CA"
          },
          "jobEffectiveDate": "2022-08-01",
          "producerCode": {
            "id": "pc:16"
          },
          "product": {
            "id": "PersonalAuto"
          }
        }
      }
    },
    "vars": [
      {
        "name": "jobId",
        "path": "$.data.attributes.id"
      }
    ]
  },
  {
    "method": "patch",
    "uri": "/job/v1/jobs/${jobId}/questions",
    "body": {
      "data": {
        "attributes": {
          "answers": {
            "PACurrentlyInsured": {
                "choiceValue": {
                  "code": "newdriver"
                }
            }
          }
        }
      }
    }
  },
  {
    "method": "post",
    "uri": "/job/v1/jobs/${jobId}/lines/PersonalAutoLine/coverages",
    "body": {
      "data": {
        "attributes": {
          "pattern": {
            "id": "PALossOfUseCov"
          }
        }
      }
    }
  },
  {
    "method": "post",
    "uri": "/job/v1/jobs/${jobId}/lines/PersonalAutoLine/vehicles",
    "body": {
      "data": {
        "attributes": {
            "make": "Toyota",
            "model": "Camry",
            "modelYear": 2010,
            "costNew": {
                "amount": "33000",
```

```
                "currency": "usd"
            },
            "licenseState": {
                "code": "CA"
            },
            "vin": "14HEW8RLGMDSP03AA"
        }
      }
    },
    "vars": [
      {
        "name": "vehicleId",
        "path": "$.data.attributes.id"
      }
    ]
  },
  {
    "method": "post",
    "uri": "/job/v1/jobs/${jobId}/lines/PersonalAutoLine/vehicles/${vehicleId}/coverages",
    "body": {
      "data": {
        "attributes": {
          "pattern": {
            "id": "PARentalCov"
          },
          "terms": {
            "PARental": {
              "choiceValue": {
                "code": "60/20"
              }
            }
          }
        }
      }
    }
  },
  {
    "method": "patch",
    "uri": "/job/v1/jobs/${jobId}/contacts/${driverId}",
    "body": {
      "data": {
        "attributes": {
            "dateOfBirth": "1980-10-10",
            "licenseNumber": "CA7732839",
            "licenseState": {
                "code": "CA"
            },
            "numberOfAccidents": {
                "code": "0"
            },
            "numberOfViolations": {
                "code": "0"
            },
            "policyNumberOfAccidents": {
                "code": "0"
            },
            "policyNumberOfViolations": {
                "code": "0"
            }
        }
      }
    }
  },
  {
    "method": "post",
    "uri": "/job/v1/jobs/${jobId}/lines/PersonalAutoLine/vehicles/${vehicleId}/drivers",
    "body": {
      "data": {
        "attributes": {
            "percentageDriven": 100,
            "policyDriver": {
                "id": "${driverId}"
            }
        }
      }
    }
  },
  {
    "method": "patch",
    "uri": "/job/v1/jobs/${jobId}/lines/PersonalAutoLine/vehicles/${vehicleId}/modifiers/PAAntiLockBrakes",
    "body": {
      "data": {
        "attributes": {
          "booleanModifier": true
        }
      }
    }
  },
  {
    "method": "post",
```

```
      "uri": "/job/v1/jobs/${jobId}/quote"
    }
  ]
}
```

chapter 42

Codegen config files

Cloud API has endpoints that third-party applications can use to interact with policies. This includes retrieving information about policies, creating policies, and modifying policies.

The base configuration contains endpoints for LOB-generic policy contents, such as policy contacts and policy locations. But, it does not initially contain endpoints for LOB-specific policy contents, such as coverables and coverages. In order to work with LOB-specific contents through Cloud API, you must first generate LOB-specific endpoints for each line.

A *codegen config file* is a file that provides PolicyCenter with information about how to generate LOB-specific endpoints. Codegen config files can provide the following types of information:

- Mapping for fields whose names in the visualized product are different than the names in the installed product
- Overrides for certain behaviors that exist in the product but are not reflected in the template or the visualized product

This topic describes how to use a codegen config file to influence LOB-specific endpoint generation.

Codegen config file location and names

File location

Codegen config files are stored in the `modules\configuration\config\integration\apis\installedlobs` directory.

File names

Codegen config files are named using the following convention:

```
<lob_prefix>_codegen_config_ext-1.0.yaml
```

The lob-prefix is the code for the line that is found in the `InstalledPolicyLine` typelist, lowercased. For example, suppose you have a Personal Auto product whose lob prefix is "pa". The codegen config file for this product is `pa_codegen_config_ext-1.0.yaml`.

Base configuration files

The base configuration contains a set of predefined codegen configure files. Some of the files are for products that were a part of the base configuration in a previous release. Other files are for products installed through Standards Based Templates (SBTs). You can use these files to modify how the endpoints are generated. They can also be used as examples for when you need to create your own files.

Files for custom products

If there is no codegen config file for a given product, and you need to specify information in a codegen config file, then you must create the file manually. The file name must match the following pattern:

```
<lob_prefix>_codegen_config_ext-1.0.yaml
```

For example, suppose you created a Crime product using Product Designer. The LOBs prefix is "cr". You want to generate endpoints for this product and you need to specify codegen config information. The file for this product would be: `cr_codegen_config_ext-1.0.yaml`.

Codegen config file syntax

Codegen config files can have up to two top-level properties: a `types` property and a `wizardStepIds` property. All of the information related to LOB-specific endpoints is under the `types` property.

The types property

The visualized version of a product contains a set of APD types. In this context, an *APD type* is a line (such as `PALine`), a coverable (such as `PAVehicle`), or some other type of object that stores information relevant to the policy (such as `PADriver`).

Codegen config files can have a top-level `types` property followed by one or more APD types. For example, the base configuration codegen config file for Personal Auto includes the following:

```
types:
  PADriver:
    ...
  PALine:
    ...
  PAPolicyDriverMVR:
    ...
  PAVehicle:
    ...
```

For each type, and for fields in the type, you can specify one or more overrides.

Overrides at the type level

The following properties can be listed after a type under the `types` property.

autonumber

For collection types, this specifies the field to use when sorting members of the collection. This overrides the collection's default sorting, if any.

Syntax

```
autonumber: <fieldName>
```

Examples from a Personal Auto codegen config file

For example, in the base configuration Personal Auto product, every `PAVehicle` on a given policy is assigned a sequential integer value in the `VehicleNumber` field. When a collection of `PAVehicles` is retrieved, the collection ought to be sorted based on these sequential integers. To specify this, the codegen config file contains the following:

```
types:
  PAVehicle:
    autonumber: VehicleNumber
```

canDelete, canPatch, canPost, and canSplit

By default, PolicyCenter creates endpoints that let callers DELETE, PATCH, and POST instances of each type. Where relevant, callers can also split instances of the type.

The following properties disable the DELETE method, the PATCH method, the POST method, and/or the POST `/split` business action endpoint for this type. You can specify any or all of these in any combination.

Syntax

```
canDelete: false
canPatch: false
canPost: false
canSplit: false
```

Examples from a Workers' Compensation codegen config file

```
types:
  WCEmployee:
    canSplit: false
  WCJurisdiction:
    canDelete: false
    canPatch: false
    canPost: false
```

identifier

In most cases, a type's `id` field maps to the data model entity's `PublicID` field. In the base configuration, the values of these take the form of "pc:<alphanumeric-string>", such as "`pc:Sp2FxL1061-Q_W8Bvtkj5`".

For some types, the `id` value comes from a field other than `PublicID`. For example, in Commercial Auto, IDs for the `CABAJurisdiction` type come from the `State` field. Instances of this type do not have ID values that are `PublicID` values, such as "`pc:Sp2FxL1061-Q_W8Bvtkj5`". Rather, their ID values are codes from the `State` typelist, such as "`CA`", "`NY`", or "`WY`".

In a codegen config file, you can use the `identifier` property to specify a data model entity field other than `PublicID` that the `id` property must map to.

Syntax

```
identifier: <fieldName>
```

Examples from a Commercial Auto codegen config file

```
types:
  CABAJurisdiction:
    identifier: State
```

nameOverride

By default, the name of a type is mapped to the same name in the installed product. The `nameOverride` property specifies a different name in the installed product to map the visualized product's type to. The primary use cases for nameOverride are the following:

- PolicyLine entities that do not follow the <LinePrefix>Line naming convention expected by APD.
- Coverable or exposure entities that don’t start with the exact line prefix. For example, the PersonalVehicle entity in the base configuration PersonalAuto product does not start with PA (see example below).
- Coverable or exposure entities that have names that aren’t legal in APD. The most common reason for this is names that are too long.

Syntax

```
nameOverride: <installedProductTypeName>
```

Examples from a Personal Auto codegen config file

For example, in the base configuration, the APD type `PALine` corresponds to the installed product's `PersonalAutoLine`. To resolve this mismatch, the codegen config file contains the following:

```
types:
  PALine:
    nameOverride: PersonalAutoLine
```

You can also specify name overrides at the field level. For more information, see “Overrides at the fields level” on page 239.

oneToOne

A one-to-one relationship is a relationship that two data model entities have. One acts as the parent, and the other as the child. For the parent, each instance may have an association with up to one instance of the child entity. For the child, each instance must be associated with exactly one parent.

Every APD type maps to a data model entity. In some cases, an APD type maps to an entity that is the child in a one-to-one relationship. In the codegen config file, you can specify that an APD type is a one-to-one child by using the `oneToOne` property. This must be set to the name of the property on the parent data model entity that points to the data model child.

Syntax

```
oneToOne: <propertyOnDataModelParentThatPointsToDataModelChild>
```

Note that the codegen config process considers the "parent data model entity" to be the data model entity that is referenced by the parent APD type of the given APD type. For example, the `IMSignPart` APD type has a parent named `IMLine`. The `IMLine` APD type maps to the `InlandMarineLine` data model entity. Therefore, if you specify a `oneToOne` property on the `IMSignPart` APD type, the codegen config process looks for the property referenced by the `oneToOne` on the `InlandMarineLine` data model entity.

Examples from an Inland Marine codegen config file

For example, for the Inland Marine product, the `InlandMarineLine` and `IMSignPart` data model entities have a one-to-one relationship. Every `InlandMarineLine` instance can have at most one `IMSignPart` instance, and every `IMSignPart` instance must be attached to an `InlandMarineLine` instance.

To define this relationship for the purposes of codegen config, you can add the `oneToOne` property to the child APD type (`IMSignPart`). It must identify the name of the property on the parent data model entity (`InlandMarineLine`) that points to the data model child. In this case, the name of the property is also `IMSignPart`. The codegen config file looks like this:

```
type:
  IMSignPart:
    oneToOne: IMSignPart
```

toCreateAndAdd

By default, whenever an instance of a type is added to a line, PolicyCenter simply creates a new instance of the type and adds it to the line. No additional properties are set on the instance and no additional actions are taken. However, for some types, additional actions may be required. Typically, these actions are executed by a method declared in the `<LineName>Enhancement` Gosu enhancement.

For example, when a vehicle is added to a Personal Auto line, the following additional actions are needed:

- The vehicle's type must default to Passenger/Light Truck.
- The garage location must default to the first available PolicyLocation.
- Coverages, conditions, and exclusions for the vehicle must be created.
- The vehicle must be given a vehicle number based on the number of other vehicles on the line.

These actions are executed by a `createAndAddVehicle` method on the `PersonalAutoLineEnhancement` Gosu enhancement.

When the codegen config process generates LOB-specific endpoints, the default behavior for each type is to simply create a new instance of the type and add it to the line. If there is a special enhancement method that you want to use instead, you can specify this method using the `toCreateAndAdd` property in the codegen config file.

Syntax

```
toCreateAndAdd: <MethodNameFromTheGosuLineEnhancement>
```

Examples from a Personal Auto codegen config file

```
types:
  PAVehicle:
    toCreateAndAdd: createAndAddVehicle
```

toRemove

By default, whenever an instance of a type is removed from a line, PolicyCenter simply removes the instance from the line. No additional actions are taken. However, for some types, additional actions may be required. Typically, these actions are executed by a method declared on the `<LineName>Enhancement` Gosu enhancement.

For example, when a vehicle is removed from a Personal Auto line, the following additional actions are needed:

- The remaining vehicles must be renumbered to prevent any gaps in vehicle numbering.

These actions are executed by a `removeVehicle` method on the `PersonalAutoLineEnhancement` Gosu enhancement.

When the codegen config process generates LOB-specific endpoints, the default behavior for each type is to simply remove the instance. If there is a special enhancement method that you want to use instead, you can specify this method using the `toRemove` property in the codegen config file.

Syntax

```
toRemove: <MethodNameFromTheGosuLineEnhancement>
```

Examples from a Personal Auto codegen config file

```
types:
  PAVehicle:
    toRemove: removeVehicle
```

resourceName

This overrides the default resource name. Use this override when the resource name differs from the `nameOverride` name.

Syntax

```
resourceName: <resourceName>
```

Examples from a Workers' Compensation codegen config file

```
types:
  WCEmployee:
    nameOverride: WCCoveredEmployeeBase
    resourceName: WCCoveredEmployee
```

Overrides at the fields level

The following properties can be listed after the `fields` property for a given type.

createOnly

By default, the value for a field can be specified in both POSTs and PATCHes. To restrict a field to being specified in POSTs only, you can set the field's `createOnly` property to true.

Syntax

```
createOnly: true
```

Examples from a Workers' Compensation codegen config file

In a Workers' Compensation product, the `WCEmployee` type has two fields that can be specified when the `WCEmployee` is created, but not when it is PATCHed. These fields are `Location` and `SpecialCov`. To enforce this, the codegen config file contains the following.

```
types:
  WCEmployee:
    fields:
      Location:
        createOnly: true
      SpecialCov:
        createOnly: true
```

getterProperty

Defines the getter method in the mapper if it is different from the default property getter for the field.

Syntax

```
getterProperty: <getterProperty>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassResourceField:
        customType: Resource
        mapperHandling:
          getterProperty: RestV1_RiskClassResourceField
```

ignored

This prevents the field from being exposed to Cloud API.

Syntax

```
ignored: true
```

Examples from a Workers' Compensation codegen config file

```
types:
  WCWaiverOfSubro:
    fields:
      IfAnyExposure:
        ignored: true
      NumEmployees:
        ignored: true
```

mapperHandling

Provides parent level settings for mapper handling.

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassResourceField:
        customType: Resource
        mapperHandling:
          getterProperty: RestV1_RiskClassResourceField
```

nameOverride

Use nameOverride when the name in the APD model doesn’t match the name in the entity. This can happen when the entity’s field name isn’t allowed in APD so it had to be changed in the APD model. Field names that aren't allowed in

APD include names that are too long, start with a lowercase letter, or are reserved words in APD such as “Type” or “Use.” Another common use case is to map an APD model field name that was changed or truncated back to the field name on the installed product

Syntax

```
nameOverride: <installedProductFieldName>
```

Examples from a Personal Auto codegen config file

For example, in the base configuration, the `PAVehicle` type's `Year` field corresponds to the installed product `PersonalVehicle` coverable's `ModelYear` field. What this example is doing is mapping the APD field name back to the entity model.

```
types:
  PAVehicle:
    fields:
      Year:
        nameOverride: ModelYear
```

You can use `nameOverride` to resolve naming conflicts. For example, if an LOB has a field and an exposure with the same name, this can cause unexpected results. Use `nameOverride` to override the schema field name to resolve these types of conflicts.

You can also specify name overrides at the type level. For more information, see “Overrides at the type level” on page 236.

The `nameOverride` property should be used only for *Cloud Retrofit without APD App*. For *Cloud Retrofit with APD App* or APD conversion, use `schemaPropertyName`.

nullable

This prevents the caller from specifying a null value for the field in a request object.

Syntax

```
nullable: false
```

Examples from a Workers' Compensation codegen config file

```
types:
  WCEmployee:
    fields:
      IfAnyExposure:
        nullable: false
```

readOnly

By default, the value for a field can be specified in POSTs and PATCHes. To prevent a field from being specified in either case, you can set the field's `readOnly` property to true.

Syntax

```
readOnly: true
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      LineField:
        # do not generate property in updater
        readOnly: true
```

requiredForCreate

This overrides the default `requiredForCreate` value.

- Setting this to true prevents the caller from omitting the value in a POST request.
- Setting this to false allows the caller to omit the value in a POST request.

Syntax

```
requiredForCreate: <Boolean>
```

Examples from a Personal Auto codegen config file

```
types:
  PAVehicle:
    fields:
      GarageLocation:
        requiredForCreate: false
```

schemaHandling

This property provides a parent level for settings related to schema handling.

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassResourceField:
        schemaHandling:
          schemaPropertyName: anotherRiskClassResourceField
```

schemaPropertyName

Use schemaPropertyName in the following cases:

- The field name conflicts with a graph property name generated for an array of child coverables and exposures. You need to rename the property to ensure the names are unique.
- The field name conflicts with some other built-in property, such as the address properties automatically added to any location-based coverable.
- You don’t like the name in the APD model and would prefer a different name in the schema (for example, because the APD name has constraints that schema names don’t, so maybe you want a longer name).

Syntax

```
schemaPropertyName: <propertyname>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassResourceField:
        schemaHandling:
          schemaPropertyName: anotherRiskClassResourceField
```

setterProperty

Use this property for the setter method in the updater when it is different from the default property setter for the field.

Syntax

```
setterProperty: <propertyID>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassResourceField:
```

```
        updaterHandling:
          setterProperty: RestV1_RiskClassResourceField
```

updaterHandling

The parent level settings for updater handling.

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassResourceField:
        updaterHandling:
          setterProperty: RestV1_RiskClassResourceField
```

Overrides at the fields level - APD conversion EA

IMPORTANT: This functionality is available only to customers who have signed up for our Early Access (EA) program. Talk to your Guidewire representative to learn more about our eligibility criteria for EA programs. Note that EA capabilities may or may not become part of our future offerings.

The following properties can be listed after the `fields` property for a given type. The properties in this section are for use only with the APD "Other" field type.

create

A create expression for the updater.

Syntax

```
create: <create-expression>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassRefField:
        updaterHandling:
          create: SampleTestCoverable.RestV1_DefaultRiskClass()
```

customType

As of this release, possible values are `Resource` and `ClassCode`.

When `Resource` is specified, the following default behavior occurs:

- The schema will use only resourceType configs
- The mapper will have a `ResourceReference` and try to evaluate the property as `xxx.RestV1_AsEffDatedReference` (if the other field is an EffDated entity), or `xxx.RestV1_AsReference` (if the other field is not an EffDated entity but it is a KeyableBean entity)

When `ClassCode` is specified, the following default behavior occurs (where XXX is the product line prefix, such as PA or WC):

- An empty `XXXClassCodeReference` schema, mapper, and updater are created for the field.
- A default `XXXClassCodeJsonValueResolver` is created in the updater. The resolver will attempt to resolve the class code by ID.
- The default path for the resolver is `gw.rest.core.pc.policyperiod.v1.XXXClassCodeJsonValueResolver`. You can override this path using the `valueTypeResolver` property described below.

Syntax

```
customType: <customType>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      ClassCodeField:
        customType: ClassCode
      LineField:
        customType: Resource
```

extensions

Additional settings defined for the `x-gw-extensions` schema property, such as `filterable` and `sortable` properties.

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassStringField:
        schemaHandling:
          extensions:
            filterable: false
            sortable: false
```

format

This defines the format settings for the field. This field needs to be overridden only in cases where the format is not supported in APD and you want to make the field a scalar type in the schema.

Syntax

```
format: "string"
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      LineField:
        format: date-time
```

mapperRef

Use this override when the mapper reference is different from the `schemaDefinition`.

Syntax

```
mapperRef: <mapperRef>
```

Example from a test codegen config file

```
IndustryCodeField:
  schemaDefinition: ReferenceTableEntry
  mapperHandling:
      mapperRef: IndustryCode
```

resourceType

Defines the `resourceType` settings for the field. This is applied when `customType` is set as `Resource`. If the `customType` is `Resource` and this field is not defined, the system will try to use the actual entity type as the `resourceType`.

Syntax

```
resourceType: <resourceType>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassResourceField:
        customType: Resource
        schemaHandling:
          resourceType: RiskClass
```

schemaDefinition

For use when the field is a reference property. If the referenced schema is found in the core schema, no additional schema is created. If it's not found in the core schema, an empty schema will be generated so it can be extended with an extension file.

Syntax

```
schemaDefinition: <schemaDefinition>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      IndustryCodeField:
        schemaDefinition: ReferenceTableEntry
```

type

The type settings for the field. This needs to be overridden only in cases where the field is not an APD supported format but you want to make it a scalar type in the schema.

Syntax

```
type: <datatype>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassStringField:
        schemaHandling:
          type: string
```

updaterRef

Use this override when the updater reference is different from the `schemaDefinition`.

Syntax

```
updaterRef: <updater-reference>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassRefField:
        schemaDefinition: RiskClassReference
        updaterHandling:
          # using different updater from schemaDefinition
          updaterRef: RiskClassReferenceForUpdater
```

valueTypeResolver

The full path of the JSON value resolver for the field if it is referencing another entity.

Syntax

```
valueTypeResolver: <full-path-to-resolver>
```

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassResourceField:
        updaterHandling:
          valueTypeResolver: gw.rest.core.pc.policyperiod.v1.location.RiskClassJsonValueResolver
          valueTypeResolverConfigs:
            setById: true
```

valueTypeResolverConfigs

Available to provide additional optional configs for the JSON value resolver.

Example from a test codegen config file

```
types:
  SampleTestCoverable:
    fields:
      RiskClassResourceField:
        updaterHandling:
          valueTypeResolver: gw.rest.core.pc.policyperiod.v1.location.RiskClassJsonValueResolver
          valueTypeResolverConfigs:
            setById: true
```

The wizardStepIds override

A codegen config file can also have a `wizardStepIds` element. This element is used to support highlighting of fields in the PolicyCenter user interface for circumstances when the field's value triggers a warning or error from the validation rules.

This element does not have an impact on the behavior of LOB-specific endpoints.

Example codegen config file

The following is the codegen config file in the base configuration for the Personal Auto product. It has multiple type-level and field-level overrides. There are additional examples for other products in the `modules\configuration\config\integration\apis\installedlobs` directory.

```
types:
  PADriver:
    nameOverride: VehicleDriver
    fields:
      PolicyDriver:
        createOnly: true
  PALine:
    nameOverride: PersonalAutoLine
  PAPolicyDriverMVR:
    nameOverride: PolicyDriverMVR
    # PolicyDriverMVRs are managed implicitly via requests to retrieve
    #   an MVR on individual drivers and are read-only
    canDelete: false
    canPatch: false
    canPost: false
  PAVehicle:
    nameOverride: PersonalVehicle
    fields:
      GarageLocation:
        requiredForCreate: false
      Year:
        nameOverride: ModelYear
    autonumber: VehicleNumber
    toCreateAndAdd: createAndAddVehicle
```

```
    toRemove: removeVehicle
wizardStepIds: false
```

chapter 43

Endpoints for managing product templates

The Product Definition API contains endpoints that can be used to act on product templates and products. For example, you can do the following through these endpoints:

- Import a product template
- Generate LOB-specific endpoints for an installed product based on the current visualized product
- Toggle which set of endpoints (visualized or installed) are active

These endpoints may simplify the work to develop and test products. But, these endpoints are not intended to execute product development on their own. For initial product development, Guidewire recommends using APD App directly.

Product templates

Within the context of product development, a product can be in one of two states:

- **Visualized** - The product exists in Advanced Product Designer in a "draft" state.
  - Some product-specific artifacts (such as the product-specific *coverages* database table) have not yet been created.
- **Installed** - The product exists in a "finalized" state.
  - All product-specific artifacts (including the product-specific *coverages* database table) have been created.

Several Product Definition API templates use the `producttemplate` resource. In this context, a *product template* is a JSON representation of a visualized product. This could be a product that is APD-native or one that was created by importing a template that was extracted from an installed product.

Querying for visualized products

Use the following endpoints for query for product templates for existing visualized products:

- GET `/productdefinition/v1/product-templates`
- GET `/productdefinition/v1/product-templates/{productId}`

For example, the following request retrieves the product template for the visualized version of a Workers' Compensation product (whose id is `WorkersComp`).

Command

```
GET /productdefinition/v1/product-templates/WorkersComp
```

Response payload

```
{
    "data": {
        "attributes": {
            "abbreviation": "WC",
            "codeIdentifier": "WorkersComp",
            "description": "Workers' Compensation",
            "enabled": true,
            "id": "WorkersComp",
            "name": "Workers' Compensation",
            "productAccountType": {
                "code": "Any",
                "name": "Any"
            }
        },
        ...
```

Importing products

Cloud API has endpoints that let you import a product. The source of the product can be either an XML template or a mind map.

Use the following endpoint to import a product:

- POST `/productdefinition/v1/import-template`
- POST `/productdefinition/v1/import-xmind`

For most Cloud API endpoints, the request has a body with a single string of JSON text. However, the import endpoints expect the request body to be a `FormData` object, as opposed to a JSON string. The request header must have a content key whose value is the template or mind map to import. The response contains a body with a single `id` attribute that identifies the `id` of the imported product.

Example of importing an XML template

For example, suppose you had an XML template for a `PersonalAuto` product. The template is named `PA-20230516.xml`. The following request imports the template.

Command

```
POST /productdefinition/v1/import-template
```

Request header

```
content/PA-20230516.xml
```

Request body

```
{
    "data": {
        "attributes": {
            "id": "PersonalAuto"
        }
    }
}
```

Example of importing a mind map

For example, suppose you had a mind map for a `MarineCargo` product. The template is named `MarineCargo.xmind`. The following request imports the mind map.

Command

```
POST /productdefinition/v1/import-xmind
```

Request header

```
content/MarineCargo.xmind
```

Request body

```
{
    "data": {
        "attributes": {
            "id": "MarineCargo"
        }
    }
}
```

Import XML templates and mind maps using Postman

About this task

From Postman, you can POST XML templates and mind maps using FormData objects.

Procedure

1. In Postman, start a new request by clicking the + to the right of the Launchpad tab.
2. Under the Untitled Request label, select POST.
3. In the **Enter request URL** field, enter the URL for the server and the endpoint. For example:
   a. To POST an XML template to an instance of PolicyCenter on your machine, enter: `http://localhost:8180/pc/rest/productdefinition/v1/import-template`
   b. To POST a mind map to an instance of PolicyCenter on your machine, enter: `http://localhost:8180/pc/rest/productdefinition/v1/import-xmind`
4. On the **Authorization** tab, specify authorization information as appropriate.
5. Specify the request payload.
   a. In the first row of tabs (the one that starts with Params), click **Body**.
   b. In the row of radio buttons, select *form-data*.
   c. On the first line, for KEY, enter: content
   d. Click outside of the content cell. Then, mouse over the right side of the cell. A drop-down list appears. Change the value from *Text* to *File*.
   e. For VALUE, click the Select Files button and navigate to XML product file or mind map file.
6. Click Send. The response payload appears below the request payload.

Generating an installed product from a visualized product

Use the following endpoint to generate an installed product from a visualized product:

- POST `/productdefinition/v1/product-templates/{productId}/codegen`

The request body must specify the ID of the product.

The codegen process can be executed in one of four modes:

- `API_CODE`: Generate only the LOB-specific endpoints
- `BASE_CODE`: Generate only the product base code
- `EXTERNAL`: Generate only the additional registered extensions
- `ALL`: Generate all of the above. This is the default

For more information on the scope of the product base code or additional registered extensions, see *Integrating Products with PolicyCenter*.

For example, the following request generates only the LOB-specific endpoints for a Crime product whose ID is "Crime".

Command

```
POST /productdefinition/v1/product-templates/{productId}/codegen
```

Request body

```
{
    "data": {
        "attributes": {
            "productId": "Crime"
            "generationMode": "API_CODE"
        }
    }
}
```

Toggle a product's endpoints

A given product can exist at the same time as a visualized version and an installed version. Each version can have its own LOB-specific endpoints. However, only one version of the product can be active at a given time.

You can toggle the active set of endpoints through the user interface. For more information, see “Toggle a product's active endpoints through the user interface” on page 214.

You can also use the following endpoints to toggle the active set of endpoints:

- POST `/productdefinition/v1/product-templates/{productId}/enable`
  - Makes the visualized product's endpoints active
- POST `/productdefinition/v1/product-templates/{productId}/disable`
  - Makes the installed product's endpoints active

The request does not require a body. The response contains information about the product template, including an `enabled` field, which is set to either `true` (the visualized product's endpoints are active) or `false` (the installed product's endpoints are active).

For example, for the Workers' Compensation product (`WorkersComp`), the following request makes the installed version's endpoints active.

Command

```
/productdefinition/v1/product-templates/WorkersComp/disable
```

Request body

<none>

Response body

```
{
    "data": {
        "attributes": {
            "abbreviation": "WC",
            "codeIdentifier": "WorkersComp",
            "description": "Workers' Compensation",
            "enabled": false,
            ...
        },
```

Determining which endpoints are active

You can use the following endpoints to determine whether REST API endpoints are active for a particular product or product version:

- GET `/productdefinition/v1/products`
- GET `/productdefinition/v1/products/{productId}`

Included in the response for these endpoints are two properties that specify which product endpoints are active:

- `visualized`: Boolean value specifying whether the product is visualized (true) or installed (false).
- `restAPIsEnabledAndActive`: Boolean value specifying whether the product endpoints are enabled for that version of the product.

The following table describes the various states a product can be in and the flag settings used to determine each state.

visualized	restAPIsEnabledAndActive	Product state
true	false	• The product is a visualized product • REST API endpoints are not enabled
true	true	• The product is a visualized product • REST API endpoints are enabled
false	false	• The product is an installed product • REST API endpoints are not installed, or • REST API endpoints are disabled
false	true	• The product is an installed product • REST API endpoints are enabled

For each product, it's possible to have both a visualized and an installed version of the product. However, the REST APIs can be enabled for only one of those versions.

In this example, there are two products listed. One is the visualized version of product TST Product, the other is the installed version of the same product. In this example, the REST APIs for the installed version of the product are enabled.

Command

```
GET /productdefinition/v1/products
```

Response

```
{
  "count": 2,
  "data": [
    {
      "attributes": {
        …
        "description": "TST Product",
        "descriptionKey": "Product_TST.Description",
        "id": "TST",
        "name": "TST Product",
        "nameKey": "Product_TST.Name",
        …
        "restAPIsEnabledAndActive": false,
        "visualized": true
      },
      …
    },
    {
      "attributes": {
        …
        "description": "TST Product",
```

```
        "descriptionKey": "Product_TST.Description",
        "id": "TST",
        "name": "TST Product",
        "nameKey": "Product_TST.Name",
        …
        "restAPIsEnabledAndActive": true,
        "visualized": false
      },
    …
  ]
```

Working with product editions

A *product edition* defines product model properties and subclause relationships for a product line. Product editions can be used to introduce changes to the product after a product has gone into production. A product edition is packaged as an APD template and can be imported, provided the base product is already installed.

To activate the product edition, a caller can submit a POST request to the `/productdefinition/v1/lines/{lineId}/activate-editions` endpoint. The request body must contain a `lineId` property containing the product edition identifier:

```
{
  "data": {
    "attributes": {
      "lineId": "Crime-2"
    }
  }
}
```

See *Cloud API Consumer Guide* for additional information on product editions.

Removing a visualized product

Use the following endpoint to remove a visualized product:

- DELETE `/productdefinition/v1/product-templates/{productId}`

For example, the following request deletes a visualized Workers' Compensation product.

Command

```
DELETE /productdefinition/v1/product-templates/WorkersComp
```

Request body

```
<no body>
```

part 7

Configuration for other specific use cases

The following topics discuss configuration for additional specific uses cases. This includes:

- Configuring custom batch processes
- Configuring address locales
- REST preload requests

chapter 44

Configuring batch processes

Cloud API supports the ability to start batch processes. For information on how to use the `/systemtools/v1/batch-processes/{batchProcessType}/start` endpoint, see the *Cloud API Consumer Guide*.

Some batch processes let you specify arguments when you run the batch process. These arguments perform the function of input parameters, and they can change the way the batch process runs.

You can create custom batch processes which take arguments. If you want to be able to submit arguments to the `/start` endpoint, then some configuration of Cloud API is required. This topic describes this configuration.

Configuring Cloud API to support custom batch process arguments

InsuranceSuite applications do not have a uniform format for arguments for batch processes. Some batch processes may expect strings, while others may expect custom POJOs (Plain Old Java Objects). There is no general way for Cloud API to support batch process arguments for all batch process types. Therefore, if you have implemented a custom batch process that uses arguments and you want to start that batch process through Cloud API with arguments, you must configure Cloud API to handle the arguments.

This configuration requires two steps:

- Adding a schema property to the `BatchProcessArguments` schema.
- Adding handling code to the `BatchProcessExtResource` class to convert the payload into arguments

Configuring the BatchProcessArguments schema

The `BatchProcessArguments` schema defines the information that can be included as arguments with a call to the `/systemtools/v1/batch-processes/{batchProcessType}/start` endpoint. For more information on general schema configuration, see “Endpoint architecture” on page 15.

To configure a custom batch process for arguments, you must add a property to this schema.

- The name of the property must be the same as the code of the `BatchProcessType` typecode, ignoring case. Guidewire recommends using all lowercase letters.
- The type of this property depends on the underlying argument type that the batch process expects. Guidewire recommends that the property be an object with well-defined property names that describe what the arguments conceptually represent.

The schema extension must be declared in the `systemtools_ext-1.0.schema.json` schema file.

For example, suppose you created a custom batch process named GroupMetrics_Ext that calculates performance metrics for every group based on work assigned to users in the group. It takes an optional argument of a single group name, specified as a string. To implement this argument, the `systemtools_ext-1.0.schema.json` schema file would include the following:

```
"definitions": {
  "BatchProcessArguments": {
    "properties": {
      "groupmetrics_ext": {
        "title": "GroupMetrics_Ext",
        "description": "Arguments for the GroupMetrics_Ext batch process",
        "$ref": "#/definitions/GroupMetrics_ExtArguments"
      }
    }
  },
  "GroupMetrics_ExtArguments": {
    "title": "GroupMetrics_ExtArguments",
    "description": "Arguments for the GroupMetrics_Ext batch process",
    "type": "object",
    "properties": {
      "groupName": {
        "title": "GroupName",
        "description": "The name of the group to process",
        "type": "string"
      }
    }
  }
}
```

Configuring the BatchProcessExtResource class

The `BatchProcessExtResource` resource class is a Gosu file that defines required behaviors for working with batch process elements.

To configure a custom batch process for arguments, you must add handling code to the `populateCustomBatchProcessArgumentsArray` method in the `BatchProcessExtResource` resource class to convert the payload into arguments. Override the method (if it has not already been overridden) and add an `if` statement for this batch process type that inspects the JSON object, extracts the relevant data, and puts it into an array.

Following on from the previous example, suppose you are configuring the `BatchProcessExtResource` resource class to implement the `GroupName` argument for the GroupMetrics_Ext custom batch process. The `populateCustomBatchProcessArgumentsArray` method would look like this:

```
protected override function populateCustomBatchProcessArgumentsArray
                                  (json : JsonObject) : Serializable[] {
   var result : Serializable[]
   if (this.Element.getType() == BatchProcessType.TC_GroupMetrics_Ext) {
     result = {json.getString("groupName")}
   } else if ( /* other batch process type handling goes here */ ) {
     ...
   }
   return result
 }
```

chapter 45

Configuring address locales

The fields in an address that are either valid or required can vary based on the country or region that the address belongs to. For example, when designating a specific region in a country:

- A US address uses *state*
- A Canadian address uses *province*
- A Japanese address uses *prefecture*

The base configuration of Cloud API comes with locale settings for a wide range of countries. you may need to configure the base configuration locales, and you may need to add new locales.

Properties in the Address schema

In Cloud API, addresses are managed by the `Address` schema. There are two properties used in the schema to address locale-specific issues.

- The schema itself has a `discriminatorProperty` attribute. This attribute defines the field in each address that can be used to determine which fields are required or valid. In the base configuration, this property is set to `country`. In other words, it is the country of an address that determines which other fields are valid or required.
- Every property in the schema can have a `countryRestricted` Boolean property set to true. This marks the property as being restricted to certain countries.

Note that the Address schema does not identify the required and valid fields for each country. This information is declared in the `addresses.i18n.yaml` file.

The Address schema

The `Address` schema is defined in `common_pl-1.0.schema.json`. The following is a portion of the schema. Note the following:

- The `Address` itself has the `discriminatorProperty` set to `country`.
- The `addressLine1` property is valid for all locales. Therefore, there is no `countryRestricted` attribute.
- The `province` property is not valid for all locales. Therefore, there is an `x-gw-extensions.countryRestricted` attribute and it is set to true.

```
"Address": {
  "title": "Address",
  "description": "An `Address` represents a postal address. The fields available on an `Address`
                  will depend upon the `country` value for the `Address`.",
  "type": "object",
```

```
  "x-gw-extensions": {
    "discriminatorProperty": "country"
  },
  "properties": {
    "addressLine1": {
      "title": "Address line 1",
      "description": "The first line of the address",
      "type": "string",
      "x-gw-nullable": true
    },
    ...
    "province": {
      "title": "Province",
      "description": "The province of the address. Only applicable in certain countries.",
      "$ref": "#/definitions/TypeKeyReference",
      "x-gw-nullable": true,
      "x-gw-extensions": {
        "countryRestricted": true,
        "typelist": "State"
      }
    },
```

The Address mapping

The schema mapping is defined in `common_pl-1.0.mapping.json`. The following is a portion of the mapping file. Note that if a property is not available for all locales, it has an additional `predicate` attribute set to `Address.RestV1_isFieldAvailable('<propertyName>')`.

```
"Address": {
  "schemaDefinition": "Address",
  "root": "entity.Address",
  "properties": {
    "addressLine1": {
      "path": "Address.AddressLine1"
    },
  "province": {
    "path": "Address.State",
    "mapper": "#/mappers/TypeKeyReference",
    "predicate": "Address.RestV1_isFieldAvailable('province')"
  },
```

The Address updaters

The schema updater is defined in `common_pl-1.0.updater.json`. The following is a portion of the updater file. Note that if a property is not available for all locales, it has an additional `allowed` attribute set to `Address.RestV1_validateInputField('<propertyName>', srcJson)`.

```
"Address": {
  "schemaDefinition": "Address",
  "root": "entity.Address",
  ...
  },
  "properties": {
    "addressLine1": {
      "path": "Address.AddressLine1"
    },
  "province": {
    "path": "Address.State",
    "allowed": "Address.RestV1_validateInputField('province', srcJson)",
    "valueResolver": {
      "typeName": "TypeKeyValueResolver"
    }
  },
```

Properties in the addresses.i18n.yaml file

The `addresses.i18n.yaml` file provides information about the required and valid properties for an address based on its country. Every country has its own listing with the following information:

- The country code, as defined in the `Country` typelist
- The name of the country
- The valid fields for an address in that country
- The required fields for an address in that country

The following is a portion of the `addresses.il8n.yaml` file. Note the following:

- For Canada:
  - The `province` property is valid and required.
  - The `addressLine2` field is valid, but not required.
- For Japan:
  - The `prefecture` property is valid and required.
  - The `addressLine1Kanji` field is valid, but not required.
- For the United States:
  - The `state` property is valid and required.
  - The `addressLine2` field is valid, but not required.

```
countries:
  CA:
    name: Canada
    addressFields: addressLine1, addressLine2, addressLine3, city, county, province, postalCode
  addressRequire: addressLine1, city, province, postalCode
  JP:
    name: Japan
    addressFields: addressLine1, addressLine1Kanji, addressLine2, addressLine2Kanji, addressLine3, city, cityKanji,
prefecture, postalCode
    addressRequire: addressLine1, city, prefecture, postalCode
US:
    name: United States
    addressFields: addressLine1, addressLine2, addressLine3, city, county, state, postalCode
    addressRequire: addressLine1, city, state, postalCode
```

Configuration tasks

Schema extensions for the Address schema must be declared in the following files:

- The Common API schema extension file: `common_ext-1.0.schema.json`
- The Common API mapping extension file: `common_ext-1.0.mapping.json`
- The Common API updater extension file: `common_ext-1.0.updater.json`

For detailed information on how to configure schemas, see “Endpoint architecture” on page 15.

Properties that are valid for all locales

If you want to add a custom `Address` property that is valid for all locales:

1. Add the field to the `Address` data model entity.
2. In the Common API schema extension file, add a property for the field to the `Address` schema.
3. In the Common API mapping file, add mapping information for the property.
4. In the Common API updater file, add updater information for the property.
5. Add the field to the `addressFields` section of every country entries in the `addresses.il8n.yaml` file.

Properties that are valid for some locales

If you want to add a custom `Address` property that is valid for only some locales:

1. Add the field to the `Address` data model entity.
2. In the Common API schema extension file, add a property for the field to the `Address` schema.
   - Set the property's `x-gw-extensions.countryRestricted` attribute to true.
3. In the Common API mapping file, add mapping information for the property.
   - Add a `predicate` attribute set to `Address.RestV1_isFieldAvailable('<propertyName>')`.

4. In the Common API updater file, add updater information for the property.
   - Add an allowed attribute set to `Address.RestV1_validateInputField('<propertyName>', srcJson)`
5. Add the field to the `addressFields` section of the appropriate country entries in the `addresses.il8n.yaml` file.

Properties that are valid and required for some locales

If you want to add a custom `Address` property that is valid for only some locales:

1. Add the field to the `Address` data model entity.
2. In the Common API schema extension file, add a property for the field to the `Address` schema.
   - Set the property's `x-gw-extensions.countryRestricted` attribute to true.
3. In the Common API mapping file, add mapping information for the property.
   - Add a `predicate` attribute set to `Address.RestV1_isFieldAvailable('<propertyName>')`.
4. In the Common API updater file, add updater information for the property.
   - Add an allowed attribute set to `Address.RestV1_validateInputField('<propertyName>', srcJson)`
5. Add the field to the `addressFields` section and the `addressRequire` section of the appropriate country entries in the `addresses.il8n.yaml` file.

chapter 46

REST preload requests

When an InsuranceSuite server is first started, initial REST API requests may be slow to execute. Guidewire allows you to define "preload" requests for REST APIs to ensure that initial API requests from external callers are processed quickly.

Preload requests are REST API calls that are executed automatically after server startup to warm up endpoints. These calls are treated similarly to regular API calls, triggering loading of configuration files, just-in-time compilation, cache loading, and so on. However, they are made by the InsuranceSuite application itself and not by an external caller.

This mechanism is primarily intended for use with Cloud API endpoints, but it can also be used with REST endpoints built directly on the InsuranceSuite REST Framework.

This mechanism complements the REST servlet preload option, which is controlled by the `PreloadRestServletConfig` configuration parameter. This configuration parameter must be set to `true` for preload requests to be executed.

For more on warming up endpoints, see the *Application Guide*.

Defining preload requests

Preload requests can be defined by adding JSON files in the `configuration/config/integration/preload` directory or any of its subdirectories. You must create the `preload` directory in the correct place for the application to execute the calls.

Each JSON file holds a single preload request object. Preload request objects define a single API call. Preload requests are executed in alphabetical order based on the name of the JSON file.

In each preload request object, there are two required properties:

Field	Data type	Description
`method`	String	The HTTP method for the subrequest, such as `GET` or `POST`. The string is not case sensitive.
`path`	String	The path for the request, relative to the servlet root (for example, `/admin/v1/users`).

For example, the following JSON file defines a minimal preload request for GETing the claim graph schema:

```
{
  "path": "/claim/v1/graph-schema",
  "method": "get"
}
```

There are also optional properties that define call behavior, such as body, description, and headers. The full schema for preload request objects is found at `configuration/config/integration/schemas/gw/core/pl/preload/preload_pl-1.0.schema.json`.

Note: The `PreloadRestServletConfig` parameter must be set to `true` for preload requests to execute.

Preload requests can make use of the Composite API to execute a sequence of related requests. This is useful to ensure that all endpoints for a specific workflow have been warmed up. For example, the following preload request performs all actions required to create and quote a personal auto policy in PolicyCenter, starting with creating an account.

```
{
  "path":"/composite/v1/composite",
  "method":"POST",
  "body":{
    "requests":[
      {
        "method":"post",
        "uri":"/account/v1/accounts",
        "body":{
          "data":{
            "attributes":{
              "initialAccountHolder":{
                "contactSubtype":"Person",
                "firstName":"Tamsin",
                "lastName":"Tester",
                "primaryAddress":{
                  "addressLine1":"2850 S. Delaware St.",
                  "city":"San Mateo",
                  "postalCode":"94403",
                  "state":{
                    "code":"CA"
                  }
                }
              },
              "initialPrimaryLocation":{
                "addressLine1":"2850 S. Delaware St.",
                "city":"San Mateo",
                "postalCode":"94403",
                "state":{
                  "code":"CA"
                }
              },
              "producerCodes":[
                {
                  "id":"pc:16"
                }
              ],
              "organizationType":{
                "code":"other"
              }
            }
          }
        },
        "vars":[
          {
            "name":"accountId",
            "path":"$.data.attributes.id"
          },
          {
            "name":"driverId",
            "path":"$.data.attributes.accountHolder.id"
          }
        ]
      },
      {
        "method":"post",
        "uri":"/job/v1/submissions",
        "body":{
          "data":{
            "attributes":{
              "account":{
                "id":"${accountId}"
              },
              "baseState":{
                "code":"CA"
              },
              "jobEffectiveDate":"2022-08-01",
              "producerCode":{
                "id":"pc:16"
              },
              "product":{
                "id":"PersonalAuto"
              }
            }
          }
        },
```

```
      "vars":[
        {
          "name":"jobId",
          "path":"$.data.attributes.id"
        }
      ]
    },
    {
      "method":"patch",
      "uri":"/job/v1/jobs/${jobId}/questions",
      "body":{
        "data":{
          "attributes":{
            "answers":{
              "PACurrentlyInsured":{
                "choiceValue":{
                  "code":"newdriver"
                }
              }
            }
          }
        }
      }
    },
    {
      "method":"post",
      "uri":"/job/v1/jobs/${jobId}/lines/PersonalAutoLine/coverages",
      "body":{
        "data":{
          "attributes":{
            "pattern":{
              "id":"PALossOfUseCov"
            }
          }
        }
      }
    },
    {
      "method":"post",
      "uri":"/job/v1/jobs/${jobId}/lines/PersonalAutoLine/vehicles",
      "body":{
        "data":{
          "attributes":{
            "make":"Toyota",
            "model":"Camry",
            "modelYear":2010,
            "costNew":{
              "amount":"33000",
              "currency":"usd"
            },
            "licenseState":{
              "code":"CA"
            },
            "vin":"14HEW8RLGMDSP03AA"
          }
        }
      },
      "vars":[
        {
          "name":"vehicleId",
          "path":"$.data.attributes.id"
        }
      ]
    },
    {
      "method":"post",
      "uri":"/job/v1/jobs/${jobId}/lines/PersonalAutoLine/vehicles/${vehicleId}/coverages",
      "body":{
        "data":{
          "attributes":{
            "pattern":{
              "id":"PARentalCov"
            },
            "terms":{
              "PARental":{
                "choiceValue":{
                  "code":"60/20"
                }
              }
            }
          }
        }
      }
    },
    {
      "method":"patch",
      "uri":"/job/v1/jobs/${jobId}/contacts/${driverId}",
      "body":{
        "data":{
          "attributes":{
```

```
              "dateOfBirth":"1980-10-10",
              "licenseNumber":"CA7732839",
              "licenseState":{
                "code":"CA"
              },
              "numberOfAccidents":{
                "code":"0"
              },
              "numberOfViolations":{
                "code":"0"
              },
              "policyNumberOfAccidents":{
                "code":"0"
              },
              "policyNumberOfViolations":{
                "code":"0"
              }
            }
          }
        }
      },
      {
        "method":"post",
        "uri":"/job/v1/jobs/${jobId}/lines/PersonalAutoLine/vehicles/${vehicleId}/drivers",
        "body":{
          "data":{
            "attributes":{
              "percentageDriven":100,
              "policyDriver":{
                "id":"${driverId}"
              }
            }
          }
        }
      },
      {
        "method":"patch",
        "uri":"/job/v1/jobs/${jobId}/lines/PersonalAutoLine/vehicles/${vehicleId}/modifiers/PAAntiLockBrakes",
        "body":{
          "data":{
            "attributes":{
              "booleanModifier":true
            }
          }
        }
      },
      {
        "method":"post",
        "uri":"/job/v1/jobs/${jobId}/quote"
      }
    ]
  }
}
```

Behavior of preload requests

Preload requests are processed similarly to requests from caller applications.

However, data in preload POST and PATCH requests is not committed to the database. Cloud API preload requests include the `GW-DoNotCommit` header by default, which ensures that the data is not committed. For custom REST APIs, a custom header needs to be included to prevent the data from being committed.

Note that you might need to implement additional logic to prevent preload requests from triggering downstream systems, if necessary.

Authentication for preload requests

If no authentication information is provided in the preload request object, preload requests are executed using the credentials of the unrestricted user `su`. Optionally, you can use the `asUser` field in the preload request object to specify the id of another user as the caller. Use the public id of the user, which is returned in the `id` field when retrieving a user through Cloud API.

Authentication plugins `RestAuthenticationSourceCreatorPlugin` and `RestAuthenticationServicePlugin` handle authentication specially for preload requests. These plugins must be in use for preload requests to be properly executed.

Logging preload requests

Preload requests do not trigger normal logging and observability calls. Instead, they are logged only with INFO messages to the console if the request is completed successfully:

```
INFO Preload request '<name of preload request>’ completed successfully
```

If a request fails, an error message is logged to the console with the HTTP status code.

part 8

Choosing an authentication flow

Endpoints within Cloud API must control access to the data and actions within PolicyCenter. When a caller tries to access data or execute an action, the caller must be authenticated and authorized. *Authentication* is the process of verifying that the caller is who they claim to be. *Authorization* is the process of determining what operations and data the caller is allowed to access. These two process are often referred to collectively as "auth".

The following topics provide an overview of the different aspects an insurer must consider when planning an authentication approach. This includes:

- The different types of callers that Cloud API supports
- The different applications involved with Cloud API authentication
- The types of access enforced by Cloud API
- The supported authentication methods

This concludes with a topic that can help insurers determine which authentication flows are most appropriate for a given caller application.

chapter 47

Overview of authentication

Cloud API must control access to the data and actions within PolicyCenter. When a caller tries to access data or execute an action, the caller must be authenticated and authorized. *Authentication* is the process of verifying that the caller is who they claim to be. *Authorization* is the process of determining what operations and data the caller is allowed to access. These two process are often referred to collectively as "auth".

This topic provides an overview of how authentication and authorization are managed by Cloud API.

Types of callers

Within the context of Cloud API authentication, a *caller* is a user or service who triggers a Cloud API call from a caller application.

There are several different types of callers. This documentation uses the following terms to identify them:

- **Internal user** - This is a person who is listed as a user in the PolicyCenter operational database. For example, Alice Applegate, a PolicyCenter underwriter, is an internal user.
  - Note that internal users can use caller applications and trigger Cloud API calls from those applications. For example, suppose there is a location photography portal that contains pictures of covered buildings taken by a third-party field agent. An underwriter reviews and selects pictures to be saved to PolicyCenter. This action triggers a Cloud API call by an internal user from a caller application.
- **External user** - This is a person who is not listed as a user in the PolicyCenter operational database. For PolicyCenter, there are three typical types of external user:
  - **Account holders** - Users who want to interact with information about their accounts and policies. For example, Ray Newton, who is a policyholder and wants to verify what coverages he has.
  - **Producers** - Users who are producers and want to edit or view accounts, jobs, or policies for a policyholder. For example, Karen Egerston, who is a producer and wants to update the policy for Ray Newton.
  - **Anonymous user** - This is a person who is not yet known to the insurer but who may establish a business relationship with the insurer. When an unauthenticated user creates an account, a token for an anonymous account holder is created. The anonymous account holder can use that token to create, modify, and quote and bind a submission. Once the submission is bound, the anonymous account holder can authenticate as a regular account holder.
- **Service** - This is a service, also referred to as a *service-to-service application*. For example, a billing service that processes premium payments and periodically reports to PolicyCenter when a policy is delinquent. There are several ways in which a service can make a call:

    - As a **standalone service**, in which the service executes the call as itself. It does not execute the call on behalf of a specific person or through a PolicyCenter user account.
    - As a **service with user context**, in which the service presents information about itself and about a specific user. The call is able to do only the things that both the service by itself could do and the user by itself could do.
    - As a **service with service account mapping**, in which the service is mapped to an account in the PolicyCenter database and has access as determined by that account.
- **Unauthenticated caller** - This is a user or service who provides no authentication information. Unauthenticated callers can access metadata endpoints. Unauthenticated callers are typically callers who need information about Cloud API endpoints. When Cloud API authentication for anonymous users is configured, unauthenticated callers can also create accounts.

Within the context of authentication and authorization, this documentation uses the following terms in the following way:

- *User* is used exclusively for callers that are people.
- *Service* is used exclusively for callers that are not people and that take action without direct action from a person.
- *Service account* is used to refer to an account in the PolicyCenter database that is used exclusively by a service and that defines access for that service.
- *Caller* is used to collectively refer to users and services.

Authentication architecture

The authentication architecture for Cloud API consists of:

- The InsuranceSuite application (such as PolicyCenter)
- Guidewire Identity Federation Hub
- The insurer's identity provider (IdP)
- Any additional authorization application that stores caller-specific authorization information
- A set of one or more caller applications

Note that some parts of the architecture are relevant for all Cloud API calls, regardless of the type of caller. Other parts of the architecture are relevant only for certain types of callers.

Guidewire Hub

Guidewire Identity Federation Hub (Guidewire Hub) is the trusted auth server for all Guidewire cloud applications, including caller applications that insurers create to access Guidewire cloud resources. Guidewire Hub uses OAuth 2.0 and SAML for identity management services.

The primary responsibilities of Guidewire Hub are:

- For internal users and external users:
    - To receive authentication requests from InsuranceSuite applications and caller applications
    - To federate those authentication requests to the correct IdP
    - To construct JWTs that verify users and provide information about their authorization
- For services:
    - To authenticate services
    - To construct JWTs that verify services and provide information about their authorization

The insurer's identity provider

An *identity provider* (IdP) is an application or service that creates, maintains, and manages identity information for internal and external users. Every insurer using Guidewire cloud applications must provide an identity provider (IdP).

The primary responsibilities of the IdP are:

- For internal users and external users:
  - To maintain user names and passwords
  - To maintain information that identifies each user's authorization
  - To authenticate users and provide information about their authorization when a request is received from Guidewire Hub

The IdP does not play a role in service authentication or authorization.

Additional authorization applications

An *additional authorization application* is an application that contains additional information about the authorization of a specific caller. This additional information is not transferred in the JWT. Rather, it is retrieved by a plugin after the InsuranceSuite application has received the JWT but before authorization is determined.

There are several use cases for retrieving information from an additional authorization application, as opposed to storing it in the IdP and putting it in the JWT. This includes the following:

- The additional authorization application is the system of record for the relevant auth information. It is more efficient to retrieve this information directly from that system, as opposed to duplicating the information in the IdP.
- The size of the additional information is too large to store in a JWT. For example, if the caller is a producer, the producer's producer codes may be needed to determine their authorization. Some producers have hundreds of producer codes, and this is too much information to transfer in a JWT.

The primary responsibilities of the additional authorization application are:

- For all relevant types of callers:
  - To maintain information that identifies authorization for the relevant callers
  - To provide this information with a sufficiently rapid response time

For a given type of caller, if the amount of authorization information is small enough to be placed in a JWT, then use of an additional authorization application is possible but not required.

The caller applications

Every caller application that uses Cloud API must provide authentication information with every API call (except for unauthenticated calls).

From an authentication perspective, the primary responsibilities of each caller application are:

- For internal users and external users:
  - To send authentication requests to Guidewire Hub (which will then federate those requests to the appropriate IdP)
- For anonymous users:
  - To send unauthenticated requests to create accounts (The response to these requests includes a self-signed JWT created by PolicyCenter)
- For services:
  - To send authentication requests to Guidewire Hub (which are executed by Guidewire Hub without any involvement of the IdP)
- For all callers:
  - To temporarily store JWTs created by Guidewire Hub so that they can be included in Cloud API calls made for the associated callers
  - To temporarily store self-signed JWTs created by PolicyCenter for anonymous users so that they can be included in Cloud API calls made for those anonymous user

Cloud API

From an authentication perspective, the primary responsibilities of the Cloud API are:

- For authenticated callers:
  - Verify that each call includes valid authentication
  - Retrieve information from an additional authorization application when necessary
  - Limit the access of each call to only those endpoints, operations, fields, and specific resources that the user is authorized to use
- For unauthenticated callers:
  - Limit the access of each call to the appropriate endpoints, operations, fields, and resources
    - Typically, this access is limited to either API metadata only, or account creation for callers who will become anonymous users

Types of access

Authorization is the process of determining what operations and data the caller is allowed to access. Cloud API enforces authorization using the following types of access:

- *Endpoint access* defines the aspects of an endpoint's behaviors that are available to a caller. This includes:
  - What endpoints are available to the caller?
  - What operations can a caller call on the available endpoint?
  - What fields can the caller specify in a request payload or get in a response payload?
- *Resource access* defines, for a given type of resource, which instances of that resource type the caller can access. For example, for a given caller, endpoint access might grant access to a GET `/policies` endpoint. But this does not necessarily mean the caller can access every policy in the system. Resource access can limit which specific policies that caller can view.
- A *proxy user* is an internal user that is assigned to an external user or service when an external user or service triggers an API call. Whenever PolicyCenter logic must verify that the caller has a given domain-level system permission (such as permission to own an activity) or authority limit, the proxy user is checked. This is referred to as *proxy user access*.

Each type of access does not necessarily apply to every type of caller.

- All types of users are restricted by endpoint access.
- In the base configuration, only users are restricted by resource access. Services are not.
- Only external users and services are restricted by proxy user access. Internal users are not.

Authentication methods

Cloud API supports two authentication methods. The methods differ based on how authentication information is sent from the caller application to PolicyCenter.

Basic authentication

Basic authentication is an authentication method in which only the user's user name and password are provided, and they are provided in the request header.

- Internal users (and only internal users) can use basic authentication.
- With basic authentication, the authentication and authorization information is retrieved from the operational database using information in the request header.

Guidewire recommends using basic authentication only over HTTPS (SSL).

Note: Basic authentication is not supported in production environments. It can only be used in development environments. For more information, see“Basic authentication” on page 291.

Bearer token authentication

Bearer token authentication is an authentication method in which the authentication information is stored in a JSON Web Token (JWT, pronounced like "jot"). The phrase "bearer authentication" can be understood as "give access to the bearer of this token".

- Every type of caller can use bearer token authentication.
- With bearer token authentication, the JWT contains both authentication information and authorization information.

JWTs contain claims. (In some cases, this documentation uses the term "token claim" to differentiate between claims in a JWT and claims in the property and casualty insurance sense.) A JWT's *claim* is a piece of information asserted about the bearer of the token, such as the bearer's name. For bearer token authentication, authentication information is stored in claims.

Similar to basic authentication, Guidewire recommends using bearer token authentication only over HTTPS (SSL).

Constructing JWTs

Overview of JWTs

In bearer token authentication, the caller presents a *JSON Web Token (JWT)*. The JWT contains a set of claims. Each claim is a key/value pair that represents information that "the bearer of the token claims to be true". For example, a JWT could contain the following claim, which asserts the identity of the bearer of the token (in the `sub` claim, which identifies the "subject"):

```
[
  "sub": "rnewton@email.com",
 ...
]
```

Cloud API uses information in the JWT to determine the authorization to grant to the caller. This typically involves two types of information:

- Some information in the JWT identifies the API roles to assign to the caller. This determines the level of endpoint access the caller has.
- Some information in the JWT identifies the caller's resource access IDs. This determines the level of resource access the caller has. (In other words, this determines which specific resources that caller can access.)

For example, suppose Ray Newton is an insured making a request to PolicyCenter. The JWT includes the following.

```
[
  "sub": "rnewton@email.com",
  "groups": [
    "gwa.prod.pc.Account_Holder"
  ],
  "pc_accountNumbers": [
    "C000143542"
  ],
 ...
]
```

Cloud API grants authorization in this way:

- Based on the `groups` claim, the caller is given endpoint access as defined in the `Account_Holder` API role.
- Based on the `pc_accountNumbers` claim, the caller is given resource access to resources associated with the account whose account number is C000143542.

The structure and contents of a JWT vary based on the type of caller. But, for every type of caller using bearer token authentication, the JWT either contains the caller's API roles and resource access IDs, or it contains information used to determine the caller's API roles and resource access IDs.

Authorization information can come from one of three sources:

- Populating JWTS with information from the IdP
- Populating JWTS with information from the caller application
- Retrieving information with the IExpandTokenPlugin

Populating JWTs with information from the IdP

Authorization information can come from the **identity provider application (IdP)**. When the caller is a user (as opposed to a service), the caller application sends the caller's username and password to the IdP so that it can authenticate the user. The IdP provides a SAML response indicating whether the user has been authenticated. The IdP can also provide information about the user's authorization, such as:

- Which API roles ought to be granted to the user
- The resource access IDs for the user

Guidewire Hub adds any authorization information it receives from the IdP into the JWT. For more information about setting up your Idp, see *Authentication*.

For example, suppose Guidewire Hub requests authentication for Ray Newton. The IdP could respond with the following information:

- The caller is authenticated. (The provided password matches the provided username.)
- The caller's API roles are: gwa.prod.pc.Account_Holder
- The caller's resource access IDs are: C000143542

Guidewire Hub would construct a JWT that includes the following claims:

```
...
"groups": [
  "gwa.prod.pc.Account_Holder"
],
"pc_accountNumbers": [
  "C000143542"
],
...
```

This provides the caller with endpoint access as defined in the Account_Holder API role, and resource access to resources associated with any account whose account number is C000143542.

Populating JWTs with information from the caller application

Authorization information can come from the **caller application** itself. When the caller is a service (as opposed to a user), the caller application sends a scope to Guidewire Hub. This scope identifies the API role to grant to the caller. If the application has registered that scope with Guidewire Hub, Guidewire Hub adds this information to the JWT. (Services are not authenticated by IdPs, and services do not need to provide resource access IDs as they use of a resource access strategy that has open access to all resources.)

For example, suppose Guidewire Hub receives a request from the ACME FNOL Reporter service with the following information:

- The caller's scope is scp.cc.acme_fnolreporter.

Guidewire Hub would construct a JWT that includes the following claim:

```
...
"scp": [
  "scp.cc.acme_fnolreporter"
],
...
```

This provides the caller with endpoint access as defined in the acme_fnolreporter API role. Because the caller is a service, it is not bound by resource access.

Retrieving information through the IExpandTokenPlugin

Authorization information can come from the **IExpandTokenPlugin** plugin. This plugin is called after the JWT has been received from the caller but before authorization has been determined. Cloud API extracts the information from

the JWT into a "token map". It then calls the plugin, which can make a call to an external system and then add information from the external system to the token map. This map is then used to determine the caller's authorization.

For example, suppose an insurer wants to provide access to insureds, but they do not want to store API role information for insureds in the IdP. This insurer could configure the IExpandTokenPlugin plugin to do the following:

- Extract the value of the `sub` (subject) claim from the token map.
- Send the value to the appropriate external system, which responds with a list of API roles that define endpoint access for the insured
- Add the API roles to the token map's `groups` claim.

The JWT received from Guidewire Hub would look like this:

```
...
"sub": "rnewton@email.com",
"groups": [
],
...
```

After the IExpandTokenPlugin plugin makes it call, the token map would look like this:

```
...
"sub": "rnewton@email.com",
"groups": [
  "gwa.prod.pc.Account_Holder"
],
...
```

As another example, suppose an insurer wants to provide access for producers. Resource access for producers is determined by producer codes. Each producer can have up to several hundred producer codes, and the amount of data may exceed what can be stored in a JWT. This insurer could configure the IExpandTokenPlugin plugin to do the following:

- Extract the value of the `sub` (subject) claim.
- Send this value to the appropriate external system, which responds with a list of producer codes that define resource access for the producer
- Add the producer codes to the token map's `cc_producerCodes` claim.

The JWT received from Guidewire Hub would look like this:

```
...
"sub": "kegerston@allrisk.com",
"producerCodes": [
],
...
```

After the IExpandTokenPlugin plugin makes its call, the token map would look like this:

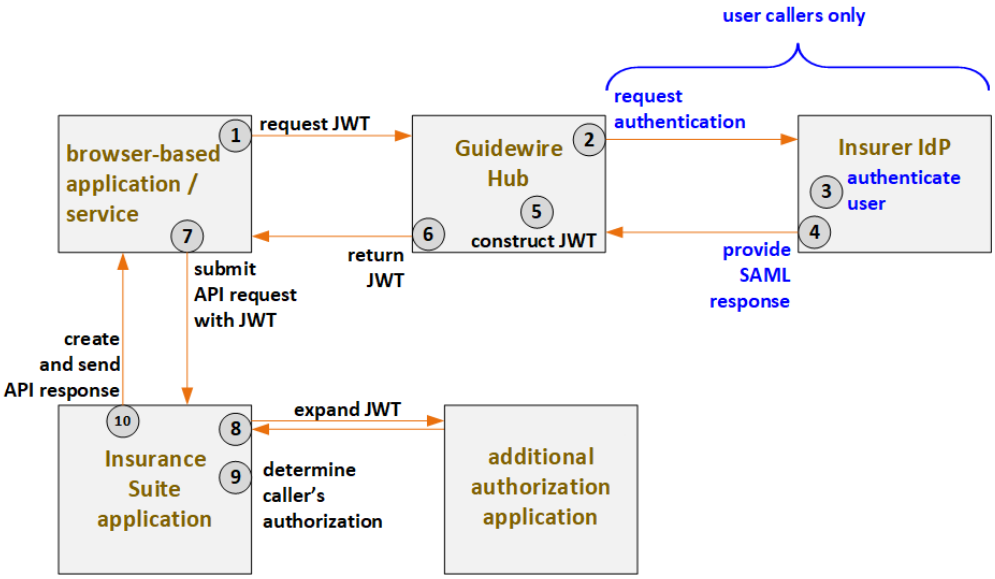
```
...
"sub": "kegerston@allrisk.com",
"producerCodes": [
  "100-002541",
  "100-002542",
  "100-002543",
  ...
],
...
```

Note: The IExpandTokenPlugin plugin does not modify the JWT itself. It modifies only the map that contains information extracted from the JWT. If the JWT is passed on to some other system after the IExpandTokenPlugin plugin has been called, the JWT will be in its original form. It will not contain information retrieved by the plugin.

For more information on configuring the IExpandTokenPlugin, see “Configuring the IExpandTokenPlugin plugin” on page 395.

Summary of JWT flow

The following diagram summarizes the way in which information is added to the JWT and then used.



1. The caller application (which could be a browser-based application supporting users or a service) requests a JWT from Guidewire Hub. If the caller application is a service, this request includes the service's API roles.
2. For user callers, Guidewire Hub requests authentication from the IdP. (For services, authentication is performed by Guidewire Hub.)
3. For user callers, the IdP authenticates the user.
4. For user callers, the IdP provides a SAML response. This response must include verification that the user has been authenticated. It must also include information about the user's API roles and resource access IDs, or any lookup values needed by the IExpandTokenPlugin plugin to retrieve API roles and resource IDs.
5. Guidewire Hub constructs the JWT.
6. Guidewire Hub returns the JWT to the caller application.
7. The caller application submits the API request to the InsuranceSuite application. The JWT is included in the request header.
8. Cloud API extracts information from the JWT and puts it into a token map. If necessary, the IExpandTokenPlugin plugin calls an external auth provider to retrieve any additional auth information and add it to the token map.
9. The caller's authorization is determined using the token map, which contains information from the original JWT and optionally information added by the IExpandTokenPlugin plugin.
10. The InsuranceSuite application processes the request and sends the response to the caller application.

Configuration requirements for JWTs

For user callers, the token map must ultimately contain the API roles and resource access IDs that define the user's authorization. These values must either:

- Be included in the SAML response from the IdP, or
- Be retrieved from an external system using a lookup value that is in the JWT constructed by Guidewire Hub

For most types of callers, use of the IExpandTokenPlugin plugin is optional. However, Guidewire does specifically recommend its use in the case where producers are accessing ClaimCenter. The access strategy for producers relies on producer codes. Producers often have large numbers of producer codes, and the number may exceed the maximum of what can be sent in a JWT. Therefore, Guidewire recommends that if you have producers accessing ClaimCenter, you always add the producer codes to the JWT through the IExpandTokenPlugin plugin.

For more information on configuring the IdP, see “Configuring the IdP” on page 370.

For more information on registering the caller application with Guidewire Hub, see “Registering the caller application with Guidewire Hub” on page 372.

For information on configuring the IExpandTokenPlugin plugin, see “Configuring the IExpandTokenPlugin plugin” on page 395.

Authentication failure error messages

Endpoints that return elements

For endpoints that return elements, when a given resource does not exist, Cloud API throws a `NotFoundException` with a user message similar to "No resource was found at path...". When a given resource does exist, but the user lacks sufficient resource access, Cloud API throws the same exception with the same user message. This approach is considered to be more secure as it prevents malicious callers from being able to verify the existence of data that they are not authorized to access.

For example, suppose a user executes GET `/activities/xc:20`. Also, suppose activity xc:20 exists but the user lacks sufficient resource access, the following error is returned:

```
"status": 404,
        "errorCode": "gw.api.rest.exceptions.NotFoundException",
        "userMessage": "No resource was found at path /activities/xc:20"
```

Endpoints that return collections

For endpoints that return collections, Cloud API returns all resources that meet the criteria and for which the user has sufficient resource access. If a resource exists, but the user lacks sufficient resource access, Cloud API omits it from the results. This approach is considered to be more secure as it prevents malicious callers from being able to verify the existence of data that they are not authorized to access.

For example, suppose a user executes GET `/activities`. Also, suppose that there are three activities in the database: xc:10, xc:20, and xc:30. The user has sufficient resource access to view xc:10 and xc:30, but not xc:20. The call returns the following:

```
{
  "count": 2
  "data": [
    {
      "attributes": {
        "id": "xc:10",
        ... },
      ...
    },
    {
      "attributes": {
        "id": "xc:30",
        ... },
      ...
    }
  "links": { ... }
}
```

List of developer tasks

There are a number of tasks an insurer must execute to enable and use authentication for each type of caller.

Configuring authentication

During implementation, for a given type of caller, an insurer may need to:

- Enable asymmetric encryption
- Provide deployment information

- Register the caller application with Guidewire Hub
- Configure the IdP
- Configure service account mapping
- Configure endpoint access
- Configure resource access
- Configure proxy user access
- Configure the IExpandTokenPlugin plugin

Making API calls

In production, for a given type of caller, the caller may need to:

- Request a code from Guidewire Hub
- Request a JWT from Guidewire Hub
- Send an API call to PolicyCenter with a JWT

chapter 48

Selecting an authentication flow

Within the context of Cloud API, an *auth flow* is a flow of authentication and authorization information for a particular type of caller. Cloud API supports multiple auth flows. This topic identifies the issues to consider when choosing an auth flow for a particular caller application.

The most important issues to consider are as follows:

- What OAuth flow must the caller application use?
- Which user is attached to the session?
- Where are authorization values stored?
- Who enforces resource access?
- What values are used as resource access IDs?

This topic assumes you are familiar with the Cloud API authentication architecture and the meaning of the terms endpoint access, resource access, and proxy user access. For more information, see "Overview of authentication" on page 271.

Auth flows to choose from

Cloud API supports the following auth flows for bearer token authentication.

Auth flow	Definition of who the caller is
Internal user	A person who is known to the insurer and listed as a user in the PolicyCenter database, such as a claims adjuster or underwriter.
External user	A person who is known to the insurer, but who is not listed as a user in the PolicyCenter database, such as a policyholder.
Standalone service	A service that executes calls as itself.
Service with internal user context	A service that executes calls on behalf of internal users.
Service with external user context	A service that executes calls on behalf of external users.
Service with service account mapping	A service that is mapped to an internal service account and whose access is determined by the settings for that service account.

Cloud API also supports auth flows that do not make use of bearer token authentication. For more information, see "Additional auth flows" on page 287.

Detailed discussion of issues to consider

When determining the best bearer token auth flow for a given caller application, Guidewire recommends you consider the issues discussed in the following topics.

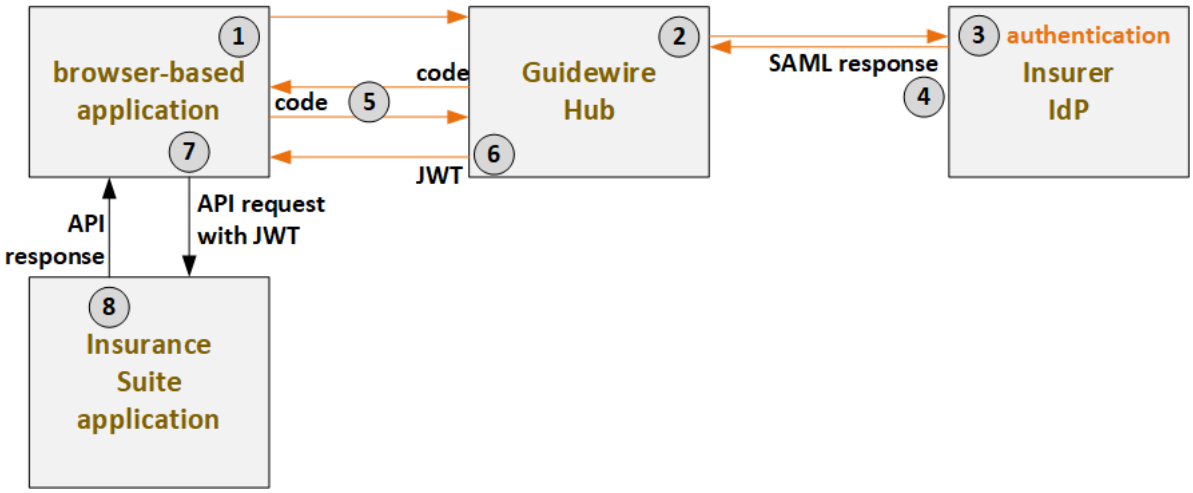
For a summary of the issues, and the behaviors for each auth flow, see “Summary of the issues to consider” on page 287.

Which OAuth flow must the caller application use?

Cloud API supports two OAuth flows: authorization code flow and client credential flow.

Authorization code flow

Authorization code flow is designed for browser-based applications that typically have a user interface and that users interact with.



Within authorization code flow:

1. The caller application requests a JWT from Guidewire Hub.
2. Guidewire Hub sends the authentication request to the appropriate IdP.
3. The IdP authenticates the user.
4. The IdP provides a SAML response with the information about the user's authorization. This could include endpoint access role names and resource access IDs.
5. Guidewire Hub sends a code to the caller application. The caller application uses this code to request a JWT.
6. Guidewire Hub sends the JWT to the caller application.

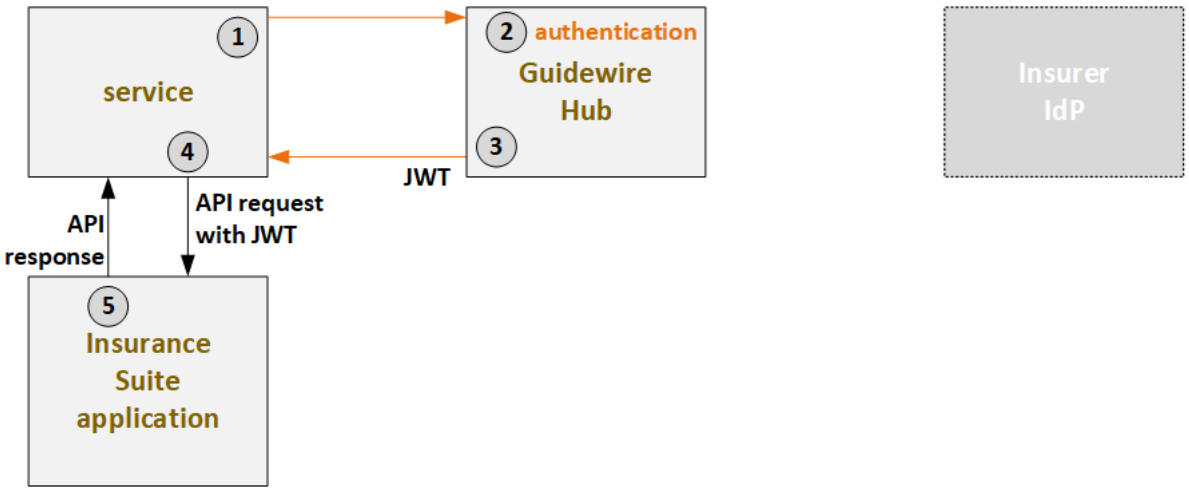
Once the caller application has the JWT, it can send the API request to PolicyCenter (7), which then processes the request and sends the reply (8).

Authorization code flow can be used with the following Cloud API auth flows:

- Internal user
- External user

Client credential flow

Client credential flow is designed for services that typically do not have a user interface and that take action without any synchronous user input.



Within client credential flow:

1. The caller requests a JWT from Guidewire Hub.
2. Guidewire Hub authenticates the caller.
3. Guidewire Hub sends the JWT to the caller application.

Once the caller application has the JWT, it can send the API request to PolicyCenter (4), which then processes the request and sends the reply (5).

Client credential code flow can be used with the following Cloud API auth flows:

- Standalone service
- Service with internal user context
- Service with external user context
- Service with service account mapping

Summary of behaviors

The following table summarizes these behaviors.

	Internal User	External User	Standalone service	Service with Internal User Context	Service with External User Context	Service with Service Account Mapping
OAuth flow	Authorization code flow	Authorization code flow	Client credential flow	Client credential flow	Client credential flow	Client credential flow

For a summary of all the issues to consider in a single table, see “Summary of the issues to consider” on page 287.

Which user is attached to the session?

Every session must have a user attached to it. This user is used in the following ways:

- If the call creates or modifies data, the user name is recorded as the `CreateUser` or `UpdateUser`.
- If the call creates a history event, the user is attached to the event.
- If the call assigns an object, the user name is used in "assigned by *user*" information.

Also, once Cloud API has confirmed there is sufficient authorization to use a given endpoint and view a given resource:

- If the call also triggers an authority limit check, the user's authority profiles are checked.

- If the call also triggers a domain-level permissions check, the user's permissions are checked.

Every caller can have their own user account

For some auth flows, every caller can have their own user account. In these situations, session users are assigned on a per-caller basis. These behaviors can also be controlled separately for each caller. The following Cloud API auth flows support this:

- Internal user
- Service with internal user context
- Service with service account mapping

Multiple callers share a single proxy user account

For other auth flows, multiple callers share a single proxy user account. This occurs in situations where individual callers are not listed in the PolicyCenter database, and therefore session users cannot vary from caller to caller. For these situations, a single proxy user is assigned to an entire type of caller (external user or service). The following Cloud API auth flows support this:

- External user
- Standalone service
- Service with external user context

Summary of behaviors

The following table summarizes these behaviors.

	Internal User	External User	Standalone service	Service with Internal User Context	Service with External User Context	Service with Service Account Mapping
Can each call have its own user attached to the session?	Yes	No (a single "external proxy user" is used for all relevant calls)	No (a single "service proxy user" is used for all relevant calls)	Yes	No (a single "service proxy user" is used for all relevant calls)	Yes

For a summary of all the issues to consider in a single table, see “Summary of the issues to consider” on page 287.

Where do authorization values come from?

For every caller, you need to store a set of values that determine the caller's endpoint access. These values determine which endpoints and operations the caller can use. For some callers, you must also store values that determine the caller's resource access. These values determine which specific instances of a given resource the caller can view or edit. These are collectively referred to here as *authorization values*.

The IdP

For some auth flows, the IdP must either store the authorization values or provide some sort of lookup value that can be used to retrieve the authorization values from an additional authorization system. The authorization values (or the lookup value) must be included in the IdP's SAML response. The following Cloud API auth flows support this:

- Internal user
- External user

The caller application itself

For other auth flows, the caller application itself must either provide the authorization values or provide some sort of lookup value that can be used to retrieve the authorization values from an additional authorization system. The following Cloud API auth flows support this:

- Standalone service
- Service with internal user context
- Service with external user context

The Guidewire configuration

For the service with service account mapping auth flow, the caller application provides a client ID. This ID is mapped to a service account in the PolicyCenter database, and this service account is used to determine the authorization values. For this auth flow, the authorization values come from the mapping information, which is stored in the Guidewire configuration itself.

Summary of behaviors

The following table summarizes these behaviors.

	Internal User	External User	Standalone service	Service with Internal User Context	Service with External User Context	Service with Service Account Mapping
Where do authorization values (or an appropriate lookup value) come from?	The IdP	The IdP	The service itself (endpoint access values only; resource access IDs are not applicable)	The service itself	The service itself	The Guidewire configuration

For a summary of all the issues to consider in a single table, see “Summary of the issues to consider” on page 287.

Who enforces resource access?

Cloud API

For some auth flows, resource access is enforced by Cloud API. For these auth flows, the call either includes resource access IDs, or includes information that can be used to retrieve resource access IDs. These IDs determine which specific instances of a resource the caller can access. The following Cloud API auth flows support this:

- Internal user
- External user
- Service with internal user context
- Service with external user context
- Service with service account mapping

The caller application itself

For other auth flows, Cloud API provides unrestricted resource access. This is done under the assumption that resource access will be enforced by the caller application itself. In this case, calls do not include resource access IDs. The caller is given access to any specific resource, provided they have sufficient endpoint access. The following Cloud API auth flows support this:

- Standalone service

Summary of behaviors

The following table summarizes these behaviors.

	Internal User	External User	Standalone service	Service with Internal User Context	Service with External User Context	Service with Service Account Mapping
Does Cloud API enforce resource access?	Yes	Yes	No (The service is expected to enforce it.)	Yes	Yes	Yes

For a summary of all the issues to consider in a single table, see "Summary of the issues to consider" on page 287.

What values are used as resource access IDs?

User names

For some auth flows, resource access IDs are user names. For these auth flows, every call must present a user name. Resource access is determined by this name. The following Cloud API auth flows support this:

- Internal user
- Service with internal user context

Business IDs

For other auth flows, resource access IDs are business IDs. These IDs represent either:

- Something the caller owns, such as account numbers (for PolicyCenter account holders)
- Who the caller is, such as contact IDs (for ClaimCenter claimants or BillingCenter contacts), producer codes (for ClaimCenter producers), or address book unique identifiers (for vendors providing services for ClaimCenter claims)

For these auth flows, every call must present one or more business IDs, or a lookup value that can be used to retrieve the business IDs from an additional authorization application. Resource access is then based on which resources the caller owns. The following Cloud API auth flows support this:

- External user
- Service with external user context

No resource IDs

There are two auth flows that do not use resource IDs.

For standalone services, resource access is enforced by the service itself, and not by Cloud API. Therefore, there is no need to provide resource access IDs.

For services with service account mapping, the service is mapped to a service account. Information in the service account is used to determine resource access, but there are no resource IDs passed within the auth flow.

Summary of behaviors

The following table summarizes these behaviors.

	Internal User	External User	Standalone service	Service with Internal User Context	Service with External User Context	Service with Service Account Mapping
What are the resource access IDs?	user names	IDs for business data (such as contact IDs and account numbers)	not applicable	user names	IDs for business data (such as contact IDs and account numbers)	not applicable

For a summary of all the issues to consider in a single table, see "Summary of the issues to consider" on page 287.

Summary of the issues to consider

The following table summarizes the issues to consider and the behavior of each auth flow regarding that issue. The last row of the table contains links to view more detailed information about the relevant auth flow.

	Internal User	External User	Standalone service	Service with Internal User Context	Service with External User Context	Service with Service Account Mapping
OAuth flow	Authorization code flow	Authorization code flow	Client credential flow	Client credential flow	Client credential flow	Client credential flow
Can each call have its own user attached to the session?	Yes	No (a single "external proxy user" is used for all relevant calls)	No (a single "service proxy user" is used for all relevant calls)	Yes	No (a single "service proxy user" is used for all relevant calls)	Yes
Where do authorization values (or an appropriate lookup value) come from?	The IdP	The IdP	The service itself (endpoint access values only; resource access IDs are not applicable)	The service itself	The service itself	The Guidewire configuration
Does Cloud API enforce resource access?	Yes	Yes	No (The service is expected to enforce it.)	Yes	Yes	Yes
What are the resource access IDs?	user names	IDs for business data (such as contact IDs and account numbers)	not applicable	user names	IDs for business data (such as contact IDs and account numbers)	not applicable
For more information on this auth flow	"OAuth2 authorization code flow: Internal users" on page 297	"OAuth2 authorization code flow: External users" on page 303	"OAuth2 client credential flow: Standalone services" on page 325	"OAuth2 client credential flow: Services with user context" on page 333	"OAuth2 client credential flow: Services with user context" on page 333	"OAuth2 client credential flow: Services with service account mapping" on page 349

Additional auth flows

Cloud API supports additional auth flows that do not make use of bearer token authentication:

Auth flow	Definition
Basic authentication	The caller is a person who is known to the insurer and listed as a user in the PolicyCenter database, such as a claims adjuster or underwriter. The call uses basic authentication (where auth information is passed in the request header) and not bearer token authentication (where auth information is passed in a JWT (JSON Web Token)). Basic authentication is not supported in production environments. Basic authentication can be an appropriate option for development environments, as it bypasses the need to have a working integration with Guidewire Hub or, when relevant, an IdP.
Anonymous user	A person who is not yet known to the insurer, but who is interested in quoting and possibly binding a policy.

Auth flow	Definition
	The anonymous user flow involves at least two calls. In the first call, the user creates an account as an unauthenticated user. This call does not involve a JWT. PolicyCenter provides a self-signed JWT in the response to the first call. In the second call, the user presents the self-signed JWT.
Unauthenticated user	The caller is a person or service who presents no authentication information. Typically, this type of caller is either a caller who accesses metadata only, or a caller who is creating an account as part of the process to become an anonymous user. Unauthenticated user access, when used by itself, can be appropriate for internal services that only need access to metadata, such as a service that is querying for the definition of an API. (Unauthenticated user access is also part of the anonymous auth flow.)

The following table summarizes the issues to consider. It identifies the options for each issue, and which auth flow supports each option. The final row of the table provides a link which you can follow to get more detailed information about that auth flow.

	Basic Auth	Anonymous User (PolicyCenter only)	Unauthenticated User
OAuth flow	not applicable	not applicable as an anonymous user (Anonymous users who bind policies become external users, and from then on they use authorization code flow.)	not applicable
Can each call have its own user attached to the session?	Yes	No (For the first call, a single "unauthenticated proxy user" is used. For the second call, a single "external proxy user" is used.)	No (a single "unauthenticated proxy user" is used for all relevant calls)
Where do authorization values (or an appropriate lookup value) come from	not applicable (no authorization values are passed within the auth flow)	The first call includes no authorization values. For the second call, the authorization values are in the self-signed JWT provided by PolicyCenter.	not applicable (no authorization values are provided)
Does Cloud API enforce resource access?	Yes	Yes	Yes
Where do the resource access IDs come from?	not applicable (no authorization values are passed within the auth flow)	PolicyCenter	not applicable (no authorization values are provided)
For more information on this auth flow	“Basic authentication” on page 291	“OAuth2 authorization code flow: Anonymous users” on page 313	“Unauthenticated callers” on page 359

part 9

Authentication flows in detail

Cloud API supports several different authentication flows. Each flow supports one of the following types of callers:

- Internal users using basic auth
- Internal users using bearer token auth
- External users
- Anonymous users
- Standalone services
- Services with user context
- Services with service account mapping
- Unauthenticated callers

This section describes each of these flows in detail.

chapter 49

Basic authentication

Within the context of Cloud API authentication, an *internal user* is a person who is listed as a user in the PolicyCenter database. For example, Alice Applegate, a PolicyCenter underwriter, is an internal user. Internal users can use caller applications and trigger Cloud API calls from that application. For example, suppose there is a location photography portal that contains pictures of covered buildings taken by a third-party field agent. An underwriter reviews and selects pictures to be saved to PolicyCenter. This action triggers a Cloud API call by an internal user from a caller application.

Internal users can be authenticated using either basic authentication or bearer token authentication. *Basic authentication* is an authentication method in which only the user's user name and password are provided, and they are provided in the request header.

- Internal users (and only internal users) can use basic authentication.
- With basic authentication, the authentication information is retrieved from the operational database using information in the request header

Basic authentication is not supported in production environments.

Basic authentication can be useful in development when you want to test aspects of endpoint behavior that are not related to authentication. Basic authentication does not require any interaction with Guidewire Hub to generate JWTs. You can authenticate a Cloud API call using only the caller application and PolicyCenter.

This topic describes how to implement basic authentication for internal users. (For information on how to implement bearer token authentication for internal users, see “OAuth2 authorization code flow: Internal users” on page 297.)

Overview of basic authentication

Authentication includes credentials and authorization. Authentication information for basic authentication is specified in request headers.

Credentials

With basic authentication, every internal user's credential information is stored in the PolicyCenter database.

The user name and password is provided by each caller in the request object's header.

Authorization

Endpoint access with basic authentication

Endpoint access defines the aspects of an endpoint's behaviors that are available to a caller. This includes:

- What endpoints and resource types are available to the caller?
- What operations can a caller call on the available endpoint?
- What fields can the caller specify in a request payload or get in a response payload?

Endpoint access is controlled by API roles. An *API role* is a list of endpoints, operations, and fields that are available to a set of callers through API calls. API roles act as allowlists. By default, a caller has no endpoint access. When the caller is associated with one or more API roles, they gain access to the endpoints, operations, and fields allowlisted in each of those API roles.

When an internal user makes a Cloud API call (using either basic authentication or bearer token authentication), PolicyCenter queries the operational database for this internal user's user roles. The user is given endpoint access to all API roles whose names corresponds to the names of the user's user roles.

For example, suppose that Alice Applegate is an internal user with two user roles: Underwriter and Reinsurance Manager. Alice Applegate triggers a Cloud API call. When the API call is received, PolicyCenter queries the database for Alice's user roles. Two user roles are returned: Underwriter and Reinsurance Manager. PolicyCenter then grants Alice the endpoint access defined in the API roles named "Underwriter" and "Reinsurance Manager".

For more information on how API roles are configured, see “Endpoint access” on page 373.

Resource access with basic authentication

Resource access defines, for a given type of resource, which instances of that resources the caller can access. For example, suppose there is a GET `/claims` endpoint that is available to policyholders, underwriters, adjusters, and service vendors. All of these callers can use the endpoint to access resources whose type is `claim`, but each caller may be restricted so that they can access only a subset of the claims. For example:

- A policyholder may be able to see only the claims associated with the policies they hold.
- An underwriter may be able to see only the claims for policies assigned to them.
- An adjuster may be able to see only the claims assigned to them.
- A service vendor may be able to see only the claims that have a service request assigned to them.

A *resource access strategy* is a set of logic that identifies the meaning of a resource access ID. The base configuration includes the following resource access strategies for internal users:

Strategy name	Persona using this strategy	The resource access ID is assumed to be...	Grants access to...
`pc_username`	Internal users	A PolicyCenter user name	Any information this internal user could see in PolicyCenter based on their associated Access Control Lists (ACLs).

When an internal user makes a Cloud API call, the user name is used as the resource access ID. The `pc_username` strategy is used automatically. This strategy consists of Cloud API logic that matches, as closely as possible, the user's access as defined in the base configuration's Access Control Lists (ACLs).

For more information on how resource access behaves, see “Resource access” on page 385.

Proxy user access with basic authentication

Proxy user access is not applicable to basic authentication.

Request headers

For basic authentication, authorization information is sent to PolicyCenter with the request's authorization header. The header must use this format:

```
Authorization: Basic <token>
```

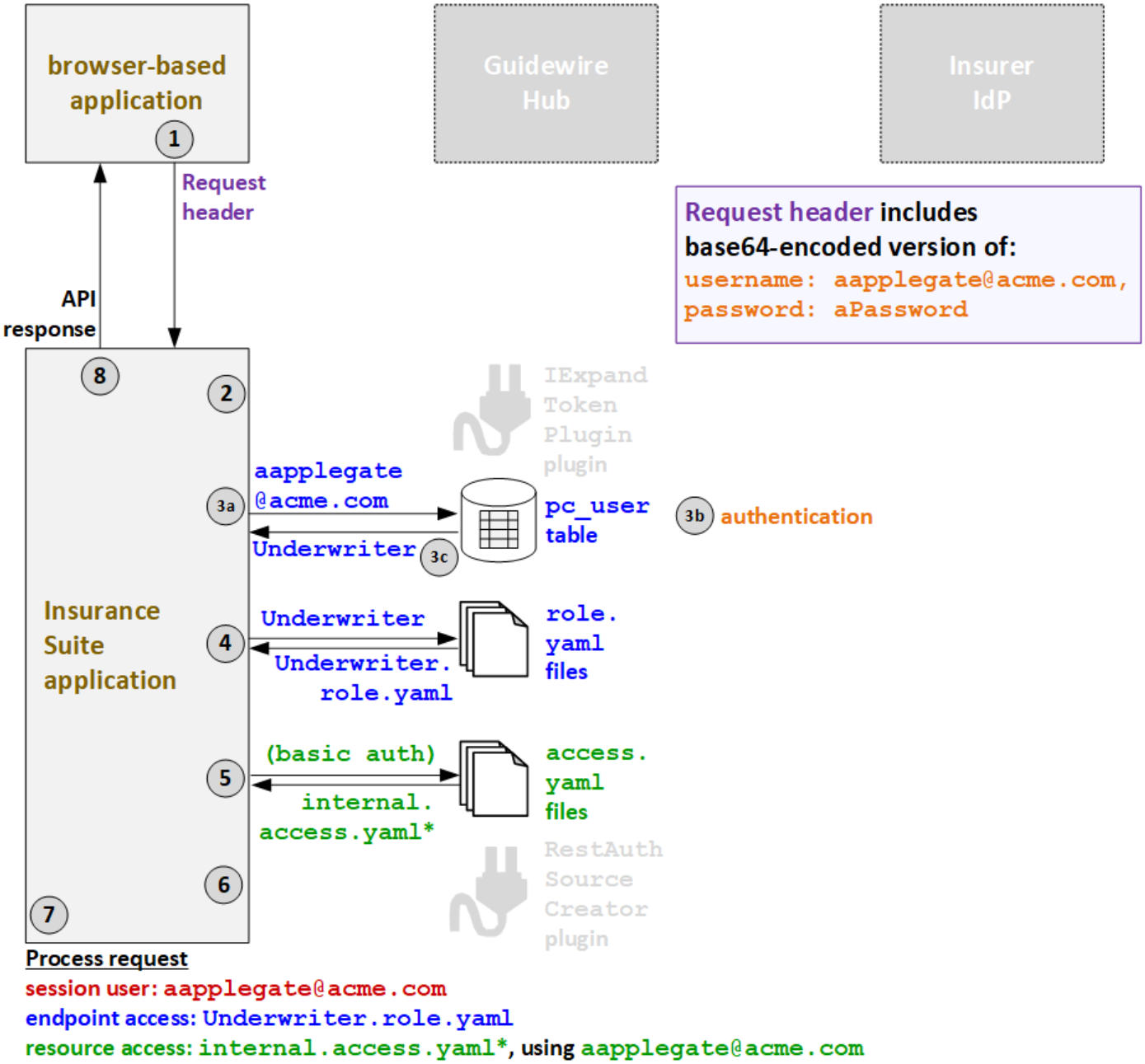
Example flow for basic authentication

The following diagram identifies the flow of authentication and authorization information for basic authentication. Colors are used in the following ways:

- Orange - credentials information
- Blue - endpoint access information
- Green - resource access information
- Red - proxy user and session user information

Sometimes, a value is used in different parts of the flow to determine different types of access. These values appear in the flow diagrams multiple times, but the color is changed to reflect how the value is being used at that point in the process.

In the following example, an API call is triggered by Alice Applegate, who is an internal user, using a browser-based application and basic authentication.



1. When Alice triggers an API call, the caller application sends the API request to PolicyCenter. The request header includes a base64-encoded version of the user's user name (`aapplegate@acme.com`) and password (`aPassword`).
2. The IExpandTokenPlugin plugin is not relevant for basic authentication.
3. PolicyCenter authenticates the user and determines the endpoint access.
   a. Using the user name in the request header (`aapplegate@acme.com`), PolicyCenter queries the user table.
   b. PolicyCenter authenticates the user by verifying that the user name and password match.
   c. PolicyCenter responds with the user roles that this user has. One role is returned: `Underwriter`.
4. Based on the returned role, the `Underwriter.role.yaml` API role file is used to define the endpoint access.
5. Next, PolicyCenter determines the resource access strategy. Because the call is using basic authentication, PolicyCenter grants resource access as defined in the `internal access.yaml` files. (* PolicyCenter starts with

`internal_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with `"internal"`.)

6. Proxy user access is not relevant for basic authentication.
7. PolicyCenter processes the request.
   a. The session user is the internal user: `aapplegate@acme.com`.
   b. The endpoint access is defined by `Underwriter.role.yaml`.
   c. The resource access is defined by `internal access.yaml` using the resource access ID of `aapplegate@acme.com`.
8. PolicyCenter provides the response to the initial call.

Supported environments for basic auth

Basic auth is not support for production environments. In this context, a "production environment" is one in which any of the following are true:

- The server is started in production mode.
- The server specifies an orbit (previously called planet class) and the orbit category is either "prod" or "preprod".

Basic auth is supported for development environments only. In this context, a "development environment" is one in which all of the following are true:

- The server is started in development mode.
- The server either has no specified orbit (planet class) or has a orbit category of "lower".

Disable basic auth in development environments

About this task

By default, basic auth is available only for instances of PolicyCenter running in a development environment. However, you can disable basic auth so that it is never available, even in development environments.

Procedure

1. Navigate to the plugin registry entry for the RestAuthenticationSourceCreatorPlugin plugin.
   - For more information on the plugin registry, see *Plugins, Prebuilt Integrations, and SOAP APIs*.
2. Set the `basicAuth` parameter to `allOff`.
3. Restart the server.

Implementation checklist for basic authentication

To configure Cloud API for basic authentication, you may need to do the following tasks:

Task	More Information
Create or modify API roles	“Endpoint access” on page 373
Review the resource access provided to internal users	“Resource access” on page 385

To make a Cloud API call for basic authentication, the caller application must:

1. Send the API call using basic authentication.

For more information, see “Sending authenticated calls with basic authentication” on page 296.

Sending authenticated calls with basic authentication

To request authentication for an internal user using basic authentication, the caller application must include the authentication information in the header of the request object. For example:

```
--header 'Authorization: Basic YWFwcGxlZ2F0ZTpndw=='
```

In the base configuration, when PolicyCenter receives a call with basic auth information in the header, it queries the database to verify that the user is a known internal user and that the password matches the user name. If these two things are true, the internal user is authenticated.

Authentication failure error messages

For endpoints that return a single element, when a given resource exists but the user lacks authorization to access it, Cloud API throws the following user message. This is the same message that is returned when the resource does not exist.

```
"status": 404,
        "errorCode": "gw.api.rest.exceptions.NotFoundException",
        "userMessage": "No resource was found at path <path>"
```

For endpoints that return collections, Cloud API returns all resources that meet the criteria and for which the user has sufficient resource access. If a resource exists, but the user lacks sufficient authorization, Cloud API omits it from the results.

These approaches are considered to be more secure as they prevent malicious callers from being able to verify the existence of data that they are not authorized to access.

Send a Postman call with basic authentication

About this task

You can use basic authentication with Cloud API calls made from Postman.

Procedure

1. Open Postman.
2. Start a new request by clicking the + to the right of the **Launchpad** tab.
3. Specify an operation and URL as appropriate.
4. To provide authorization using basic authorization:
   a) Click the **Authorization** tab.
   b) For the **Type** drop-down list, select *Basic Auth*.
   c) In the **Username** field, enter the user name (such as `aapplegate`).
   d) In the **Password** field, enter the password (such as `gw`).
5. Click the **Send** button to the right of the request field.

Results

Every Postman tab has its own authentication information. When you modify the request on an existing tab by changing the URL or choosing a new operation, you do not need to re-enter the authentication information. But when you open a new tab, you do need to provide authentication information. If you encounter a NotFoundException, such as in the following example, this could be caused by not providing correct authentication information.

```
"status": 404,
"errorCode": "gw.api.rest.exceptions.NotFoundException",
"userMessage": "No resource was found at path /common/v1/activities/cc:20"
```

chapter 50

OAuth2 authorization code flow: Internal users

Within the context of Cloud API authentication, an *internal user* is a person who is listed as a user in the PolicyCenter database. For example, Alice Applegate, a PolicyCenter underwriter, is an internal user. Internal users can use caller applications and trigger Cloud API calls from that application. For example, suppose there is a location photography portal that contains pictures of covered buildings taken by a third-party field agent. An underwriter reviews and selects pictures to be saved to PolicyCenter. This action triggers a Cloud API call by an internal user from a caller application.

This topic describes how to implement Cloud API authentication for internal users using bearer token authentication. (For information on how to implement authentication for internal users using basic authentication, see "Basic authentication" on page 291.)

Overview of authentication for internal users

Authentication includes credentials and authorization. Authentication information for internal users (using bearer token authentication) is specified in JWTs, and information from these JWTs is recorded in the logs.

Credentials

An internal user's credentials consist of a user name and password. This information is stored in the IdP.

Before an internal user can make an API call, the caller application sends a request to the appropriate IdP to authenticate the user. This typically consists of confirming that the provided username and password are correct.

For more information on how to configure the IdP, see "Configuring the IdP" on page 370.

Authorization

Endpoint access for internal users

Endpoint access defines the aspects of an endpoint's behaviors that are available to a caller. This includes:

- What endpoints and resource types are available to the caller?
- What operations can a caller call on the available endpoint?
- What fields can the caller specify in a request payload or get in a response payload?

Endpoint access is controlled by API roles. An *API role* is a list of endpoints, operations, and fields that are available to a set of callers through API calls. API roles act as allowlists. By default, a caller has no endpoint access. When the

caller is associated with one or more API roles, they gain access to the endpoints, operations, and fields allowlisted in each of those API roles.

When an internal user makes a Cloud API call (using either basic authentication or bearer token authentication), PolicyCenter queries the operational database for this internal user's user roles. The user is given endpoint access to all API roles whose names corresponds to the names of the user's user roles.

For example, suppose that Alice Applegate is an internal user with two user roles: Underwriter and Reinsurance Manager. Alice Applegate triggers a system API call. When the API call is received, PolicyCenter queries the database for Alice's user roles. Two user roles are returned: Underwriter and Reinsurance Manager. PolicyCenter then grants Alice the endpoint access defined in the API roles named "Underwriter" and "Reinsurance Manager".

For more information on how API roles are configured, see “Endpoint access” on page 373.

Resource access for internal users

Resource access defines, for a given type of resource, which instances of that resources the caller can access. For example, suppose there is a GET `/claims` endpoint that is available to policyholders, underwriters, adjusters, and service vendors. All of these callers can use the endpoint to access resources whose type is `claim`, but none of the callers can access all of the claims. For example:

- A policyholder may be able to see only the claims associated with the policies they hold.
- An underwriter may be able to see only the claims for policies assigned to them.
- An adjuster may be able to see only the claims assigned to them.
- A service vendor may be able to see only the claims that have a service request assigned to them.

A *resource access strategy* is a set of logic that identifies the meaning of a resource access ID. The base configuration includes the following resource access strategies for internal users:

Strategy name	Persona using this strategy	The resource access ID is assumed to be...	Grants access to...
`pc_username`	Internal users	A PolicyCenter user name	Any information this internal user could see in PolicyCenter based on their associated Access Control Lists (ACLs).

When an internal user makes a Cloud API call, the user's user name is included in the JWT. The user name is used as the resource access ID. The `pc_username` strategy is used automatically. This strategy consists of Cloud API logic that matches, as closely as possible, the user's access as defined in the base configuration's Access Control Lists (ACLs).

For more information on how resource access behaves, see “Resource access” on page 385.

Proxy user access for internal users

Proxy user access is not applicable for internal users.

JWTs for internal users

JSON Web Tokens (JWTs) contain token claims. (In standard JWT parlance, these are referred to simply as "claims". To avoid confusion with claims in the property and casualty insurance sense, this documentation sometimes refers to JWT claims as "token claims".) A *token claim* is a piece of information asserted about the bearer of the token, such as the bearer's name. For bearer token authentication, authentication information is stored in token claims.

JWTs for internal users can include the following token claims:

- `scp` - The resource access strategy to apply to the resource access ID. (For internal users, this is set to `pc_username`.)
- `pc_username` - The resource access ID. (This is the user's user name)

For example, the following JWT is for an internal user whose user name is `aapplegate`. (Information that is not relevant to Cloud API authorization has been omitted.)

```
{
    "scp": [
        "pc_username"
    ],
    "pc_username": "aapplegate"
}
```

Note the following:

- Based on the `scp` token claim, this caller's resource access ID will be interpreted as a user name.
- Based on the `pc_username` token claim, this caller will have access to information related to what the user `aapplegate` can access.

Logging

For each call, information about the caller is logged. The following table lists the fields that provide information about who the caller is, and where the logged value comes from.

Field	Value
`sub`	The value of the `sub` token claim from the JWT
`clientId`	The value of the `cid` token claim from the JWT
`user`	The user name of the internal user

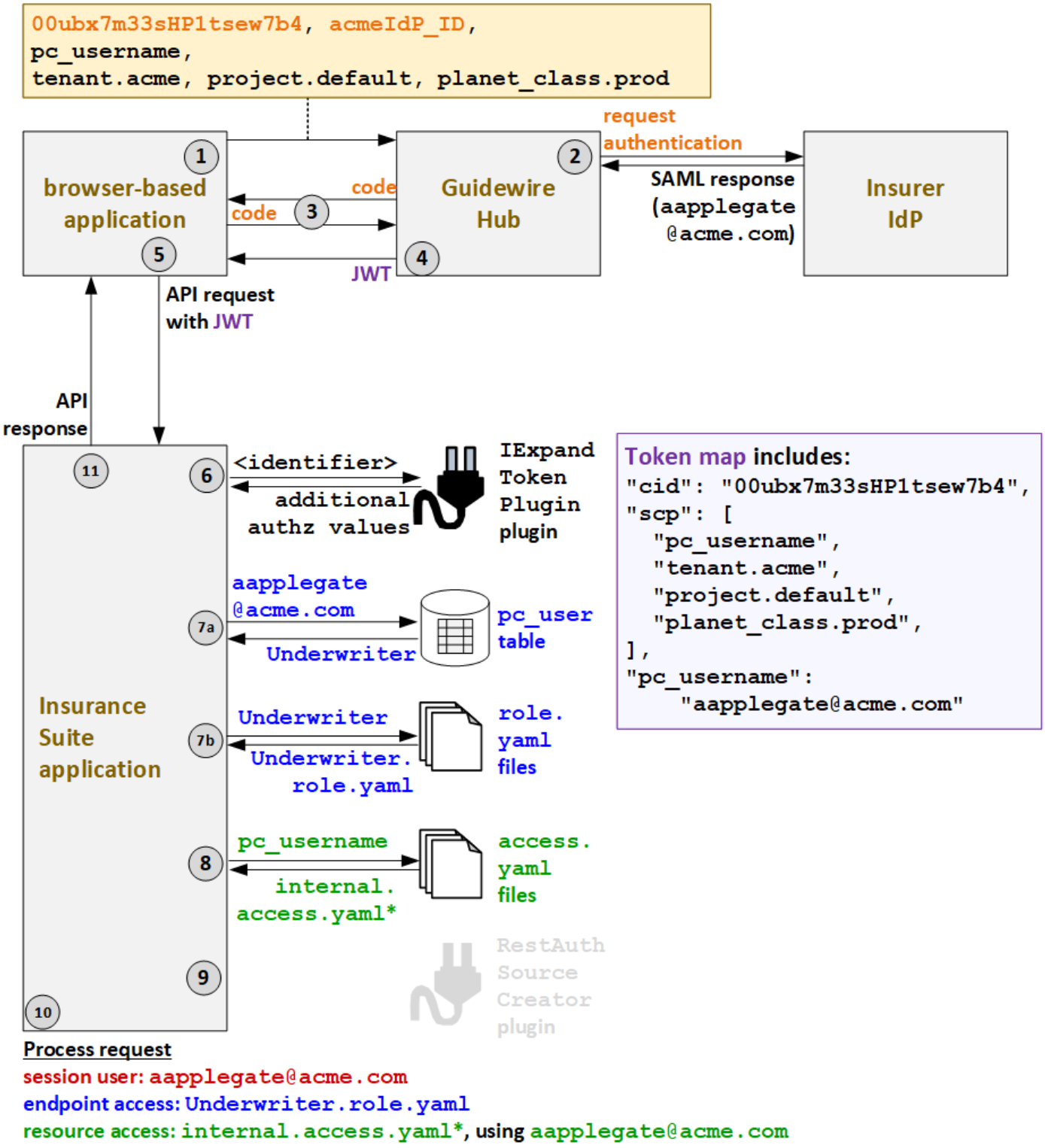
Example flow for internal users

The following diagram identifies the flow of authentication and authorization information for internal users. Colors are used in the following ways:

- Orange - credentials information
- Blue - endpoint access information
- Green - resource access information
- Red - proxy user and session user information

Some values are used to determine multiple types of access. These values initially appear as black (when they do not apply to a single type of access), and then later appear in one or more specific colors (to reflect the value is being used at that point in the process for a specific type of access).

In the following example, an API call is triggered by Alice Applegate, who is an internal user, using a browser-based application.



1. When Alice triggers an API call, the caller application must first request a JWT from Guidewire Hub. To initiate the process of getting the JWT, the caller application submits its client ID (`00ubx7m33sHP1tsew7b4`), the ID of the IdP (`acmeIdP_ID`), the application's resource access strategy (`pc_username`), and additional deployment information (`tenant.acme`, `project.default`, `planet_class.prod`).
2. Guidewire Hub sends a request to the appropriate IdP to authenticate the user. The IdP authenticates the user and provides a SAML response with information about the user, such as the user's name (`aapplegate@acme.com`).

3. Guidewire Hub sends a code to the caller application. The caller application uses this code to request a JWT.
4. Guidewire Hub generates a JWT and sends it to the caller application. This JWT includes the client ID (`cid`), a `scp` token claim which names the resource access strategy (`pc_username`) and additional deployment information. The JWT also contains any relevant information Guidewire Hub received in the SAML response, such as a `pc_username` token which names the user's resource access ID (`aapplegate@acme.com`).
5. The caller application sends the API request to PolicyCenter along with the JWT.
6. PolicyCenter extracts the information in the JWT into a token map. Then, the IExpandTokenPlugin plugin calls any relevant authorization applications to retrieve any relevant additional auth values that must be added to or modified in the token map. (For internal users, the only use case would be if the user's username was not stored in the IdP, and therefore not included in the JWT. The plugin could retrieve the user's username from the appropriate system of record.)
7. PolicyCenter determines the endpoint access.
   a. Using the user name in the token map (`aapplegate@acme.com`), PolicyCenter queries for the user roles that this user has. One role is returned: `Underwriter`.
   b. Based on the returned role, the `Underwriter.role.yaml` API role file is used to define the endpoint access.
8. Next, PolicyCenter determines the resource access strategy. Based on the resource access strategy value in the token map (`pc_username`), it grants resource access as defined in the `internal access.yaml` files. (* PolicyCenter starts with `internal_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with "`internal`".)
9. Proxy user access is not relevant for internal users.
10. PolicyCenter processes the request.
    a. The session user is the internal user: `aapplegate@acme.com`.
    b. The endpoint access is defined by `Underwriter.role.yaml`.
    c. The resource access is defined by `internal access.yaml` using the resource access ID of `aapplegate@acme.com`.
11. PolicyCenter provides the response to the initial call.

Implementation checklist for internal users

To configure Cloud API for authentication for internal users (using bearer token authentication), you may need to do the following tasks:

Task	More Information
Enable asymmetric encryption	"Enabling bearer token authentication" on page 369
Provide deployment information	"Enabling bearer token authentication" on page 369
Configure the IdP to store user information	"Enabling bearer token authentication" on page 369
Register the caller application with Guidewire Hub	"Enabling bearer token authentication" on page 369
Create or modify API roles	"Endpoint access" on page 373
Review the resource access provided by the `pc_username` resource access strategy	"Resource access" on page 385
Configure the IExpandTokenPlugin plugin to retrieve additional authorization values, if needed	"Configuring the IExpandTokenPlugin plugin" on page 395

To make a Cloud API call for internal users (using bearer token authentication), the caller application must:

1. Request a code from Guidewire Hub
2. Use the code to request a JWT from Guidewire Hub

3. Include the JWT with the system API call

For more information, see “Sending authenticated calls for internal users” on page 302.

Sending authenticated calls for internal users

When a caller application wants to make a Cloud API call for an internal user using bearer token authentication, the caller application must:

1. Request an authorization code from Guidewire Hub
2. Use the code to request a JWT from Guidewire Hub
3. Include the JWT with the system API call

Requesting codes and JWTs from Guidewire Hub

For more information on how to request codes and JWTs from Guidewire Hub, see *Authentication*.

Including JWTs with API calls

Once a JWT has been received from Guidewire Hub, it must be sent to PolicyCenter in the request object's Authorization header. The header must use this format:

```
Authorization: Bearer <token>
```

Authentication failure error messages

For endpoints that return elements, when a given resource exists but the user lacks authorization to access it, Cloud API throws the following user message. This is the same message that is returned when the resource does not exist.

```
"status": 404,
        "errorCode": "gw.api.rest.exceptions.NotFoundException",
        "userMessage": "No resource was found at path <path>"
```

For endpoints that return collections, Cloud API returns all resources that meet the criteria and for which the user has sufficient resource access. If a resource exists, but the user lacks sufficient authorization, Cloud API omits it from the results.

These approaches are considered to be more secure as they prevent malicious callers from being able to verify the existence of data that they are not authorized to access.

chapter 51

OAuth2 authorization code flow: External users

Within the context of system API authentication, an *external user* is a person who is known to the insurer but who is not listed as a user in the PolicyCenter database. For PolicyCenter, these are the types of external users:

- **Account holders** - Insureds who want to interact with information about their accounts and policies. For example, Ray Newton, who is a policyholder and wants to verify what coverage he has.
- **Producers** - Producers who want to view or edit accounts, jobs, or policies for a policyholder. For example, Karen Egerston, who is a producer for Ray Newton's policy and wants to update his personal auto policy.

This topic describes how to implement Cloud API authentication for external users.

Overview of authentication for external users

Authentication includes credentials and authorization. Authentication information for external users is specified in JWTs, and information from these JWTs is recorded in the logs.

Credentials

An external user's credentials consist of a user name and password. This information is stored in the IdP.

Before an external user can make an API call, the caller application sends a request to the appropriate IdP to authenticate the user. This typically consists of confirming that the provided username and password are correct.

For more information on how to configure the IdP, see “Configuring the IdP” on page 370.

Authorization

Endpoint access for external users

Endpoint access defines the aspects of an endpoint's behaviors that are available to a caller. This includes:

- What endpoints and resource types are available to the caller?
- What operations can a caller call on the available endpoint?
- What fields can the caller specify in a request payload or get in a response payload?

Endpoint access is controlled by API roles. An *API role* is a list of endpoints, operations, and fields that are available to a set of callers through API calls. API roles act as allowlists. By default, a caller has no endpoint access. When the

caller is associated with one or more API roles, they gain access to the endpoints, operations, and fields allowlisted in each of those API roles.

When an external user makes a Cloud API call, the call has an associated token map. This consists of information stored in the JWT that came with the call, as well as any information added to the map by the IExpandTokenPlugin plugin. The token map includes a list of one or more API roles. The user is given endpoint access to all API roles whose names correspond to the roles listed in the token map. For example, suppose that Ray Newton is a policyholder. Ray Newton triggers a Cloud API call. The token map identifies the Insured role. Ray Newton is given the endpoint access defined in the API role named "Insured".

For more information on how API roles are configured, see “Endpoint access” on page 373.

Resource access for external users

Resource access defines, for a given type of resource, which instances of that resource the caller can access. For example, suppose there is a GET `/claims` endpoint that is available to policyholders, underwriters, adjusters, and service vendors. All of these callers can use the endpoint to access resources whose type is `claim`, but none of the callers can access all of the claims. For example:

- A policyholder may be able to see only the claims associated with the policies they hold.
- An underwriter may be able to see only the claims for policies assigned to them.
- An adjuster may be able to see only the claims assigned to them.
- A service vendor may be able to see only the claims that have a service request assigned to them.

A *resource access strategy* is a set of logic that identifies the meaning of a resource access ID. The base configuration includes the following resource access strategies for external users:

Strategy name	Persona using this strategy	The resource access ID is assumed to be...	Grants access to...
`pc_accountNumbers`	Account holders	An array of account numbers	Information associated with the account
`pc_producerCodes`	Producers	An array of producer codes and roles	Information associated with the account, jobs, and policies

When an external user makes a Cloud API call, PolicyCenter checks for a resource access token claim.

- If the resource access token is `pc_accountNumbers`, the resource access IDs are treated as a list of account numbers. The user is given access to all accounts with these numbers.
- If the resource access token is `pc_producerCodes`, the resource access IDs are treated as one or more pair of producer codes and roles (*`producerCode`*`|`*`role`*). The user is given access to account, job, and policy APIs where the producer code in the token matches the producer code for the account, job, or policy. And then, the role determines the types of permissions the producers have on the resources.

Cloud API requires that the token map have no more than one resource access strategy token.

- If no resource strategy token is present, the caller is assigned the "default" resource access strategy. In the base configuration, this resource strategy grants access to metadata endpoints only.
- If multiple resource strategy tokens are present, the call is rejected.

For more information on how resource access behaves, see “Resource access” on page 385.

Proxy user access for external users

When a caller makes a Cloud API call, the internal PolicyCenter logic may trigger checks that are unrelated to endpoint access or resource access. For example:

- A caller may attempt to assign an activity to themselves. PolicyCenter must check to see if the caller has sufficient permission to own an activity.
- A caller may attempt to create a collision coverage with a deductible less than $1000. PolicyCenter must check to see if the amount of the coverage term is within the caller's authority limit.

External users are not listed in the PolicyCenter operational database, and therefore do not have any system permissions or authority limits tied to them. To execute these checks, Cloud API makes use of proxy users. A *proxy user* is an internal user that is assigned to an external user or service when the API call is made. Whenever internal PolicyCenter logic must check to see if the caller has sufficient access, the proxy user is checked.

For more information on how proxy user access behaves, see “Proxy user access” on page 391.

JWTs for external users

JSON Web Tokens (JWTs) contain token claims. (In standard JWT parlance, these are referred to simply as "claims". To avoid confusion with claims in the property and casualty insurance sense, this documentation sometimes refers to JWT claims as "token claims".) A *token claim* is a piece of information asserted about the bearer of the token, such as the bearer's name. For bearer token authentication, authentication information is stored in token claims.

JWTs for external users can include the following token claims:

- `groups` - The API roles to assign to the external user.
  - In this token claim, the group name is prefixed by "`gwa.<planetclass>.<xc>.`", where *`<planetclass>`* is set to either "`prod`", "`preprod`", or "`lower`", and where <*xc*> is the application code (such as "cc" or "pc").
- `scp` - The resource access strategy to apply to the resource access IDs
- `pc_accountNumbers` - The resource access IDs (when `scp` includes `pc_accountNumbers`)
- `pc_producerCodes` - The resource access IDs (when `scp` includes `pc_producerCodes`)

For example, the following JWT is for an external user who is a policyholder with two accounts: C000456352 and C000456377. (Information that is not relevant to Cloud API authorization has been omitted.)

```
{
    "groups" : [
        "gwa.prod.pc.Account_Holder"
    ],
    "scp": [
        "pc_accountNumbers"
    ],
    "pc_accountNumbers": [
        "C000456352",
        "C000456377"
    ]
}
```

Note the following:

- Based on the `groups` token claim, this caller will be given endpoint access as defined in the role named "Account_Holder".
- Based on the `scp` token claim, this caller's resource access IDs will be interpreted as account numbers.
- Based on the `pc_accountNumbers` token claim, this caller will have access to information related to account C000456352 and account C000456377.

As a second example, the following JWT is for an external user who is a producer. (Information that is not relevant to Cloud API authorization has been omitted.)

```
{
    "scp": [
        "pc_producerCodes",
        "groups"
    ],
    "pc_producerCodes": [
        "ProducerABC|Producer"
    ],
    "groups": [
        "gwa.lower.pc.External Producer Code"
    ]
}
```

Note the following:

- Based on the `groups` token claim, this caller will be given endpoint access as defined in the role named "External Producer Code".
- Based on the `scp` token claim, this caller's resource access IDs will be interpreted as producer codes and roles.
- Based on the `pc_producerCodes` token claim, this caller will have access to information related to producer ProducerABC and role Producer. The caller is given access to account, job, and policy APIs where ProducerABC is the producer code for the account, job, or policy. And, the caller will have Producer permissions on the resources.

Logging

For each call, information about the caller is logged. The following table lists the fields that provide information about who the caller is, and where the logged value comes from.

Field	Value
`sub`	The value of the `sub` token claim from the JWT
`clientId`	The value of the `cid` token claim from the JWT
`user`	The user name of the external user

Example flow for external users

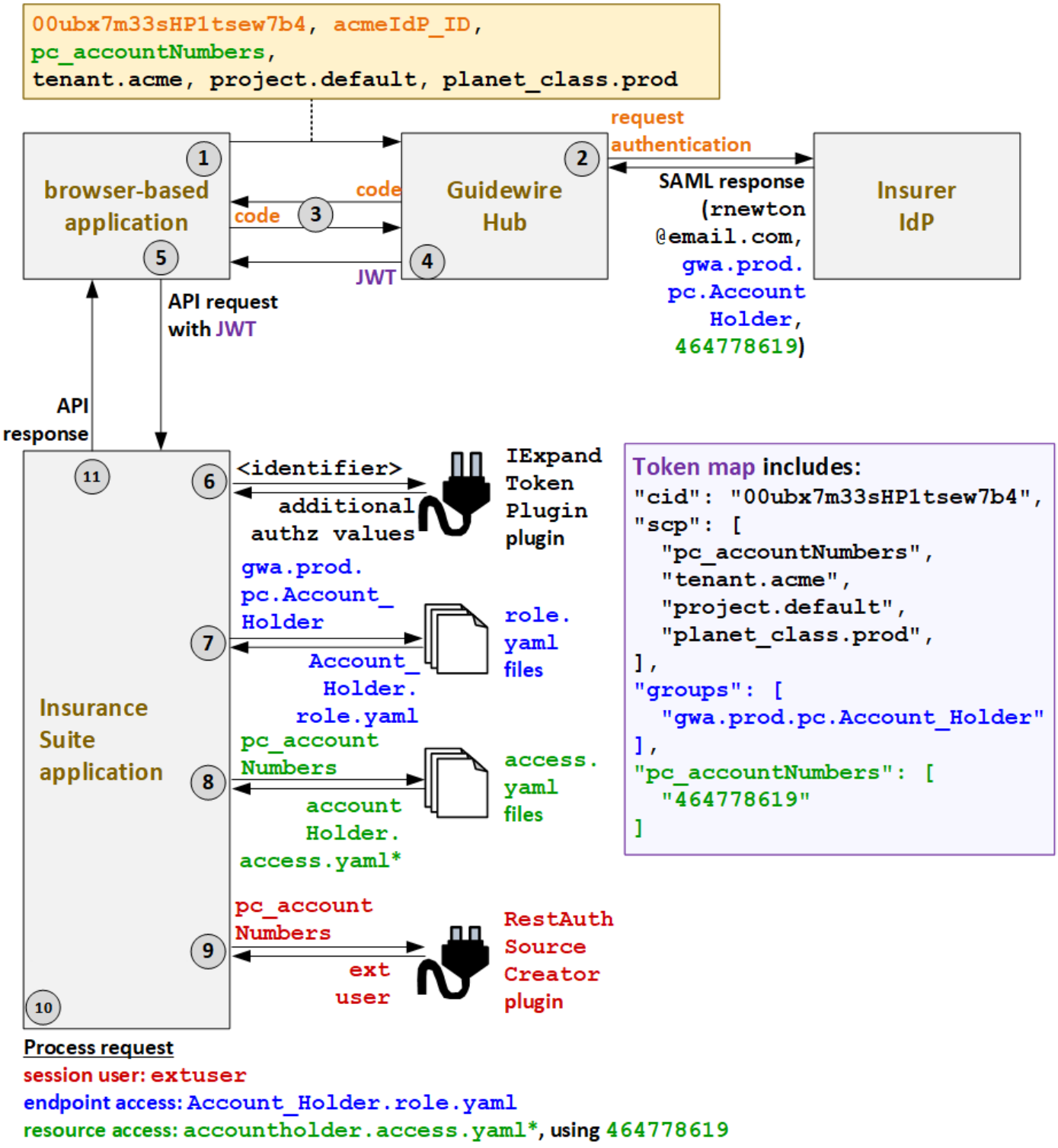
The following diagram identifies the flow of authentication and authorization information for external users. Colors are used in the following ways:

- Orange - credentials information
- Blue - endpoint access information
- Green - resource access information
- Red - proxy user and session user information

Some values are used to determine multiple types of access. These values initially appear as black (when they do not apply to a single type of access), and then later appear in one or more specific colors (to reflect the value is being used at that point in the process for a specific type of access).

External account holder example

In the following example, an API call is triggered by Ray Newton, who is an external user, using a browser-based application.



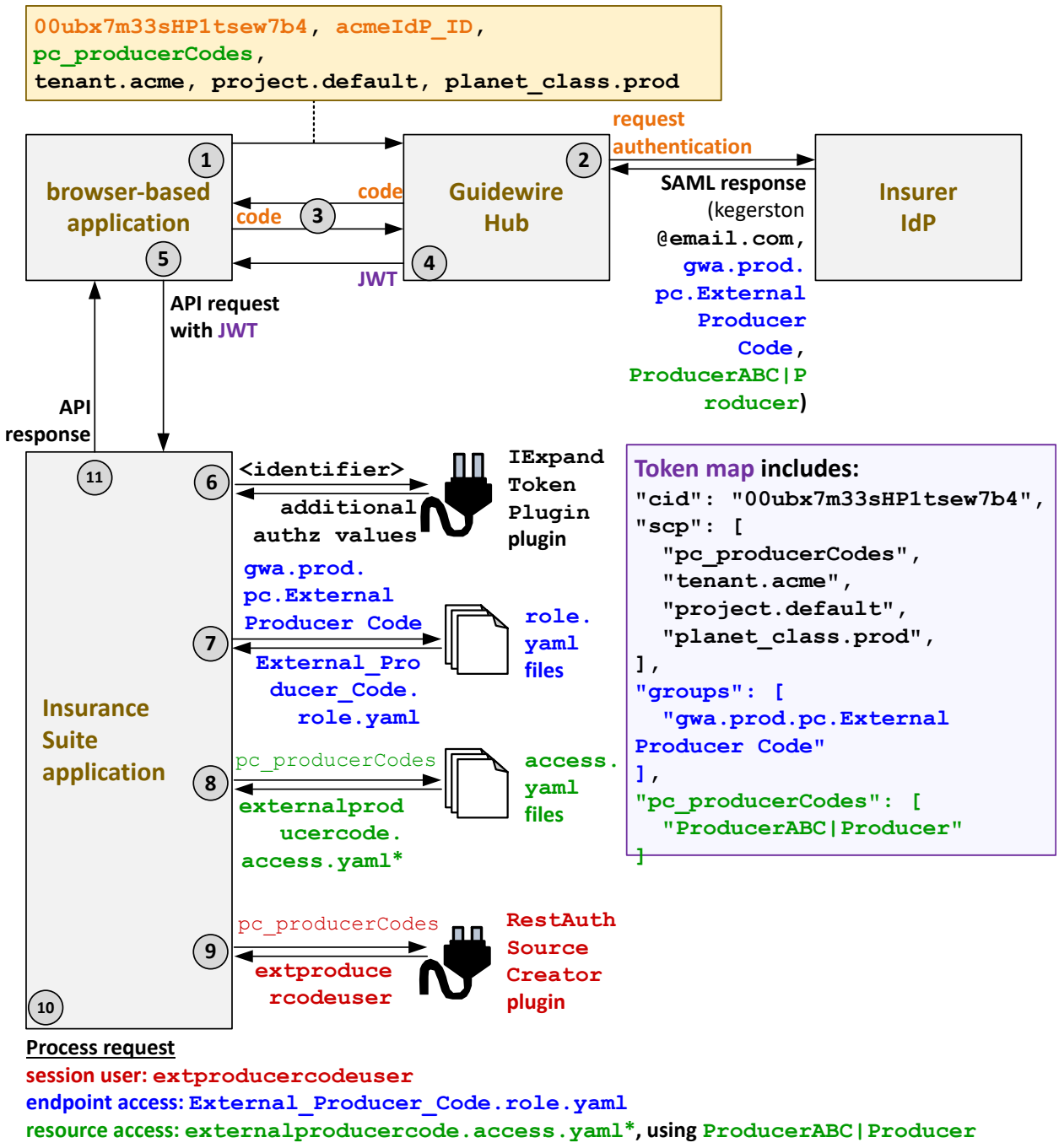
1. When Ray triggers an API call, the caller application must first request a JWT from Guidewire Hub. To initiate the process of getting the JWT, the caller application submits its client ID (`00ubx7m33sHP1tsew7b4`), the ID of the IdP (`acmeIdP_ID`), the application's resource access strategy (`pc_accountNumbers`), and additional deployment information (`tenant.acme`, `project.default`, `planet_class.prod`).
2. Guidewire Hub sends a request to the appropriate IdP to authenticate the user. The IdP authenticates the user and provides a SAML response with information about the user, such as the user's name (`rnewton@email.com`). If

API roles and/or resource access IDs are stored in the IdP, the SAML response may also include this information (such as the role `gwa.prod.pc.Account_Holder` or the resource access ID `464778619`).

3. Guidewire Hub sends a code to the caller application. The caller application uses this code to request a JWT.
4. Guidewire Hub generates a JWT and sends it to the caller application. This JWT includes the client ID (`cid`), a `scp` token claim which names the resource access strategy (`pc_accountNumbers`) and additional deployment information. The JWT also contains any relevant information Guidewire Hub received in the SAML response, such as a `groups` token which names the user's API roles (`gwa.prod.pc.Account_Holder`), or a `pc_accountNumbers` token which names the user's resource access IDs (`464778619`).
5. The caller application sends the API request to PolicyCenter along with the JWT.
6. PolicyCenter extracts the information in the JWT into a token map. Then, the IExpandTokenPlugin plugin calls any relevant authorization applications to retrieve any relevant additional auth values that must be added to or modified in the token map. (For external users, this could include API roles and/or resource access IDs that are not stored in the IdP.)
7. PolicyCenter determines the endpoint access. Based on the groups listed in the token map (`gwa.prod.pc.Account_Holder`), the `Account_Holder.role.yaml` API role file is used to define the endpoint access.
8. Next, PolicyCenter determines the resource access strategy. Based on the resource access strategy value in the token map (`pc_accountNumbers`), it grants resource access as defined in the `accountholder access.yaml` files. (* PolicyCenter starts with `accountholder_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with "`accountholder`".)
9. To determine which proxy user to assign to the session, PolicyCenter calls the `RestAuthenticationSourceCreator` plugin. The token map specified a resource access strategy of `pc_accountNumbers`. So, the plugin returns the proxy user for external users: `extuser`.
10. PolicyCenter processes the request.
    a. The session user is the proxy external user: `extuser`.
    b. The endpoint access is defined by `Account_Holder.role.yaml`.
    c. The resource access is defined by `accountholder access.yaml` using the resource access ID of `464778619`.
11. PolicyCenter provides the response to the initial call.

External producer example

In the following example, an API call is triggered by Karen Egerston, who is an external producer user, using a browser-based application.



1. When Karen triggers an API call, the caller application must first request a JWT from Guidewire Hub. To initiate the process of getting the JWT, the caller application submits its client ID (`00ubx7m33sHP1tsew7b4`), the ID of the IdP (`acmeIdP_ID`), the application's resource access strategy (`pc_producerCodes`), and additional deployment information (`tenant.acme`, `project.default`, `planet_class.prod`).
2. Guidewire Hub sends a request to the appropriate IdP to authenticate the user. The IdP authenticates the user and provides a SAML response with information about the user, such as the user's name (`kegerston@email.com`). If API roles and/or resource access IDs are stored in the IdP, the SAML response may also include this information (such as the role `gwa.prod.pc.External Producer Code` or the resource access ID `ProducerABC|Producer`).

3. Guidewire Hub sends a code to the caller application. The caller application uses this code to request a JWT.
4. Guidewire Hub generates a JWT and sends it to the caller application. This JWT includes the client ID (`cid`), a `scp` token claim which names the resource access strategy (`pc_producerCodes`) and additional deployment information. The JWT also contains any relevant information Guidewire Hub received in the SAML response, such as a `groups` token which names the user's API roles (`gwa.prod.pc.External Producer Code`), or a `pc_producerCodes` token which names the user's resource access IDs (`ProducerABC|Producer`).
5. The caller application sends the API request to PolicyCenter along with the JWT.
6. PolicyCenter extracts the information in the JWT into a token map. Then, the IExpandTokenPlugin plugin calls any relevant authorization applications to retrieve any relevant additional auth values that must be added to or modified in the token map. (For external users, this could include API roles and/or resource access IDs that are not stored in the IdP.)
7. PolicyCenter determines the endpoint access. Based on the groups listed in the token map (`gwa.prod.pc.External Producer Code`), the `External_Producer_Code.role.yaml` API role file is used to define the endpoint access.
8. Next, PolicyCenter determines the resource access strategy. Based on the resource access strategy value in the token map (`pc_producerCodes`), it grants resource access as defined in the `externalproducercode access.yaml` files. (* PolicyCenter starts with `externalproducercode_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with "`externalproducercode`".)
9. To determine which proxy user to assign to the session, PolicyCenter calls the `RestAuthenticationSourceCreator` plugin. The token map specified a resource access strategy of `pc_producerCodes`. So, the plugin returns the proxy user for external producer code users: `extproducercodeuser`.
10. PolicyCenter processes the request.
    a. The session user is the proxy external user: `extproducercodeuser`.
    b. The endpoint access is defined by `External_Producer_Code.role.yaml`.
    c. The resource access is defined by `externalproducercode access.yaml` using the resource access ID of `ProducerABC|Producer`.
    d. For access to internal or sensitive notes and documents associated with the accessible resources, permissions are optionally defined in `External_Producer_Code.role.yaml`. For more information, see "API role special permissions" on page 377
11. PolicyCenter provides the response to the initial call.

Implementation checklist for external users

To configure the system APIs for authentication for external users, you may need to do the following tasks:

Task	More Information
Enable asymmetric encryption	"Enabling bearer token authentication" on page 369
Provide deployment information	"Enabling bearer token authentication" on page 369
Configure the IdP to store user information	"Enabling bearer token authentication" on page 369
Register the caller application with Guidewire Hub	"Enabling bearer token authentication" on page 369
Create or modify API roles	"Endpoint access" on page 373
Review the resource access provided by the `pc_accountNumbers` resource strategy for account holders or the `pc_producerCodes` resource strategy for producers	"Resource access" on page 385
Configure the proxy user	"Proxy user access" on page 391

Task	More Information
Configure the `IExpandTokenPlugin` plugin to retrieve additional authorization values, if needed	"Configuring the IExpandTokenPlugin plugin" on page 395
Configure the `RestV1JobTypesConfigurationPlugin` plugin to provide account holders access to additional job types, if needed	"Configuring access to job types for external account holders" on page 365

To make a Cloud API call for external users, the caller application must:

1. Request a code from Guidewire Hub
2. Use the code to request a JWT from Guidewire Hub
3. Include the JWT with the system API call

For more information, see "Sending authenticated calls for external users" on page 311.

Sending authenticated calls for external users

When a caller application wants to make a system API call for an external user, the caller application must:

1. Request an authorization code from Guidewire Hub
2. Use the code to request a JWT from Guidewire Hub
3. Include the JWT with the system API call

Requesting codes and JWTs from Guidewire Hub

For more information on how to request codes and JWTs from Guidewire Hub, refer to *Authentication with Guidewire Identity Federation Hub* in the *Guidewire Cloud Platform* documentation set.

Including JWTs with API calls

Once a JWT has been received from Guidewire Hub, it must be sent to PolicyCenter in the request object's Authorization header. The header must use this format:

```
Authorization: Bearer <token>
```

Authentication failure error messages

For endpoints that return elements, when a given resource exists but the user lacks authorization to access it, Cloud API throws the following user message. This is the same message that is returned when the resource does not exist.

```
"status": 404,
        "errorCode": "gw.api.rest.exceptions.NotFoundException",
        "userMessage": "No resource was found at path <path>"
```

For endpoints that return collections, Cloud API returns all resources that meet the criteria and for which the user has sufficient resource access. If a resource exists, but the user lacks sufficient authorization, Cloud API omits it from the results.

These approaches are considered to be more secure as they prevent malicious callers from being able to verify the existence of data that they are not authorized to access.

chapter 52

OAuth2 authorization code flow: Anonymous users

Within the context of Cloud API authentication, an *anonymous user* is a person who is not yet known to the insurer but who may establish a business relationship with the insurer. Typically, an anonymous user can only create an account (and its associated objects), quote a submission, and bind a submission. Once an anonymous user binds a submission, they logically move from being an anonymous user to an external user.

PolicyCenter is the only InsuranceSuite application that supports anonymous users for Cloud API.

This topic describes how to implement Cloud API authentication for anonymous users.

Overview of authentication for anonymous users

Life cycle of an anonymous user

From a technical perspective, an anonymous user starts out as an unauthenticated user who creates an account using the POST `/account/v1/accounts` endpoint. PolicyCenter creates a self-signed JWT, which it sends with the response to the API call. The caller application can save this JWT and use it for later calls in the same session. During those later calls, the user is considered an anonymous user.

In some situations, a caller may start but not complete a submission in the first session. They wish to continue the work in some subsequent session. The second session could occur because the first session timed out, or the caller switched to a different device. The caller can execute a POST `/recover-new-jobs` endpoint to recover any incomplete job associated with the account and a new self-signed JWT. Note that, in the base configuration, this endpoint returns no results and no JWT. The insurer must configure the endpoint. For more information, see “Configuring the reauthorize anonymous user flow” on page 403.

If the user binds a policy, either in the first session or a subsequent session, then presumably the user's information is sent to the IdP. On subsequent visits, the user will be an external user. (Note that most JWTs used in bearer token authentication come from Guidewire Hub. The anonymous user flow is the one flow that uses JWTs made from somewhere other than Guidewire Hub.)

Credentials

By definition, an anonymous user does not initially have any credentials. If the user binds a policy, they logically become an external user. At this point, the user's user name and password would be stored in the IdP.

For more information on how configuration of the IdP, see “Configuring the IdP” on page 370.

Authorization

Endpoint access for anonymous users

Endpoint access defines the aspects of an endpoint's behaviors that are available to a caller. This includes:

- What endpoints and resource types are available to the caller?
- What operations can a caller call on the available endpoint?
- What fields can the caller specify in a request payload or get in a response payload?

Endpoint access is controlled by API roles. An *API role* is a list of endpoints, operations, and fields that are available to a set of callers through API calls. API roles act as allowlists. By default, a caller has no endpoint access. When the caller is associated with one or more API roles, they gain access to the endpoints, operations, and fields allowlisted in each of those API roles.

In the anonymous flow, when the caller makes their first Cloud API call, Cloud API automatically assigns them to the unauthenticated role. This role gives them access to endpoints for creating an account and its child objects. Once the account has been created, when the caller makes additional Cloud API calls, Cloud API automatically assigns them to the anonymous role.

For more information on how API roles are configured, see “Endpoint access” on page 373.

Resource access for anonymous users

Resource access defines, for a given type of resource, which instances of that resources the caller can access. For example, suppose there is a GET `/claims` endpoint that is available to policyholders, underwriters, adjusters, and service vendors. All of these callers can use the endpoint to access resources whose type is `claim`, but none of the callers can access all of the claims. For example:

- A policyholder may be able to see only the claims associated with the policies they hold.
- An underwriter may be able to see only the claims for policies assigned to them.
- An adjuster may be able to see only the claims assigned to them.
- A service vendor may be able to see only the claims that have a service request assigned to them.

A *resource access strategy* is a set of logic that identifies the meaning of a resource access ID. Anonymous users use the same resource access strategy as that used by external users. The base configuration includes the following resource access strategies for external users:

Strategy name	Persona using this strategy	The resource access ID is assumed to be...	Grants access to...
`pc_accountNumbers`	Account holders (including anonymous users who have created an account)	An array of account numbers	Information associated with the account

For anonymous users, the `pc_accountNumbers` strategy is used automatically. Any resource access ID is treated as a list of account numbers. The user is given access to all accounts with these numbers.

For more information on how resource access behaves, see “Resource access” on page 385.

Proxy user access for anonymous users

When a caller makes a Cloud API call, the internal PolicyCenter logic may trigger checks that are unrelated to endpoint access or resource access. For example:

- A caller may attempt to assign an activity to themselves. PolicyCenter must check to see if the caller has sufficient permission to own an activity.
- A caller may attempt to create a collision coverage with a deductible less than $1000. PolicyCenter must check to see if the amount of the coverage term is within the caller's authority limit.

Anonymous users are not listed in the PolicyCenter operational database, and therefore do not have any system permissions or authority limits tied to them. To execute these checks, Cloud API makes use of proxy users. A *proxy*

user is an internal user that is assigned to an external user or service when the API call is made. Whenever internal PolicyCenter logic must check to see if the caller has sufficient access, the proxy user is checked. Anonymous users are assigned proxy user access as if they were external users.

For more information on how proxy user access behaves, see “Proxy user access” on page 391.

JWTs for anonymous users

JSON Web Tokens (JWTs) contain token claims. (In standard JWT parlance, these are referred to simply as "claims". To avoid confusion with claims in the property and casualty insurance sense, this documentation sometimes refers to JWT claims as "token claims".) A *token claim* is a piece of information asserted about the bearer of the token, such as the bearer's name. For bearer token authentication, authentication information is stored in token claims.

JWTs for anonymous users include the following token claims:

- `groups` - The API roles to assign to the anonymous user
- `scp` - The name of the resource access strategy (`pc_accountNumbers`)
- `pc_accountNumbers` - The resource access IDs (which are treated as account numbers)

For example, the following JWT is for an anonymous user who has created account C000999111. (Information that is not relevant to Cloud API authorization has been omitted.)

```
{
    "groups" : [
        "pc.anonymous"
    ],
    "scp": [
        "pc_accountNumbers"
    ],
    "pc_accountNumbers": [
        "C000999111"
    ]
}
```

Note the following:

- Based on the `groups` token claim, this caller will be given endpoint access as defined in the role named "anonymous".
- Based on the `scp` token claim, this caller's resource access IDs will be interpreted as account numbers.
- Based on the `pc_accountNumbers` token claim, this caller will have access to information related to account C000999111.

Example flow for anonymous users

The following diagram identifies the flow of authentication and authorization information for unauthenticated callers. Colors are used in the following ways:

- Orange - credentials information
- Blue - endpoint access information
- Green - resource access information
- Red - proxy user and session user information

Some values are used to determine multiple types of access. These values initially appear as black (when they do not apply to a single type of access), and then later appear in one or more specific colors (to reflect the value is being used at that point in the process for a specific type of access).

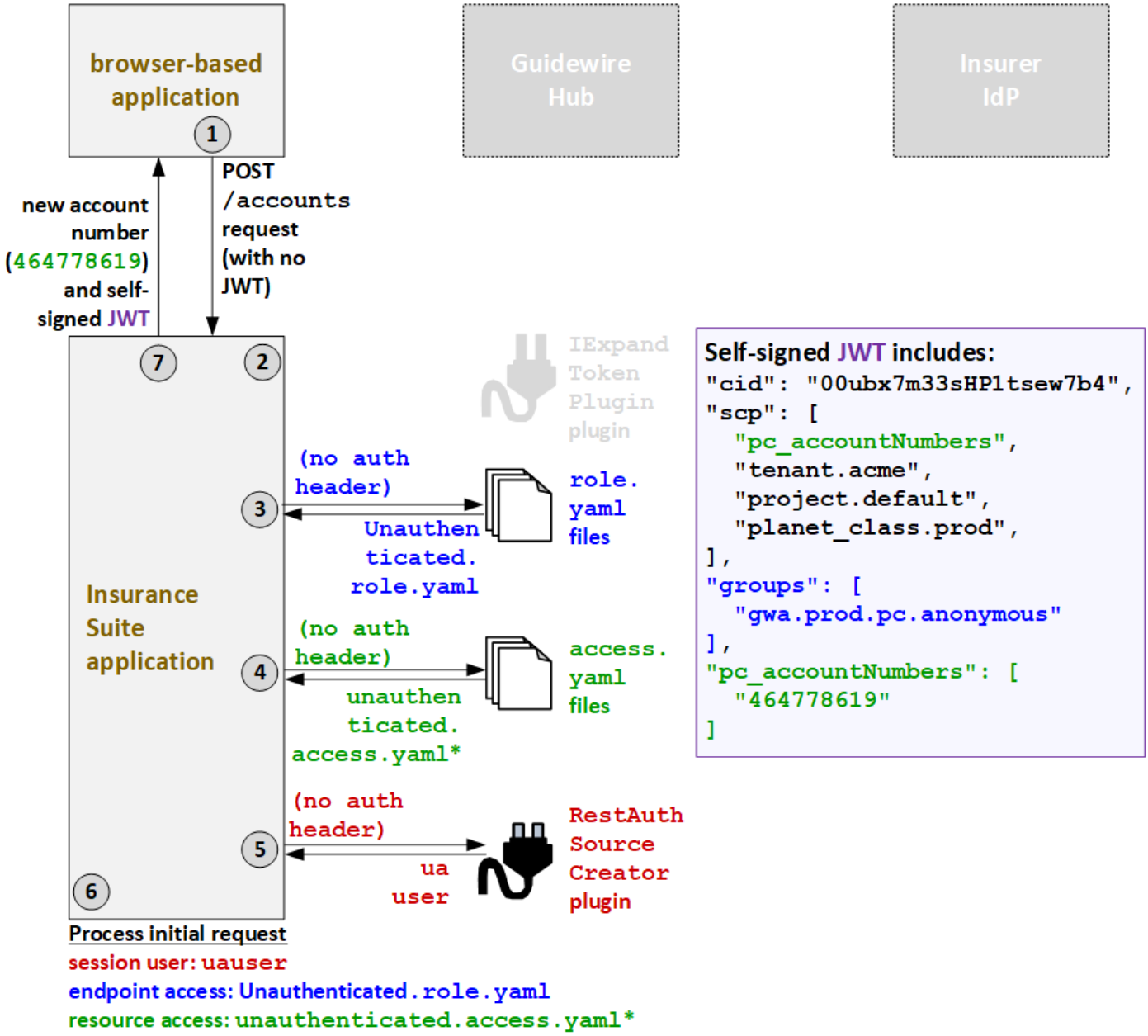
The anonymous user flow always involves multiple calls.

- For the initial call that starts a new session and creates the account, the user is unauthenticated.
- If a call is sent to create a new session to continue work from a previous session, the user is unauthenticated.

- For all other calls in an existing session, the user is anonymous.

The initial call as an unauthenticated user

In the following example, an API call is triggered by an unauthenticated caller who creates an account and later plans to bind a policy submission.

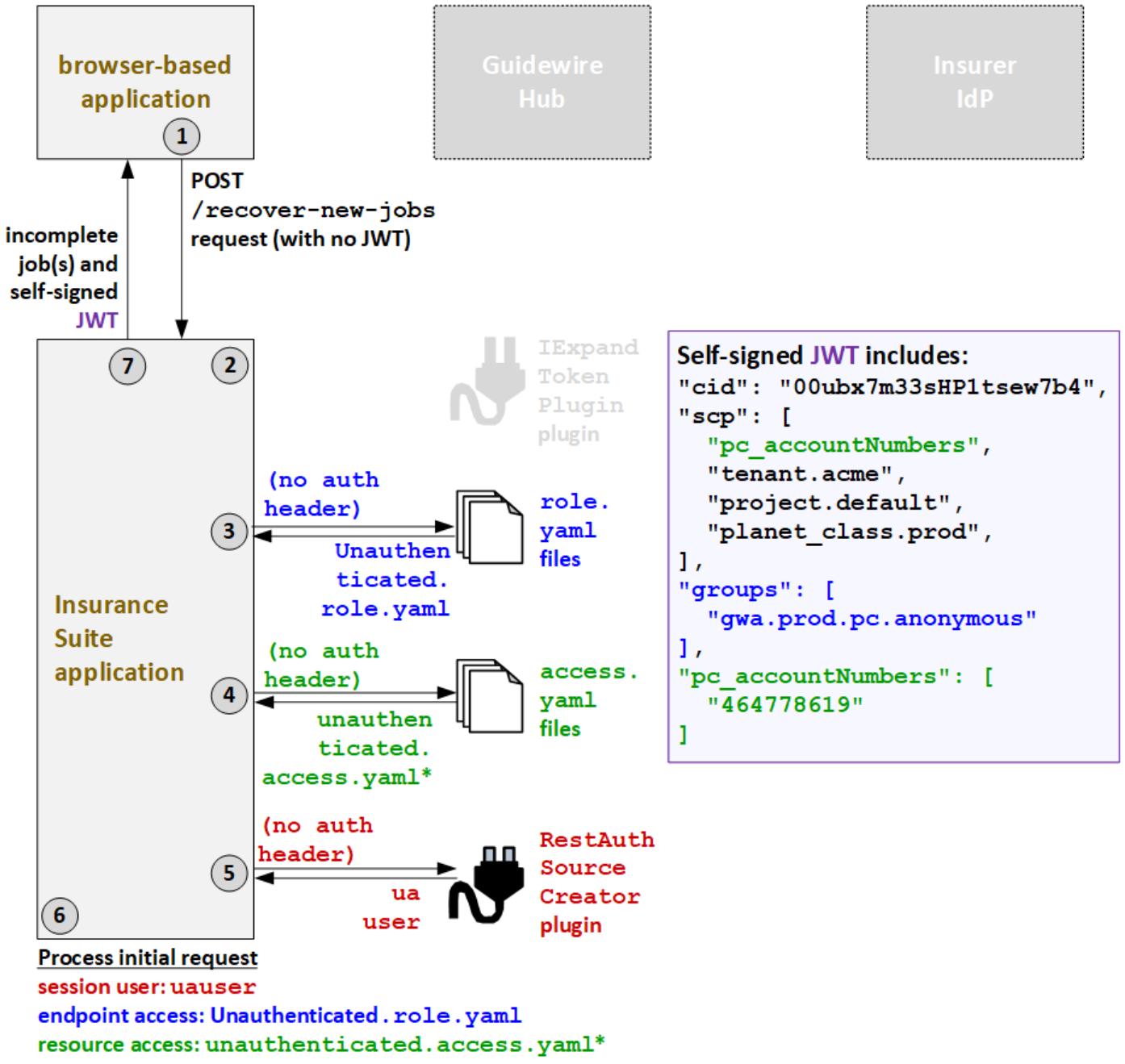


1. The user triggers an API call by creating a new account. The caller application sends the API request to PolicyCenter. The call includes no JWT, and no authentication information in the header.
2. The IExpandTokenPlugin plugin is not relevant for anonymous users.
3. Because the call has no authentication header, PolicyCenter grants endpoint access as defined in the `Unauthenticated.role.yaml` API role file.
4. Because the call has no authentication header, PolicyCenter grants resource access as defined in the `unauthenticatedUser.access.yaml` API role file. (This provides no access to existing business resources.)

5. To determine which proxy user to assign to the session, PolicyCenter calls the `RestAuthenticationSourceCreator` plugin. The call has no authentication header. So, the plugin returns the proxy user for unauthenticated users: `uauser`.
6. PolicyCenter processes the request.
   a. The session user is the proxy unauthenticated user: `uauser`.
   b. The endpoint access is defined by `Unauthenticated.role.yaml`. (This API role provides sufficient endpoint access to create a new account and the required child objects, such as contacts and locations.)
   c. The resource access is defined by `unauthenticatedUser.access.yaml`.
7. PolicyCenter provides the response to the initial call. The response includes the account number for the newly created account (464778619) and a self-signed JWT generated by PolicyCenter. The JWT includes the client ID (`cid`), a `scp` token claim which names the resource access strategy (`pc_accountNumbers`) and additional deployment information, a `groups` token which names the user's groups (`gwa.prod.pc.anonymous`), and a `pc_accountNumbers` token which names the user's resource access IDs (`464778619`).

A subsequent call to reauthorize an unauthenticated user

In the following example, an API call is triggered by an unauthenticated caller who has already created an account and started a submission at some previous point in time.

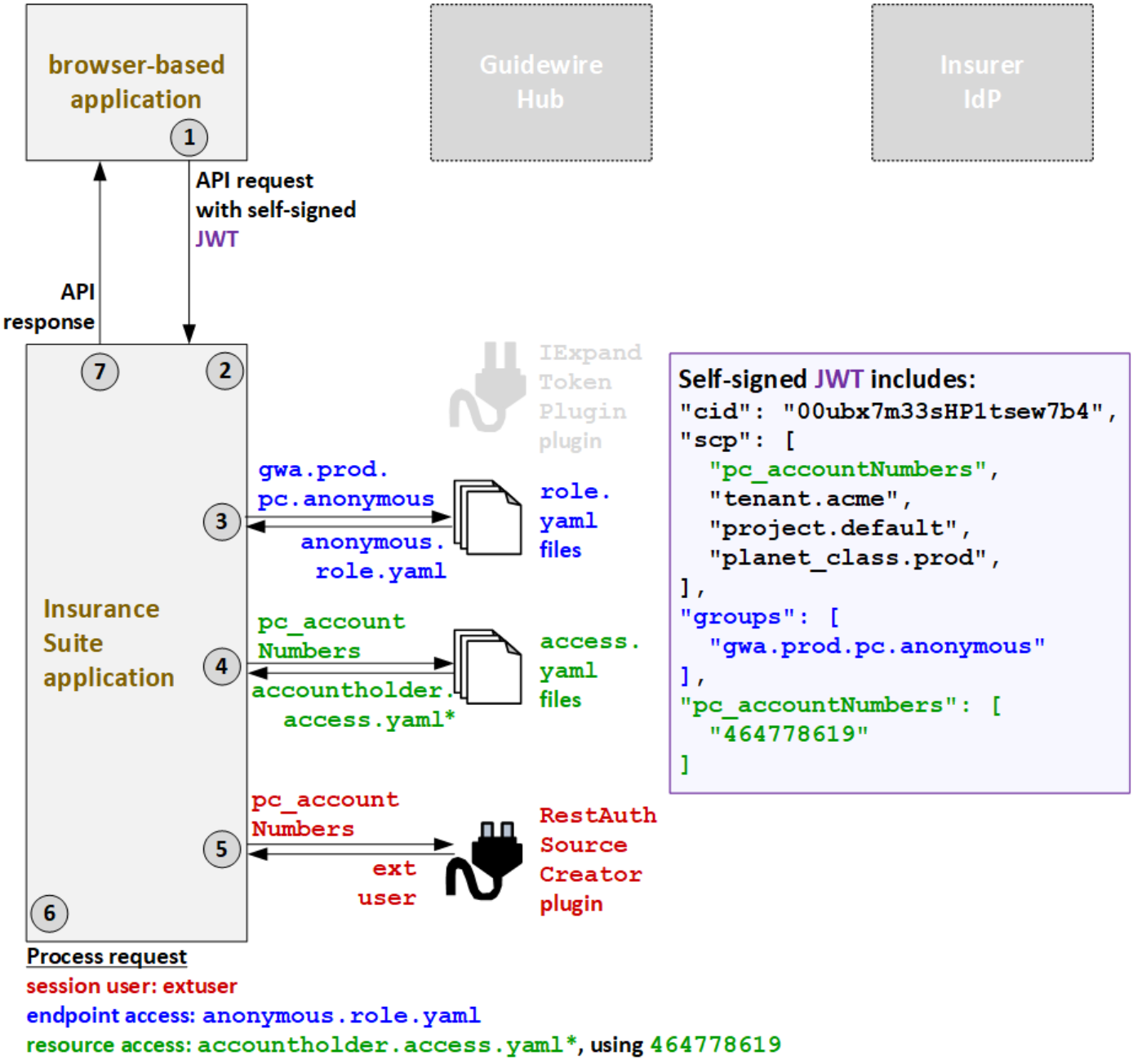


1. The user triggers an API call to complete a previously initiated submission for an existing anonymous user account. The caller application sends the API request to PolicyCenter using the `/recover-new-jobs` endpoint as configured by the insurer. The call includes search criteria configured by the insurer (such as an account number or user name), but it has no JWT and no authentication information in the header.
2. The IExpandTokenPlugin plugin is not relevant for anonymous users.
3. Because the call has no authentication header, PolicyCenter grants endpoint access as defined in the `Unauthenticated.role.yaml` API role file.
4. Because the call has no authentication header, PolicyCenter grants resource access as defined in the `unauthenticatedUser.access.yaml` API role file.

5. To determine which proxy user to assign to the session, PolicyCenter calls the `RestAuthenticationSourceCreator` plugin. The call has no authentication header. So, the plugin returns the proxy user for unauthenticated users: `uauser`.
6. PolicyCenter processes the request.
   a. The session user is the proxy unauthenticated user: `uauser`.
   b. The endpoint access is defined by `Unauthenticated.role.yaml`. (This API role provides sufficient endpoint access to create a new account and the required child objects, such as contacts and locations, and to complete a submission.)
   c. The resource access is defined by `unauthenticatedUser.access.yaml`.
7. PolicyCenter provides the response to the initial call. The response includes the unbound submissions that meet the search criteria provided with the initial call. It also includes a new self-signed JWT generated by PolicyCenter. As was the case with the first self-signed JWT, this JWT includes the client ID (`cid`), a `scp` token claim which names the resource access strategy (`pc_accountNumbers`) and additional deployment information, a `groups` token which names the user's groups (`gwa.prod.pc.anonymous`), and a `pc_accountNumbers` token which names the user's resource access IDs (`464778619`).

The later call as an anonymous user

After a previous call to create an account or recover an unbound submission, another API call is triggered by the previously unauthenticated caller who is now anonymous. This call creates and creates, modifies, or binds a policy submission.



1. The user triggers an API call by attempting to create, modify, or bind a submission. The caller application sends the API request to PolicyCenter. The call includes the self-signed JWT.
2. The IExpandTokenPlugin plugin is not relevant for anonymous users.
3. PolicyCenter determines the endpoint access. Based on the groups listed in the JWT (`gwa.prod.pc.anonymous`), the `anonymous.role.yaml` API role file is used to define the endpoint access.
4. Next, PolicyCenter determines the resource access strategy. Based on the resource access strategy value in the JWT (`pc_accountNumbers`), it grants resource access as defined in the `accountholder.access.yaml` files. (* PolicyCenter starts with `accountholder_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with "`accountholder`".)

5. To determine which proxy user to assign to the session, PolicyCenter calls the `RestAuthenticationSourceCreator` plugin. The JWT specified a resource access strategy of `pc_accountNumbers`. So, the plugin returns the proxy user for external users: `extuser`.
6. PolicyCenter processes the request.
   a. The session user is the proxy external user: `extuser`.
   b. The endpoint access is defined by `anonymous.role.yaml`. (This API role provides access to actions appropriate for anonymous users, such as quoting and binding submissions.)
   c. The resource access is defined by `accountholder.access.yaml`.
7. PolicyCenter provides the response to the call.

If the anonymous user's policy is bound, then the user becomes "known" to the insurer. At some point in the future, the user's information is sent to the IdP. Once this occurs, the user will be treated as an external user.

Implementation checklist for anonymous users

To configure Cloud API for authentication for anonymous users, you may need to do the following tasks:

Task	More Information
Provide deployment information	"Enabling bearer token authentication" on page 369
Register the caller application with Guidewire Hub	"Enabling bearer token authentication" on page 369
Review and modify the `anonymous` API role	"Endpoint access" on page 373
Review the resource access provided by the `pc_accountNumbers` resource access strategy	"Resource access" on page 385
Configure the proxy user	"Proxy user access" on page 391
Configure the `/recover-new-jobs` endpoint	"Configuring the reauthorize anonymous user flow" on page 403

To make a Cloud API call for anonymous users, the caller application must:

1. Create an account as an unauthenticated user, and retain the self-signed JWT that is provided in the response object
2. Include the self-signed JWT with subsequent system API calls

For more information, see "Sending calls as an anonymous user" on page 324.

Creating an account as an unauthenticated user

Unauthenticated users have the ability to create accounts by POSTing to the `/accounts` endpoint.

- Unauthenticated users can also create contacts and locations for the account using POST `/accounts/*/contacts` and POST `/accounts/*/locations`. However, this must be done in a single POST `/accounts` call that uses request inclusion for any contacts and locations. You cannot call the POST `/accounts/*/contacts` or POST `/accounts/*/locations` endpoints on their own.
- For a complete list of the endpoint access available to unauthenticated users, refer to the `Unauthenticated.role.yaml` file in the **Integration** > **roles** directory.
- For more information on how to create accounts using the `/accounts` endpoint, see the *Cloud API Consumer Guide*.

In response to this POST, PolicyCenter sends a response object. The body of the response contains information about the new account, such as the account's account number. The response header contains a `GW-Access-Token` attribute whose value is a self-signed JWT. The caller application must save this JWT so that it can be included in any subsequent call that the caller wants to make for the account that was just created.

For example, suppose there is an unauthenticated user who wants to create an account. The following information is true about this user:

- Name: Bill Presley
- Primary Address: 1234 Hillsdale Blvd, Foster City, CA, 12345
- Producer: Armstrong and Company (whose public ID is "pc:6")

The caller application can create an account for this user by executing a POST `/accounts` with the following request payload:

```
{
  "data": {
    "attributes": {
      "accountHolder": {
        "refid": "newperson"
      },
      "organizationType": {
        "code": "individual"
      },
      "preferredCoverageCurrency": {
        "code": "USD"
      },
      "preferredSettlementCurrency": {
        "code": "USD"
      },
      "primaryLocation": {
        "refid": "newloc"
      },
      "producerCodes": [
        {
          "id": "pc:6"
        }
      ]
    }
  },
  "included": {
    "AccountContact": [
      {
        "attributes": {
          "contactSubtype": "Person",
          "firstName": "Bill",
          "lastName": "Presley",
          "primaryAddress": {
            "addressLine1": "1234 Hillsdale Blvd",
            "city": "Foster City",
            "postalCode": "12345",
            "state": {
              "code": "CA"
            }
          }
        },
        "method": "post",
        "refid": "newperson",
        "uri": "/account/v1/accounts/this/contacts"
      }
    ],
    "AccountLocation": [
      {
        "attributes": {
          "locationCode": "0001",
          "locationName": "Location 0001",
          "nonSpecific": true,
          "postalCode": "12345",
          "state": {
            "code": "CA"
          }
        },
        "method": "post",
        "refid": "newloc",
        "uri": "/account/v1/accounts/this/locations"
      }
    ]
  }
}
```

PolicyCenter creates an account, which in this case is assigned an account number of 2558363138. PolicyCenter also sends a response object. In the response header, the `GW-Access-Token` attribute is set to the following:

```
eyJhbGciOiJIUzUxMiIsImtpZCI6ImN1cnJlbnRfa2V5IiwidHlwIjoiSldUIn0.eyJleHAiOjE1OTU1NjYzNjksImdyb3VwcyI6WyJwYy5hbm9ueW1
vdXMiXSwiaWF0IjoxNTk1NTU1NTY5LCJpc3MiOiJQQyIsImp0aSI6InJCMEVDYVdoOVh1Y2U5M3cyYkFETnVXOUszdkZoUGxuS0FpbVR3NVdFNWNueW
```

```
9VM0FBQUFCQS4uIiwicGNfYWNjb3VudE51bWJlcnMiOlsiMjU1ODM2MzEzOCJdLCJzdWIiOiJhdXRoIiwidGVuYW50X2lkIjoiTm9UZW5hbnQiLCJ0e
XBlIjoiYWNjb3VudEhvbGRlciJ9.Ix4GCz4nJg_QM3AsC-jVyZU_V8ysGBgWfvIxAIS59t7EN2C6Pi2QgJRs09y0ThqFX-_1-ucD58Vunqs5dMivJg
```

The decoded payload for this JWT is:

```
{
  "exp": 1595566369,
  "groups": [
    "pc.anonymous"
  ],
  "iat": 1595555569,
  "iss": "PC",
  "jti": "rB0ECaWh9Xuce93w2bADNuW9K3vFhPlnKAimTw5WE5cnyoU3AAAABA..",
  "pc_accountNumbers": [
    "2558363138"
  ],
  "sub": "auth",
  "tenant_id": "NoTenant",
  "type": "accountHolder"
}
```

Note the following:

- The `groups` token claim includes `pc.anonymous`. On subsequent calls, the bearer of this token will be granted the API role named "anonymous".
- The `pc_accountNumbers` token claim lists 2558363138. On subsequent calls, the bearer of this token will be able to view and create data related to account 2558363138 and any of its related information.

Recovering incomplete submissions as an unauthenticated user

In some situations, a caller may start but not complete a submission in the first session. They wish to continue the work in some subsequent session. The second session could occur because the first session timed out, or the caller switched to a different device. The caller can execute a POST `/recover-new-jobs` endpoint to recover any incomplete job associated with the account and a new self-signed JWT.

> **Note:** In the base configuration, the POST `/recover-new-jobs` endpoint returns no results and no JWT. The insurer must configure the endpoint. For more information, see “Configuring the reauthorize anonymous user flow” on page 403.

Once you have implemented the `/recover-new-jobs` endpoint, you can use it in the same way that you would use a base configuration search endpoint. The request includes a body with the appropriate search criteria. The response includes a collection of jobs meeting the implicit and configured search criteria. If the response includes one or more jobs, the response header also contains a self-signed JWT that can be used for subsequent calls.

For example, suppose you configure the `/recover-new-jobs` endpoint so that an anonymous user can locate incomplete submissions by providing their first and last name. Then, suppose you wanted to find all incomplete submissions where the primary insured's first name is "Francine" and last name is "Michaels". The request would look like this.

```
POST /job/v1/recover-new-jobs

{
  "data": {
    "attributes": {
      "firstName_Ext": "Francine",
      "lastName_Ext": "Michaels"
    }
  }
}
```

Suppose there is one job that matches the search criteria. The response would look like this. For the sake of clarity, some properties have been omitted.

```
RESPONSE BODY:
{
    "count": 1,
    "data": [
        {
```

```
            "attributes": {
                "account": {
                    "displayName": "C000143542",
                    "id": "pc:Srx-YL_xDmdj455LyQtIu",
                    "type": "Account",
                    "uri": "/account/v1/accounts/pc:Srx-YL_xDmdj455LyQtIu"
                },
                "id": "pc:SVAyPWiBiNuBrYFPjd7ZX",
                "jobNumber": "47586734721",
                "jobStatus": {
                    "code": "Quoted",
                    "name": "Quoted"
                },
                "jobType": {
                    "code": "Submission",
                    "name": "Submission"
                },
                "policy": {
                    "displayName": "P000143542",
                    "id": "pc:SH21ByixaZ0O-2Hteuwc8",
                    "type": "Policy",
                    "uri": "/policy/v1/policies/pc:SH21ByixaZ0O-2Hteuwc8"
                },
                "policyNumber": "P000143542",
                "primaryInsured": {
                    "displayName": "Francine Michaels",
                    "id": "test_pp:2",
                    "type": "PolicyContact",
                    "uri": "/job/v1/jobs/pc:SVAyPWiBiNuBrYFPjd7ZX/contacts/test_pp:2"
                }
            },
            ...

RESPONSE HEADER:
GW-Access-Token:
eyJhbGciOiJIUzUxMiIsImtpZCI6ImN1cnJlbnRfa2V5IiwidHlwIjoiSldUIn0.eyJleHAiOjE1OTU1NjYzN
jksImdyb3VwcyI6WyJwYy5hbm9ueW1vdXMiXSwiaWF0IjoxNTk1NTU1NTY5LCJpc3MiOiJQQyIsImp0aSI6In
JCMEVDYVdoOVh1Y2U5M3cyYkFETnVXOUszdkZoUGxuS0FpbVR3NVdFNWNueW9VM0FBQUFCQS4uIiwicGNfYWN
jb3VudE51bWJlcnMiOlsiMjU1ODM2MzEzOCJdLCJzdWIiOiJhdXRoIiwidGVuYW50X2lkIjoiTm9UZW5hbnQi
LCJ0eXBlIjoiYWNjb3VudEhvbGRlciJ9.Ix4GCz4nJg_QM3AsC-jVyZU_V8ysGBgWfvIxAIS59t7EN2C6Pi2Q
gJRs09y0ThqFX-_1-ucD58Vunqs5dMivJg
```

Sending calls as an anonymous user

Once the user has created an account, the caller application must retain the self-signed JWT. The application must provide the JWT with all Cloud API calls for that caller. The JWT must be sent in the Authorization header when making requests to protected resources using this format:

```
Authorization: Bearer <token>
```

Authentication failure error messages

For endpoints that return elements, when a given resource exists but the user lacks authorization to access it, Cloud API throws the following user message. This is the same message that is returned when the resource does not exist.

```
"status": 404,
        "errorCode": "gw.api.rest.exceptions.NotFoundException",
        "userMessage": "No resource was found at path <path>"
```

For endpoints that return collections, Cloud API returns all resources that meet the criteria and for which the user has sufficient resource access. If a resource exists, but the user lacks sufficient authorization, Cloud API omits it from the results.

These approaches are considered to be more secure as they prevent malicious callers from being able to verify the existence of data that they are not authorized to access.

chapter 53

OAuth2 client credential flow: Standalone services

A *service* is an application that typically executes action without human intervention. Services typically have no user interface. Examples of services include:

- A billing application that alerts PolicyCenter about a delinquent policy that needs to be canceled.
- An application that uploads pictures of a covered location or vehicle, either when a policy is bound or after a loss has occurred.
- An external document management system.

This topic discusses how to execute authentication for standalone services.

Authentication options for services

There are several ways a service can execute authentication with Cloud API.

Standalone service

A service can authenticate **as a standalone service**. In this case, the service executes the call as itself. It does not execute the call as a specific person or on behalf of a specific person. The service does not execute the call using a service account stored in PolicyCenter.

PolicyCenter designates a single internal user as the "proxy service user" for all standalone service calls. This proxy service user is attached to the standalone service session. If the call creates or modifies an object, the proxy service user is recorded as the user of record.

The primary advantage to this approach is that you need to manage authentication and authorization information at the service level only. There is no need to create and manage user accounts, user permissions, or additional mappings.

The primary disadvantage is that all standalone service calls share a single proxy service user. When a standalone service call creates or modifies an object, it may not be possible to identify which service made the call.

Service with user context

A service can authenticate **with user context**. In this case, the service presents information about itself. The call also includes a `GW-User-Context` header that provides information about a specific user. The user does not necessarily exist in the PolicyCenter database. The call is able to do only the things that both the service by itself could do and the user by itself could do.

The specified user can be an internal user (a user who is listed in the PolicyCenter database). When this is the case, this internal user is attached to the session. If the call creates or modifies an object, this internal user is recorded as the user of record.

The specified user can be an external user (a user who is not listed in the PolicyCenter database). PolicyCenter designates a single internal user as the "proxy external user" for all service with user context calls that reference external users. When the specified user is an external user, the external proxy user is attached to the session. If the call creates or modifies an object, the external proxy user is recorded as the user of record.

The primary advantage to this approach is that a single service can send calls on behalf of different users. At the service level, you can specify service-level access. But, you can also further control access for each associated user.

There are two primary disadvantages. First, you must maintain access information at two levels: the service level and at the user level. Second, a service can specify any user in its header. There is no way to restrict a given set of users for use by a given service.

Service with service account mapping

A service can authenticate **with service account mapping**. In this case, the service is automatically mapped to a "service account". The mapping information is specified elsewhere in the environment. The service account is a user account in the PolicyCenter database that is intended to be used only by the service and not by any person.

The primary advantage to this approach is that permissions and auditing for the call is tied to a service account that is listed in the PolicyCenter database. You can create a different service account for each service and have fine control over the permissions available to each service.

The primary disadvantage is that you must create and maintain service accounts in PolicyCenter for calls made by external applications. Also, you must maintain the mapping information that maps each service to its service account.

Comparing the different approaches

The following table compares each approach.

	Standalone service	Service with user context	Service with service account mapping
Does the call provide information about the service?	Yes, in the JWT.	Yes, in the JWT.	Yes, in the JWT.
Does there need to be a user account in the PolicyCenter database for the call?	No	If the associated user is an internal user, yes. If the associated user is an external user, no.	Yes. (This user account is the "service account".)
Does the call include information about a user or user account?	No	Yes, in the `GW-User-Context` header.	No. The call provides a client ID for the service, but the mapping of client ID to service account is stored elsewhere.
Which endpoints can the call access?	The endpoints available to the service's API roles	The endpoints available to both the service's API roles and the user's API roles.	The endpoints available to the service account.
Which resources can the call access?	All resources (in the base configuration).	The resources available to both the service and the user.	The resources available to the service account.
What is the session user set to?	The proxy service user.	If the associated user is an internal user, the internal user. If the associated user is an external user, the proxy external user.	The service account.

This topic focuses on authentication for standalone services.

- For more information on authentication for services with user context, see “OAuth2 client credential flow: Services with user context” on page 333.

- For more information on authentication for services with service account mapping, see “OAuth2 client credential flow: Services with service account mapping” on page 349.

Overview of authentication for standalone services

Authentication includes credentials and authorization. Authentication information for services is specified in JWTs, and information from these JWTs is recorded in the logs.

Credentials

When a service makes an API call, the service sends a client ID and secret to Guidewire Hub. Guidewire Hub authenticates the service by confirming that the client secret is correct. This is true for standalone services, services with user context, and services with service account mapping.

For more information on how client IDs and secrets are registered with Guidewire Hub, see “Registering the caller application with Guidewire Hub” on page 372.

Authorization

Endpoint access for standalone services

Endpoint access defines the aspects of an endpoint's behaviors that are available to a caller. This includes:

- What endpoints and resource types are available to the caller?
- What operations can a caller call on the available endpoint?
- What fields can the caller specify in a request payload or get in a response payload?

Endpoint access is controlled by API roles. An *API role* is a list of endpoints, operations, and fields that are available to a set of callers through API calls. API roles act as allowlists. By default, a caller has no endpoint access. When the caller is associated with one or more API roles, they gain access to the endpoints, operations, and fields allowlisted in each of those API roles.

Theoretically, a standalone service can be associated with multiple API roles. Typically, insurers create one API role for each service and this role is used only by this service.

For a standalone service call, Cloud API checks the API role or roles assigned to the service. The call has access to the endpoints, operations, and fields specified in those roles. For example, suppose that the ACME External Document Manager service has the following API role with the following endpoint access:

- `acme_externaldocumentmanager`
  - GET `/documents`
  - POST `/documents`

Then, suppose ACME External Document Manager service makes a standalone service call. The call would have access to GET `/documents` and POST `/documents`. But if there was a DELETE `/documents` endpoint, the call would not have access to it because it has not been specified in the `acme_externaldocumentmanager` role.

For more information on how API roles are configured, see “Endpoint access” on page 373.

Resource access for standalone services

Resource access defines, for a given type of resource, which instances of that resource the caller can access. For example, suppose there is a GET `/claims` endpoint that is available to policyholders, underwriters, adjusters, and service vendors. All of these callers can use the endpoint to access resources whose type is `claim`, but none of the callers can access all of the claims. For example:

- A policyholder may be able to see only the claims associated with the policies they hold.
- An underwriter may be able to see only the claims for policies assigned to them.

- An adjuster may be able to see only the claims assigned to them.
- A service vendor may be able to see only the claims that have a service request assigned to them.

A *resource access strategy* is a set of logic that identifies which resources a caller can access. The base configuration includes resource access strategies for standalone services. But, these strategies do not involve resource access IDs or restrict the service in any way. Once a service is given access to a set of endpoints, the service can access any resource available to those endpoints. These applications are expected to be configured such that they access only the resources appropriate for the circumstance.

Strategy name	Persona using this strategy	The resource access ID is assumed to be...	Grants access to...
`pc.service`	Services	Not applicable	All resources

For more information on how resource access behaves, see “Resource access” on page 385.

Proxy user access for standalone services

Every Cloud API call occurs within the context of a session. A user listed in the PolicyCenter database must be attached to this session. PolicyCenter can use this "session user" in different ways.

- If the call creates or modifies an object, the session user is recorded as the `CreateUser` or `UpdateUser` of that object.
- If the call triggers an authority limit check, the session user's authority limits are checked.
- There is a small chance the call could trigger logic that must check to see if the caller has a specific domain-level system permission, such as the permission to own an activity. When this occurs, the session user's system permissions are checked.

A *proxy user* is an internal user that is assigned to a session for an API call made by an external user or service. Proxy users are assigned by the `RestAuthenticationSourceCreatorPlugin` plugin. This plugin specifies four proxy users. One of them, the proxy service user, is used for calls made by standalone services.

- If the call creates or modifies an object, the proxy service user is recorded as the `CreateUser` or `UpdateUser` of that object.
- If the call triggers an authority limit or domain-level system permission, the limits and permissions of the proxy service user are checked.

For more information on proxy users, see “Proxy user access” on page 391.

JWTs for standalone services

JSON Web Tokens (JWTs) contain token claims. (In standard JWT parlance, these are referred to simply as "claims". To avoid confusion with claims in the property and casualty insurance sense, this documentation sometimes refers to JWT claims as "token claims".) A *token claim* is a piece of information asserted about the bearer of the token, such as the bearer's name. For bearer token authentication, authentication information is stored in token claims.

When a service makes a standalone service call, only some of the information in the JWT is specific to Cloud API authentication. The following are the token claims in a JWT for a standalone service call that pertain to Cloud API authentication.

```
"sub": "<clientId>",
"cid": "<clientId>",
"scp": [
  "pc.service",
  "scp.pc.<serviceAPIRole>"
]
```

- `sub` is the subject of the token. This is set to the service's client ID.
- `cid` is the client ID of the service. This is also set to the service's client ID.
- The `scp` token claim has at least the following entries:

- The `pc.service` value, which specifies the caller is a service.
- A list of one or more API roles associated with the service. These roles are prefixed with "`scp.pc.`".

For example, the following JWT is for ACME's external billing application. (Information that is not relevant to Cloud API authorization has been omitted.)

```
{
    "sub": "acme_externalbillingapp",
    "cid": "acme_externalbillingapp",
    "scp": [
        "pc.service"
        "scp.pc.acme_externalbillingapp"
    ]
}
```

Note the following:

- Based on the `scp` token claim, this caller will be given endpoint access as defined in the role named "acme_externalbillingapp". Because of the `pc.service` entry, this caller uses a resource access strategy that provides access to all resources.

Logging

For each call, information about the caller is logged. The following table lists the fields that provide information about who the caller is, and where the logged value comes from.

Field	Value
`sub`	The value of the `sub` token claim from the JWT
`clientId`	The value of the `cid` token claim from the JWT
`user`	An empty string

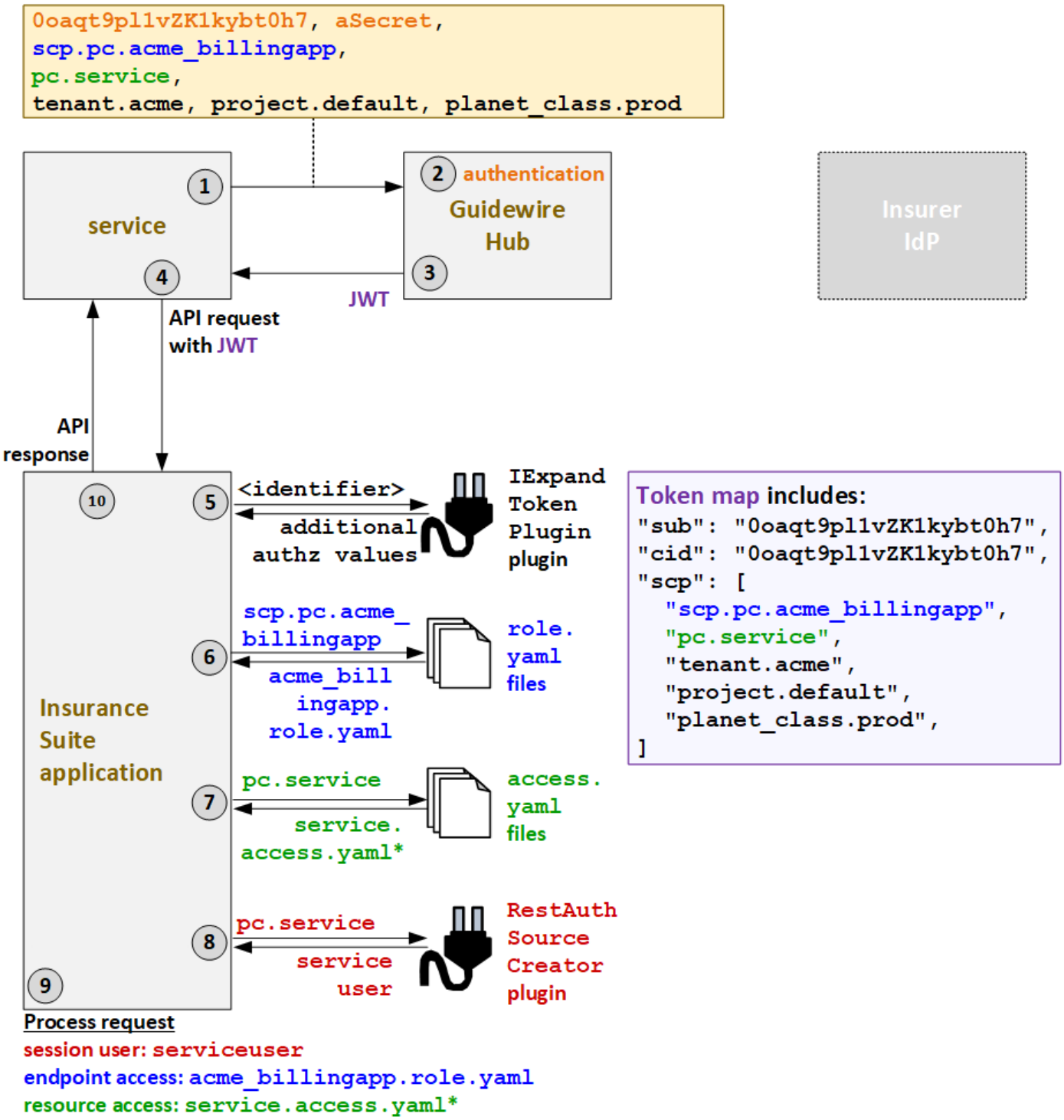
Example flow for standalone services

The following diagram identifies the flow of authentication and authorization information for standalone services. Colors are used in the following ways:

- Orange - credentials information
- Blue - endpoint access information
- Green - resource access information
- Red - proxy user and session user information

Some values are used to determine multiple types of access. These values initially appear as black (when they do not apply to a single type of access), and then later appear in one or more specific colors (to reflect the value is being used at that point in the process for a specific type of access).

In the following example, an API call is triggered by the Acme policy billing application, BillingApp.



1. When BillingApp triggers an API call, it must first request a JWT from Guidewire Hub. The request for the JWT includes the client ID (`0oaqt9pl1vZK1kybt0h7`), the secret (`aSecret`), the application's API role (`scp.pc.acme_billingapp`), the application's resource access strategy (`pc.service`), and additional deployment information (`tenant.acme`, `project.default`, `planet_class.prod`).
2. Guidewire Hub authenticates the services based on the client ID and secret. It also verifies that the API role and resource access strategy provided in the request match what was specified when the service was registered with Guidewire Hub.

3. Guidewire Hub generates a JWT and sends it to the service. This JWT includes the client ID (`cid`) and a `scp` token claim which names the API role (`scp.pc.acme_billingapp`), the resource access strategy (`pc.service`), and additional deployment information.
4. The service sends the API request to PolicyCenter along with the JWT.
5. PolicyCenter extracts the information in the JWT into a token map. Then, the IExpandTokenPlugin plugin calls any relevant authorization applications to retrieve any relevant additional auth values that must be added to or modified in the token map. (For standalone services, there is no need for resource access IDs. But, an insurer could choose to retrieve either the API role (such as `scp.pc.acme_billingapp`) or the resource access strategy name (`pc.service`) using the IExpandTokenPlugin plugin instead of sending them in the JWT.)
6. PolicyCenter determines the endpoint access. Based on the "`scp.pc.`" value listed in the token map (`scp.pc.acme_billingapp`), the `acme_billingapp.role.yaml` API role file is used to define the endpoint access.
7. Next, PolicyCenter determines the resource access strategy. Based on the resource access strategy value in the token map (`pc.service`), it grants resource access as defined in the `service access.yaml` files. (* PolicyCenter starts with `service_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with "`service`".)
8. To determine which proxy user to assign to the session, PolicyCenter calls the `RestAuthenticationSourceCreator` plugin. The token map specified a resource access strategy of `pc.service`. So, the plugin returns the proxy user for services: `serviceuser`.
9. PolicyCenter processes the request.
   a. The session user is the proxy service user: `serviceuser`.
   b. The endpoint access is defined by `acme_billingapp.role.yaml`.
   c. The resource access is defined by `service access.yaml`. In the base configuration, the `service access.yaml` files make all resources available. Therefore, logically speaking, there are no resource access restrictions.
10. PolicyCenter provides the response to the initial call.

Implementation checklist for standalone services

To configure Cloud API for authentication for standalone services, you may need to do the following tasks:

Task	More Information
Enable asymmetric encryption	“Enabling bearer token authentication” on page 369
Provide deployment information	“Enabling bearer token authentication” on page 369
Register the caller application with Guidewire Hub	“Enabling bearer token authentication” on page 369
Create or modify API roles	“Endpoint access” on page 373
Configure the proxy user	“Proxy user access” on page 391
Configure the IExpandTokenPlugin plugin to retrieve additional authorization values, if needed	“Configuring the IExpandTokenPlugin plugin” on page 395

Sending authenticated calls for standalone services

When a caller application wants to make a Cloud API call for a standalone service, the caller application must:

1. Request a JWT from Guidewire Hub
2. Include the JWT with the system API call

Requesting codes and JWTs from Guidewire Hub

For more information on how to request JWTs from Guidewire Hub, refer to *Authentication with Guidewire Identity Federation Hub* in the *Guidewire Cloud Platform* documentation set.

Including JWTs with API calls

Once a JWT has been received from Guidewire Hub, it must be sent to PolicyCenter in the request object's Authorization header. The header must use this format:

```
Authorization: Bearer <token>
```

Authentication failure error messages

For endpoints that return an individual element, when the resource exists but the user lacks authorization to access it, Cloud API throws the following user message. This is the same message that is returned when the resource does not exist.

```
"status": 404,
        "errorCode": "gw.api.rest.exceptions.NotFoundException",
        "userMessage": "No resource was found at path <path>"
```

For endpoints that return collections, Cloud API returns all resources that meet the criteria and for which the user has sufficient resource access. If a resource exists, but the user lacks sufficient authorization, Cloud API omits it from the results.

These approaches are considered to be more secure as they prevent malicious callers from being able to verify the existence of data that they are not authorized to access.

chapter 54

OAuth2 client credential flow: Services with user context

A *service* is an application that typically executes action without human intervention. Services typically have no user interface. Examples of services include:

- A billing application that alerts PolicyCenter about a delinquent policy that needs to be canceled.
- An application that uploads pictures of a covered location or vehicle, either when a policy is bound or after a loss has occurred.
- An external document management system.

This topic discusses how to execute authentication for services with user context.

Authentication options for services

There are several ways a service can execute authentication with Cloud API.

Standalone service

A service can authenticate **as a standalone service**. In this case, the service executes the call as itself. It does not execute the call as a specific person or on behalf of a specific person. The service does not execute the call using a service account stored in PolicyCenter.

PolicyCenter designates a single internal user as the "proxy service user" for all standalone service calls. This proxy service user is attached to the standalone service session. If the call creates or modifies an object, the proxy service user is recorded as the user of record.

The primary advantage to this approach is that you need to manage authentication and authorization information at the service level only. There is no need to create and manage user accounts, user permissions, or additional mappings.

The primary disadvantage is that all standalone service calls share a single proxy service user. When a standalone service call creates or modifies an object, it may not be possible to identify which service made the call.

Service with user context

A service can authenticate **with user context**. In this case, the service presents information about itself. The call also includes a `GW-User-Context` header that provides information about a specific user. The user does not necessarily exist in the PolicyCenter database. The call is able to do only the things that both the service by itself could do and the user by itself could do.

The specified user can be an internal user (a user who is listed in the PolicyCenter database). When this is the case, this internal user is attached to the session. If the call creates or modifies an object, this internal user is recorded as the user of record.

The specified user can be an external user (a user who is not listed in the PolicyCenter database). PolicyCenter designates a single internal user as the "proxy external user" for all service with user context calls that reference external users. When the specified user is an external user, the external proxy user is attached to the session. If the call creates or modifies an object, the external proxy user is recorded as the user of record.

The primary advantage to this approach is that a single service can send calls on behalf of different users. At the service level, you can specify service-level access. But, you can also further control access for each associated user.

There are two primary disadvantages. First, you must maintain access information at two levels: the service level and at the user level. Second, a service can specify any user in its header. There is no way to restrict a given set of users for use by a given service.

Service with service account mapping

A service can authenticate **with service account mapping**. In this case, the service is automatically mapped to a "service account". The mapping information is specified elsewhere in the environment. The service account is a user account in the PolicyCenter database that is intended to be used only by the service and not by any person.

The primary advantage to this approach is that permissions and auditing for the call is tied to a service account that is listed in the PolicyCenter database. You can create a different service account for each service and have fine control over the permissions available to each service.

The primary disadvantage is that you must create and maintain service accounts in PolicyCenter for calls made by external applications. Also, you must maintain the mapping information that maps each service to its service account.

Comparing the different approaches

The following table compares each approach.

	Standalone service	Service with user context	Service with service account mapping
Does the call provide information about the service?	Yes, in the JWT.	Yes, in the JWT.	Yes, in the JWT.
Does there need to be a user account in the PolicyCenter database for the call?	No	If the associated user is an internal user, yes. If the associated user is an external user, no.	Yes. (This user account is the "service account".)
Does the call include information about a user or user account?	No	Yes, in the `GW-User-Context` header.	No. The call provides a client ID for the service, but the mapping of client ID to service account is stored elsewhere.
Which endpoints can the call access?	The endpoints available to the service's API roles	The endpoints available to both the service's API roles and the user's API roles.	The endpoints available to the service account.
Which resources can the call access?	All resources (in the base configuration).	The resources available to both the service and the user.	The resources available to the service account.
What is the session user set to?	The proxy service user.	If the associated user is an internal user, the internal user. If the associated user is an external user, the proxy external user.	The service account.

This topic focuses on authentication for services with user context.

- For more information on authentication for standalone services, see “OAuth2 client credential flow: Standalone services” on page 325.

- For more information on authentication for services with service account mapping, see “OAuth2 client credential flow: Services with service account mapping” on page 349.

Overview of authentication for services with user context

Authentication includes credentials and authorization. Authentication information for services is specified in JWTs, and information from these JWTs is recorded in the logs.

Note: When using the service with user context flow, you cannot specify the application's unrestricted user as the user of context. In the base configuration, the unrestricted user is `su`.

Credentials

When a service makes an API call, the service sends a client ID and secret to Guidewire Hub. Guidewire Hub authenticates the service by confirming that the client secret is correct. This is true for standalone services, services with user context, and services with service account mapping.

When a service authenticates with user context, it provides information about a user. However, there is no authentication at the user level. Authentication occurs only at the service level.

For more information on how client IDs and secrets are registered with Guidewire Hub, see “Registering the caller application with Guidewire Hub” on page 372.

Authorization

Endpoint access for services with user context

Endpoint access defines the aspects of an endpoint's behaviors that are available to a caller. This includes:

- What endpoints and resource types are available to the caller?
- What operations can a caller call on the available endpoint?
- What fields can the caller specify in a request payload or get in a response payload?

Endpoint access is controlled by API roles. An *API role* is a list of endpoints, operations, and fields that are available to a set of callers through API calls. API roles act as allowlists. By default, a caller has no endpoint access. When the caller is associated with one or more API roles, they gain access to the endpoints, operations, and fields allowlisted in each of those API roles.

For a service-with-user-context call, Cloud API checks two sets of API roles:

- The API roles assigned to the service
- The API roles assigned to the user

The endpoint access granted to the call is the intersection of the endpoint access granted by these two sets. In other words, in order to access an endpoint, operation, or field, the access must be granted to at least one API role assigned to the service and at least one API role assigned to the user account.

For example, suppose that the ACME External Document Manager service has the following API role with the following endpoint access:

- `acme_externaldocumentmanager`
  - GET `/documents`
  - POST `/documents`

And, suppose that Ray Newton has the following API role with the following endpoint access:

- `Insured`
  - GET `/documents`

    - GET `/coverages`

Suppose ACME External Document Manager service makes a call as a service with user account using the Ray Newton user account. The call would have access to GET `/documents`, as this endpoint has been granted to both the service and the user account. The call would not have access to either POST `/documents` or GET `/coverages`, as neither of these endpoints have been granted to both the service and the user account.

For more information on how API roles are configured, see “Endpoint access” on page 373.

Resource access for services with user context

Resource access defines, for a given type of resource, which instances of that resource the caller can access. For example, suppose there is a GET `/claims` endpoint that is available to policyholders, underwriters, adjusters, and service vendors. All of these callers can use the endpoint to access resources whose type is claim, but none of the callers can access all of the claims. For example:

- A policyholder may be able to see only the claims associated with the policies they hold.
- An underwriter may be able to see only the claims for policies assigned to them.
- An adjuster may be able to see only the claims assigned to them.
- A service vendor may be able to see only the claims that have a service request assigned to them.

A *resource access strategy* is a set of logic that identifies which resources a caller can access. The base configuration includes resource access strategies for users and services.

- Resource access strategies for users typically restrict the caller to objects owned by the user. For example, an account holder can see objects owned (directly or indirectly) by the account.
- Resource access strategies for services typically do not restrict the caller in any way.

Similar the endpoint access, the resource access for a service-with-user-context call is the intersection of the resource access for the service and the resource access for the user. In the base configuration, services have access to all resources. Therefore, the resource access for a service-with-user-context call is logically equivalent to the resource access for the user.

For example, suppose that the ACME External Document Manager service has access to the following documents:

- (all documents)

And, suppose that Ray Newton has access to the following documents:

- Documents associated with policy 55-123456
    - Document xc:127
    - Document xc:356
- Documents associated directly with account C000324667
    - Document xc:888

If ACME External Document Manager service makes a service-with-user-account call for Ray Newton, the call would have access to document xc:127, xc:356, and xc:888, as these resources can be accessed by both the service and the user. The call would not have access to any other documents, as there are no other documents that both the service and the user have access to.

For more information on how resource access behaves, see “Resource access” on page 385.

Proxy user access for services with user context

Every Cloud API call occurs within the context of a session. A user listed in the PolicyCenter database must be attached to this session. PolicyCenter can use this "session user" in different ways.

- If the call creates or modifies an object, the session user is recorded as the `CreateUser` or `UpdateUser` of that object.

- If the call triggers an authority limit check, the session user's authority limits are checked.
- There is a small chance the call could trigger logic that must check to see if the caller has a specific domain-level system permission, such as the permission to own an activity. When this occurs, the session user's system permissions are checked.

When a service makes a call with a user context, the associated user could be an internal user. An *internal user* is a person who is listed as a user in the PolicyCenter operational database. In this case, that internal user is used as the session user.

When a service makes a call with a user context, the associated user could be an external user. An *external user* is a person who is known to the insurer but who is not listed as a user in the PolicyCenter operational database. For example, this could be a policyholder, vendor, or account holder. When the user context specifies an external user, a proxy user must be assigned to the session.

A *proxy user* is an internal user that is assigned to a session for an API call made by an external user or service. Proxy users are assigned by the `RestAuthenticationSourceCreatorPlugin` plugin. This plugin specifies four proxy users. One of them, the proxy external user, is used for calls made either by external users or by services with user context where the user context specifies an external user.

For more information on proxy users, see “Proxy user access” on page 391.

JWTs for services with user context

JSON Web Tokens (JWTs) contain token claims. (In standard JWT parlance, these are referred to simply as "claims". To avoid confusion with claims in the property and casualty insurance sense, this documentation always refers to JWT claims as "token claims".) A *token claim* is a piece of information asserted about the bearer of the token, such as the bearer's name. For bearer token authentication, authentication information is stored in token claims.

When a service makes a call with user context, only some of the information in the JWT is specific to Cloud API authentication. The following are the token claims in a JWT for a service-for-user-context call that pertain to Cloud API authentication.

```
"sub": "<clientId>",
"cid": "<clientId>",
"scp": [
  "pc.service",
  "scp.pc.<serviceAPIRole>",
  "pc.allowusercontext"
]
```

- `sub` is the subject of the token. This is set to the service's client ID.
- `cid` is the client ID of the service. This is also set to the service's client ID.
- The `scp` token claim has at least the following entries:
  - The `pc.service` value, which specifies the caller is a service.
  - A list of one or more API roles associated with the service. These roles are prefixed with "`scp.pc.`".
  - The `pc.allowusercontext` value, which allows the call to specify additional user context in the header. (If the header contains a `GW-User-Context` header, then this is treated as a service-with-user-context call and not a standalone service call).

The allowusercontext header

The `pc.allowusercontext` value indicates that the call is also presenting a user context. Information about the user is specified in a `GW-User-Context` header.

- For internal users, the header:
  - Must specify the user name and the resource access strategy and resource access IDs.
  - Must be base64-encoded
- For external users, the header:

- Must specify the user name, the user roles, and the resource access strategy and resource access IDs
- Must be base64-encoded

Note: When using the service with user context flow, you cannot specify the application's unrestricted user as the user of context. In the base configuration, the unrestricted user is `su`.

Syntax for the JSON object

The header must be a JSON payload that is formatted as described in the following paragraphs.

For an internal user, the syntax of the `GW-User-Context` header is:

```
{
  "sub": "<userName>",
  "pc_username" : "<userName>"
}
```

Note the following:

- The user name is specified in the `sub` token claim.
- The resource access strategy is specified by the presence of the `pc_username` token claim.
- The resource access ID is the user's name, which is specified within the `pc_username` token claim.

For an external user who is an account holder, the syntax of the `GW-User-Context` header is:

```
{
  "sub": "<userName>",
  "groups": [
    "<userAPIroleList>"
  ],
  "pc_accountNumbers" : [
    "<accountNumbers>"
  ]
}
```

Note the following:

- The user name is specified in the `sub` token claim. (This is used for logging, but otherwise it has no functional impact.)
- The user's API roles are listed in the `groups` token claim. Every role name must be prefixed with "`gwa.<planet_class>.pc.`".
- The resource access strategy is specified by the presence of the `pc_accountNumbers` token claim.
- The resource access IDs are a list of account numbers, which are specified within the `pc_accountNumbers` token claim.

Making the JSON object base64-encoded

The JSON object cannot be added to the header as JSON. It must be base64-encoded.

For example, suppose you had the following JSON object for internal user aapplegate:

```
{
  "sub": "aapplegate@acme.com",
  "pc_username" : "aapplegate"
}
```

The header must contain the base64-encoded version of this object, as shown below.

```
ewogICJzdWIiOiAiYWFwcGxlZ2F0ZUBhY21lLmNvbSIsCiAgInBjX3VzZXJuYW1lIiA6ICJhYXBw
bGVnYXRlQGFjbWUuY29tIgp9
```

Adding the base64-encoded object to the header

Finally, this object must be attached to a request header named "GW-User-Context". For example, if you were sending the previous request through Postman, the values would be as follows:

- Key: GW-User-Context
- Value: ewogICJzdWIiOiAiYWFwcGxlZ2F0ZUBhY21lLmNvbSIsCiAgInBjX3VzZXJuYW1lIiA6ICJhYXBwbGVnYXRlQGFjbWUuY29tIgp9

Note: If a call includes a JWT with the `pc.allowusercontext` token claim, but the request object's header does not contain a user context header, Cloud API treats the call as if it were coming from a standalone service. In other words, the call will be restricted to the access provided to the service. No user-based restrictions are applied because there was no user context header specifying a user.

Logging

For each call, information about the caller is logged. The following table lists the fields that provide information about who the caller is, and where the logged value comes from.

Field	Value
`sub`	The value of the `sub` token claim from the JWT
`clientId`	The value of the `cid` token claim from the JWT
`user`	If the user in the user context is an internal user, the user name of the internal user. If the user in the user context is an external user, the `sub` value from the user context.

Example flow for services with user context

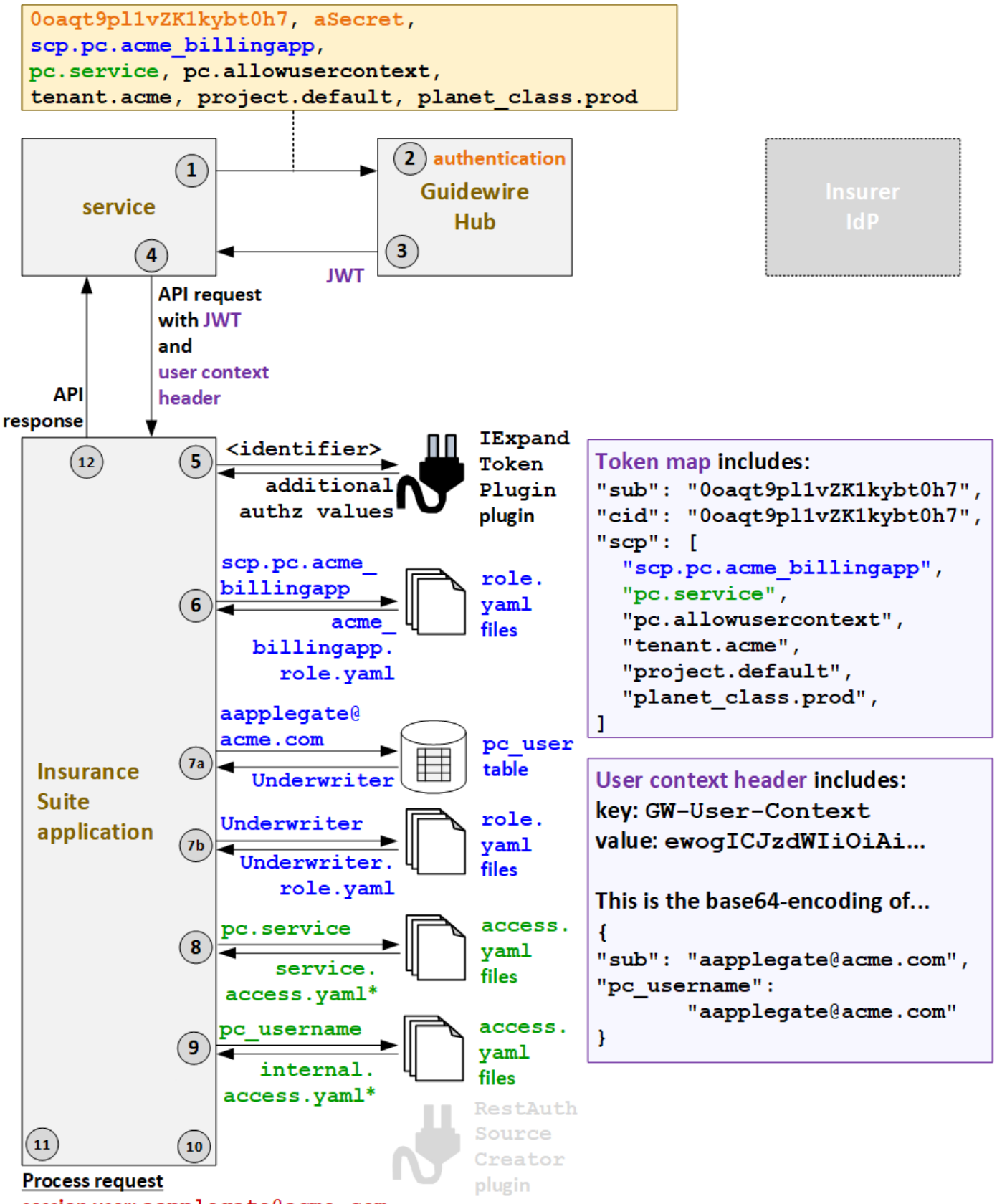
The following diagram identifies the flow of authentication and authorization information for services with user context. Colors are used in the following ways:

- Orange - credentials information
- Blue - endpoint access information
- Green - resource access information
- Red - proxy user and session user information

Some values are used to determine multiple types of access. These values initially appear as black (when they do not apply to a single type of access), and then later appear in one or more specific colors (to reflect the value is being used at that point in the process for a specific type of access).

Services with user context: internal users

In the following example, an API call is triggered by the Acme policy billing application, BillingApp, on behalf of Alice Applegate, who is an internal user.


0oaqt9pl1vZK1kybt0h7, aSecret,
scp.pc.acme_billingapp,
pc.service, pc.allowusercontext,
tenant.acme, project.default, planet_class.prod
service
1
4
2 authentication
Guidewire Hub
3
JWT
Insurer IdP
API request with JWT and user context header
API response
Insurance Suite application
12
5
<identifier>
additional authz values
IExpand Token Plugin plugin
6
scp.pc.acme_billingapp
acme_billingapp.role.yaml
role.yaml files
7a
aapplegate@acme.com
Underwriter
pc_user table
7b
Underwriter
Underwriter.role.yaml
role.yaml files
8
pc.service
service.access.yaml*
access.yaml files
9
pc_username
internal.access.yaml*
access.yaml files
RestAuth Source Creator plugin
11
10
Process request
session user: aapplegate@acme.com
endpoint access: intersection of acme_billingapp.role.yaml and Underwriter.role.yaml
resource access: intersection of service.access.yaml* and internal.access.yaml*, using aapplegate@acme.com
Token map includes:
"sub": "0oaqt9pl1vZK1kybt0h7",
"cid": "0oaqt9pl1vZK1kybt0h7",
"scp": [
"scp.pc.acme_billingapp",
"pc.service",
"pc.allowusercontext",
"tenant.acme",
"project.default",
"planet_class.prod",
]
User context header includes:
key: GW-User-Context
value: ewogICJzdWIiOiAi...
This is the base64-encoding of...
{
"sub": "aapplegate@acme.com",
"pc_username":
"aapplegate@acme.com"
}

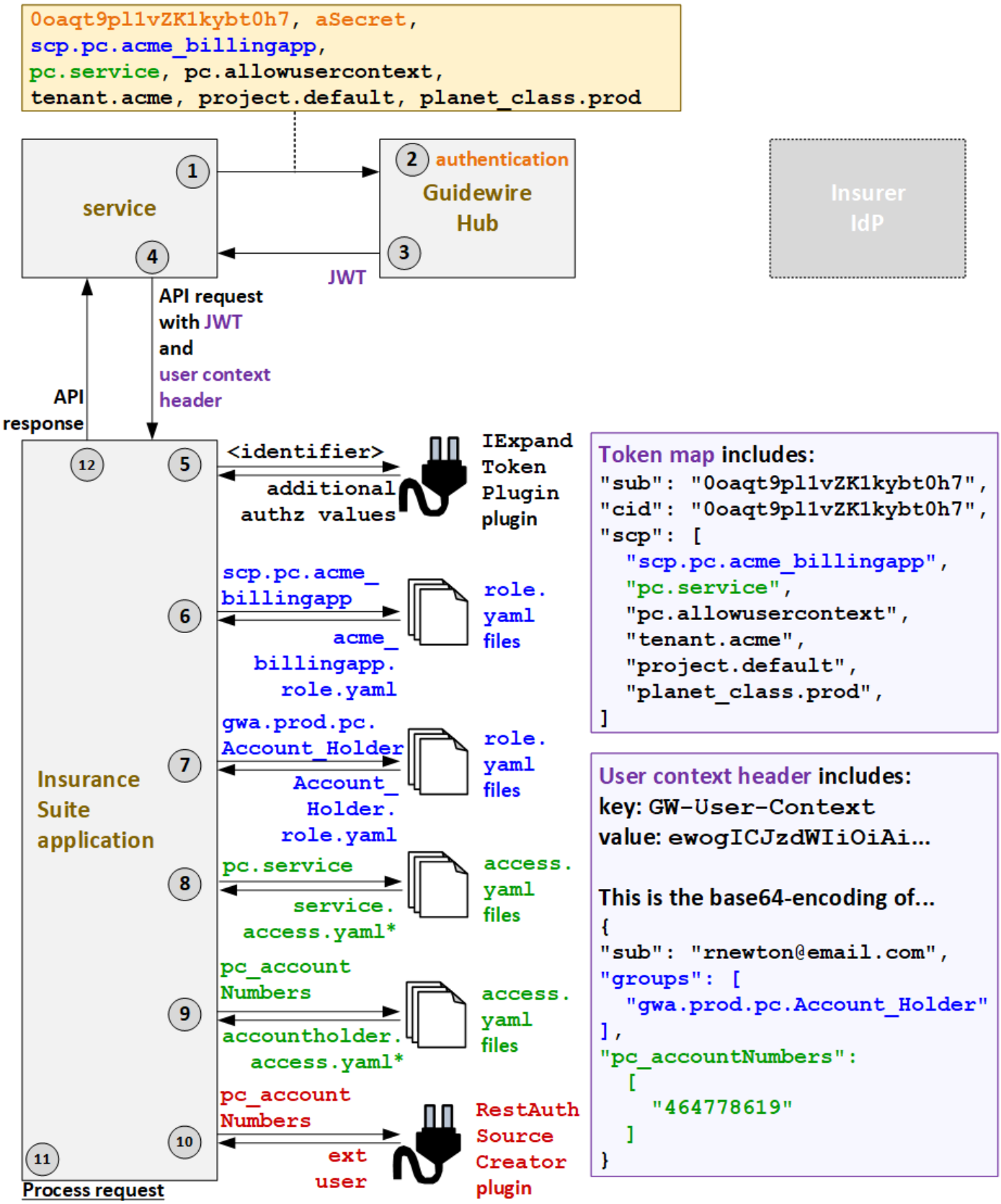
1. When BillingApp triggers an API call, it must first request a JWT from Guidewire Hub. The request for the JWT includes the client ID (`0oaqt9pl1vZK1kybt0h7`), the secret (`aSecret`), the application's API role (`scp.pc.acme_billingapp`), the application's resource access strategy (`pc.service`), the fact that the call will be made for a user (`pc.allowusercontext`) and additional deployment information (`tenant.acme`, `project.default`, `planet_class.prod`).
2. Guidewire Hub authenticates the services based on the client ID and secret. It also verifies that the API role and resource access strategy provided in the request match what was specified when the service was registered with Guidewire Hub.
3. Guidewire Hub generates a JWT and sends it to the service. This JWT includes the client ID (`cid`) and a `scp` token claim which names the API role (`scp.pc.acme_billingapp`), the resource access strategy (`pc.service`), the `pc.allowusercontext` value (which indicates that user information is specified in an additional user context header), and additional deployment information.
4. The service sends the API request to PolicyCenter along with the JWT and a user context header that identifies the user (`aapplegate@acme.com`), the user's resource access strategy (`pc_username`), and resource access ID (`aapplegate@acme.com`).
5. PolicyCenter extracts the information in the JWT into a token map. Then, the IExpandTokenPlugin plugin calls any relevant authorization applications to retrieve any relevant additional auth values that must be added to or modified in the token map. (The IExpandTokenPlugin plugin can affect only the information that normally comes in the JWT. It cannot affect information in a user context header. Therefore, for services with user context, an insurer could choose to retrieve either the service's API role (such as `scp.pc.acme_billingapp`) or the service's resource access strategy name (`pc.service`) using the IExpandTokenPlugin plugin. But the API roles and resource access IDs for the user must be in the user context header when the call is sent to PolicyCenter.)
6. PolicyCenter must determine the endpoint access at both the service level and the user level. It starts at the service level. Based on the API role value in the JWT (`scp.pc.acme_billingapp`), the `acme_billingapp.role.yaml` API role file is used to define the service-level access.
7. Next, PolicyCenter determines the user-level endpoint access.
   a. Using the user name in the user context header (`aapplegate@acme.com`), PolicyCenter queries for the user roles that this user has. One role is returned: `Underwriter`.
   b. Based on the returned role, the `Underwriter.role.yaml` API role file is used to define the user-level access.
8. PolicyCenter must also determine the resource access at the service level and the user level. It starts with the service-level resource access strategy. Based on the resource access strategy value in the JWT (`pc.service`), it grants service-level resource access as defined in the `service access.yaml` files. (* PolicyCenter starts with `service_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with "`service`".)
9. PolicyCenter determines the user-level resource access strategy. Based on the resource access strategy value in the user context header (`pc_username`), it grants user-level resource access as defined in the `internal access.yaml` files. (* PolicyCenter starts with `internal_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with "`internal`".)
10. Proxy user access is not relevant for services with user context when the user is an internal user.
11. PolicyCenter processes the request.
    a. The session user is the internal user: `aapplegate@acme.com`.
    b. The endpoint access is the intersection of the endpoints and operations defined granted at the service level (`acme_billingapp.role.yaml`) and at the user level (`Underwriter.role.yaml`). Endpoints, operations, and fields must be listed at both levels to be available to the call.
    c. The resource access is the intersection of the resources accessible to the service (as defined in the `service access.yaml`) and the resources available to the user (as defined in the `internal access.yaml` using the resource access ID of `aapplegate@acme.com`). In the base configuration, the `service access.yaml` files make all resources available. Therefore, logically speaking, the service-level resource access does not

specify any restrictions. The call can access any resource provided it is available through the user-level resource access.

12. PolicyCenter provides the response to the initial call.

Services with user context: external users

In the following example, an API call is triggered by the Acme policy billing application, BillingApp, on behalf of Ray Newton, who is an external user.



Token map includes:

```
"sub": "0oaqt9pl1vZK1kybt0h7",
"cid": "0oaqt9pl1vZK1kybt0h7",
"scp": [
  "scp.pc.acme_billingapp",
  "pc.service",
  "pc.allowusercontext",
  "tenant.acme",
  "project.default",
  "planet_class.prod",
]
```

User context header includes:

key: GW-User-Context

value: ewogICJzdWIiOiAi...

This is the base64-encoding of...

```
{
"sub": "rnewton@email.com",
"groups": [
  "gwa.prod.pc.Account_Holder"
],
"pc_accountNumbers":
  [
    "464778619"
  ]
}
```

1. When BillingApp triggers an API call, it must first request a JWT from Guidewire Hub. The request for the JWT includes the client ID (`0oaqt9pl1vZK1kybt0h7`), the secret (`aSecret`), the application's API role (`scp.pc.acme_billingapp`), the application's resource access strategy (`pc.service`), the fact that the call will be made for a user (`pc.allowusercontext`) and additional deployment information (`tenant.acme`, `project.default`, `planet_class.prod`).
2. Guidewire Hub authenticate the services based on the client ID and secret. It also verifies that the API role and resource access strategy provided in the request match what was specified when the service was registered with Guidewire Hub.
3. Guidewire Hub generates a JWT and sends it to the service. This JWT includes the client ID (`cid`) and a `scp` token claim which names the API role (`scp.pc.acme_billingapp`), the resource access strategy (`pc.service`), the `pc.allowusercontext` value (which indicates that user information is specified in an additional user context header), and additional deployment information.
4. The service sends the API request to PolicyCenter along with the JWT and a user context header that identifies the user (`rnewton@email.com`), the user's resource access strategy (`pc_accountNumbers`), and resource access ID (`464778619`).
5. PolicyCenter extracts the information in the JWT into a token map. Then, the IExpandTokenPlugin plugin calls any relevant authorization applications to retrieve any relevant additional auth values that must be added to or modified in the token map. (The IExpandTokenPlugin plugin can affect only the information that normally comes in the JWT. It cannot affect information in a user context header. Therefore, for services with user context, an insurer could choose to retrieve either the service's API role (such as `scp.pc.acme_billingapp`) or the service's resource access strategy name (`pc.service`) using the IExpandTokenPlugin plugin. But the API roles and resource access IDs for the user must be in the user context header when the call is sent to PolicyCenter.)
6. PolicyCenter must determine the endpoint access at both the service level and the user level. It starts at the service level. Based on the API role value in the JWT (`scp.pc.acme_billingapp`), the `acme_billingapp.role.yaml` API role file is used to define the service-level access.
7. Next, PolicyCenter determines the user-level endpoint access. Based on the contents of the `groups` token claim in the user context header (`gwa.prod.pc.Account_Holder`), the `Account_Holder.role.yaml` API role file is used to define the user-level access.
8. PolicyCenter must also determine the resource access at the service level and the user level. It starts with the service-level resource access strategy. Based on the resource access strategy value in the JWT (`pc.service`), PolicyCenter grants service-level resource access as defined in the `service` `access.yaml` files. (* PolicyCenter starts with service_ext-1.0.access.yaml, but this file references additional `access.yaml` files whose name starts with "`service`".)
9. Next, PolicyCenter determines the user-level resource access strategy. Based on the resource access strategy value in the user context header (`pc_accountNumbers`), it grants user-level resource access as defined in the `accountholder` `access.yaml` files. (* PolicyCenter starts with `accountholder_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with "`accountholder`".)
10. To determine which proxy user to assign to the session, PolicyCenter calls the `RestAuthenticationSourceCreator` plugin. The user context header specified a resource access strategy of `pc_accountNumbers`. So, the plugin returns the proxy user for external users: `extuser`.
11. PolicyCenter processes the request.
    a. The session user is the proxy external user: `extuser`.
    b. The endpoint access is the intersection of the endpoints and operations defined granted at the service level (`acme_billingapp.role.yaml`) and at the user level (`Account_Holder.role.yaml`). Endpoints, operations, and fields must be listed at both levels to be available to the call.
    c. The resource access is the intersection of the resources accessible to the service (as defined in the `service` `access.yaml`) and the resources available to the user (as defined in the `accountNumbers` `access.yaml` using the resource access ID of `464778619`). In the base configuration, the `service` `access.yaml` files make all resources available. Therefore, logically speaking, the service-level resource access does not

specify any restrictions. The call can access any resource provided it is available through the user-level resource access.

12. PolicyCenter provides the response to the initial call.

Implementation checklist for services with user context

To configure Cloud API for authentication for services with user context, you may need to do the following tasks:

Task	More Information
Enable asymmetric encryption	“Enabling bearer token authentication” on page 369
Provide deployment information	“Enabling bearer token authentication” on page 369
Register the caller application with Guidewire Hub	“Enabling bearer token authentication” on page 369
Create or modify API roles	“Endpoint access” on page 373
Review the resource access strategies provided in the base configuration	“Resource access” on page 385
Configure the proxy user	“Proxy user access” on page 391
Configure the IExpandTokenPlugin plugin to retrieve additional authorization values, if needed	“Configuring the IExpandTokenPlugin plugin” on page 395

Sending authenticated calls for services with user context

When a caller application wants to make a Cloud API call for a service with user context, the caller application must:

1. Request a JWT from Guidewire Hub
2. Include the JWT (and the `GW-User-Context` header) with the Cloud API call

Requesting codes and JWTs from Guidewire Hub

For more information on how to request JWTs from Guidewire Hub, refer to *Authentication with Guidewire Identity Federation Hub* in the *Guidewire Cloud Platform* documentation set.

Including JWTs with API calls

Once a JWT has been received from Guidewire Hub, it must be sent to PolicyCenter in the request object's Authorization header. The header must use this format:

```
Authorization: Bearer <token>
```

Note: If a call includes a JWT with the `pc.allowusercontext` token claim, but the request object's header does not contain a user context header, Cloud API treats the call as if it were coming from a standalone service. In other words, the call will be restricted to the access provided to the service. No user-based restrictions are applied because there was no user context header specifying a user.

Including the GW-UserContext header with API calls

The `pc.allowusercontext` value indicates that the call is also presenting a user context. Information about the user is specified in a `GW-User-Context` header.

- For internal users, the header:
  - Must specify the user name and the resource access strategy and resource access IDs.
  - Must be base64-encoded
- For external users, the header:

- Must specify the user name, the user roles, and the resource access strategy and resource access IDs
- Must be base64-encoded

Syntax for the JSON object

The header must be a JSON payload that is formatted as described in the following paragraphs.

For an internal user, the syntax of the `GW-User-Context` header is:

```
{
        "sub": "<userName>",
        "pc_username" : "<userName>"
}
```

Note the following:

- The user name is specified in the `sub` token claim.
- The resource access strategy is specified by the presence of the `pc_username` token claim.
- The resource access ID is the user's name, which is specified within the `pc_username` token claim.

For an external user who is an account holder, the syntax of the `GW-User-Context` header is:

```
{
  "sub": "<userName>",
  "groups": [
    "<userAPIroleList>"
  ],
  "pc_accountNumbers" : [
    "<accountNumbers>"
  ]
}
```

Note the following:

- The user name is specified in the `sub` token claim. (This is used for logging, but otherwise it has no functional impact.)
- The user's API roles are listed in the `groups` token claim. Every role name must be prefixed with "`gwa.<planet_class>.pc.`".
- The resource access strategy is specified by the presence of the `pc_accountNumbers` token claim.
- The resource access IDs are a list of account numbers, which are specified within the `pc_accountNumbers` token claim.

Making the JSON object base64-encoded

The JSON object cannot be added to the header as JSON. It must be base64-encoded.

For example, suppose you had the following JSON object for internal user aapplegate:

```
{
        "sub": "aapplegate@acme.com",
        "pc_username" : "aapplegate@acme.com"
        }
```

The header must contain the base64-encoded version of this object, as shown below.

```
ewogICJzdWIiOiAiYWFwcGxlZ2F0ZUBhY21lLmNvbSIsCiAgInBjX3VzZXJuYW1lIiA6ICJhYXBw
        bGVnYXRlQGFjbWUuY29tIgp9
```

Adding the base64-encoded object to the header

Finally, this object must be attached to a request header named "GW-User-Context". For example, if you were sending the previous request through Postman, the values would be as follows:

- Key: GW-User-Context

- Value: ewogICJzdWIiOiAiYWFwcGxlZ2F0ZUBhY21lLmNvbSIsCiAgInBjX3VzZXJuYW1lIiA6ICJhYXBwbGVnYXRlQGFjbWUuY29tIgp9

Note: If a call includes a JWT with the `pc.allowusercontext` token claim, but the request object's header does not contain a user context header, Cloud API treats the call as if it were coming from a standalone service. In other words, the call will be restricted to the access provided to the service. No user-based restrictions are applied because there was no user context header specifying a user.

Authentication failure error messages

For endpoints that return an individual element, when the resource exists but the user lacks authorization to access it, Cloud API throws the following user message. This is the same message that is returned when the resource does not exist.

```
"status": 404,
        "errorCode": "gw.api.rest.exceptions.NotFoundException",
        "userMessage": "No resource was found at path <path>"
```

For endpoints that return collections, Cloud API returns all resources that meet the criteria and for which the user has sufficient resource access. If a resource exists, but the user lacks sufficient authorization, Cloud API omits it from the results.

These approaches are considered to be more secure as they prevent malicious callers from being able to verify the existence of data that they are not authorized to access.

chapter 55

OAuth2 client credential flow: Services with service account mapping

A *service* is an application that typically executes action without human intervention. Services typically have no user interface. Examples of services include:

- A billing application that alerts PolicyCenter about a delinquent policy that needs to be canceled.
- An application that uploads pictures of a covered location or vehicle, either when a policy is bound or after a loss has occurred.
- An external document management system.

This topic discusses how to execute authentication for services with service account mapping.

Authentication options for services

There are several ways a service can execute authentication with Cloud API.

Standalone service

A service can authenticate **as a standalone service**. In this case, the service executes the call as itself. It does not execute the call as a specific person or on behalf of a specific person. The service does not execute the call using a service account stored in PolicyCenter.

PolicyCenter designates a single internal user as the "proxy service user" for all standalone service calls. This proxy service user is attached to the standalone service session. If the call creates or modifies an object, the proxy service user is recorded as the user of record.

The primary advantage to this approach is that you need to manage authentication and authorization information at the service level only. There is no need to create and manage user accounts, user permissions, or additional mappings.

The primary disadvantage is that all standalone service calls share a single proxy service user. When a standalone service call creates or modifies an object, it may not be possible to identify which service made the call.

Service with user context

A service can authenticate **with user context**. In this case, the service presents information about itself. The call also includes a `GW-User-Context` header that provides information about a specific user. The user does not necessarily exist in the PolicyCenter database. The call is able to do only the things that both the service by itself could do and the user by itself could do.

The specified user can be an internal user (a user who is listed in the PolicyCenter database). When this is the case, this internal user is attached to the session. If the call creates or modifies an object, this internal user is recorded as the user of record.

The specified user can be an external user (a user who is not listed in the PolicyCenter database). PolicyCenter designates a single internal user as the "proxy external user" for all service with user context calls that reference external users. When the specified user is an external user, the external proxy user is attached to the session. If the call creates or modifies an object, the external proxy user is recorded as the user of record.

The primary advantage to this approach is that a single service can send calls on behalf of different users. At the service level, you can specify service-level access. But, you can also further control access for each associated user.

There are two primary disadvantages. First, you must maintain access information at two levels: the service level and at the user level. Second, a service can specify any user in its header. There is no way to restrict a given set of users for use by a given service.

Service with service account mapping

A service can authenticate **with service account mapping**. In this case, the service is automatically mapped to a "service account". The mapping information is specified elsewhere in the environment. The service account is a user account in the PolicyCenter database that is intended to be used only by the service and not by any person.

The primary advantage to this approach is that permissions and auditing for the call is tied to a service account that is listed in the PolicyCenter database. You can create a different service account for each service and have fine control over the permissions available to each service.

The primary disadvantage is that you must create and maintain service accounts in PolicyCenter for calls made by external applications. Also, you must maintain the mapping information that maps each service to its service account.

Comparing the different approaches

The following table compares each approach.

	Standalone service	Service with user context	Service with service account mapping
Does the call provide information about the service?	Yes, in the JWT.	Yes, in the JWT.	Yes, in the JWT.
Does there need to be a user account in the PolicyCenter database for the call?	No	If the associated user is an internal user, yes. If the associated user is an external user, no.	Yes. (This user account is the "service account".)
Does the call include information about a user or user account?	No	Yes, in the `GW-User-Context` header.	No. The call provides a client ID for the service, but the mapping of client ID to service account is stored elsewhere.
Which endpoints can the call access?	The endpoints available to the service's API roles	The endpoints available to both the service's API roles and the user's API roles.	The endpoints available to the service account.
Which resources can the call access?	All resources (in the base configuration).	The resources available to both the service and the user.	The resources available to the service account.
What is the session user set to?	The proxy service user.	If the associated user is an internal user, the internal user. If the associated user is an external user, the proxy external user.	The service account.

This topic focuses on authentication for services with service account mapping.

- For more information on authentication for standalone services, see “OAuth2 client credential flow: Standalone services” on page 325.

- For more information on authentication for services with user context, see “OAuth2 client credential flow: Services with user context” on page 333.

Overview of authentication for services with service account mapping

Authentication includes credentials and authorization. Authentication information for services is specified in JWTs, and information from these JWTs is recorded in the logs.

Credentials

When a service makes an API call, the service sends a client ID and secret to Guidewire Hub. Guidewire Hub authenticates the service by confirming that the client secret is correct. This is true for standalone services, services with user context, and services with service account mapping.

When a service authenticates with service account mapping, the service is mapped to a service account in the PolicyCenter database. However, there is no authentication at the service account level. Authentication occurs only at the service level.

For more information on how client IDs and secrets are registered with Guidewire Hub, see “Registering the caller application with Guidewire Hub” on page 372.

Authorization

Endpoint access for services with service account mapping

Endpoint access defines the aspects of an endpoint's behaviors that are available to a caller. This includes:

- What endpoints and resource types are available to the caller?
- What operations can a caller call on the available endpoint?
- What fields can the caller specify in a request payload or get in a response payload?

Endpoint access is controlled by API roles. An *API role* is a list of endpoints, operations, and fields that are available to a set of callers through API calls. API roles act as allowlists. By default, a caller has no endpoint access. When the caller is associated with one or more API roles, they gain access to the endpoints, operations, and fields allowlisted in each of those API roles.

For a service-with-service-account-mapping call, Cloud API maps the service to a service account in the PolicyCenter database. Then, PolicyCenter queries the operational database for this service account's user roles. The service is given endpoint access to all API roles whose names corresponds to the names of the service account's user roles.

For example, suppose that the ACME QuoteAndBind service is mapped to a service account named "acmeQuoteAndBind". The acmeQuoteAndBind account has two user roles: "ACME Underwriter" and "ACME Reinsurance Manager". The ACME QuoteAndBind service triggers a Cloud API call. PolicyCenter maps the service to the acmeQuoteAndBind account and queries the database for the service account's user roles. Two user roles are returned: "ACME Underwriter" and "ACME Reinsurance Manager". PolicyCenter then gives the service the endpoint access defined in the API roles named "ACME Underwriter" and "ACME Reinsurance Manager".

For more information on how API roles are configured, see “Endpoint access” on page 373.

Resource access for services with service account mapping

Resource access defines, for a given type of resource, which instances of that resource the caller can access. For example, suppose there is a GET `/claims` endpoint that is available to policyholders, underwriters, adjusters, and service vendors. All of these callers can use the endpoint to access resources whose type is claim, but none of the callers can access all of the claims. For example:

- A policyholder may be able to see only the claims associated with the policies they hold.

- An underwriter may be able to see only the claims for policies assigned to them.
- An adjuster may be able to see only the claims assigned to them.
- A service vendor may be able to see only the claims that have a service request assigned to them.

Resource access is controlled by two features: resource access IDs and resource access strategies.

A *resource access ID* is a string that defines who the caller is or what the caller owns. Resource access IDs are used to determine which resources an authenticated caller has access to through API calls.

- A resource access ID can identify *who the caller is*. For example, the resource access ID for a service provider is the provider's Address Book ID. Typically, a service provider can access all claims which include this ID in the list of associated contact IDs.
- A resource access ID can identify *what the caller owns*. For example, the resource access ID for a policyholder is a list of one or more policy numbers. Typically, a policyholder can access all policies with those policy numbers.

A resource access strategy is a set of logic that identifies the meaning of a resource access ID. The base configuration includes the following resource access strategies for service account:

Strategy name	Persona using this strategy	The resource access ID is assumed to be...	Grants access to...
`pc_username`	Internal users and service accounts	A PolicyCenter account name	Any information this account could see in PolicyCenter based on their associated Access Control Lists (ACLs).

When a service makes a service-with-service-account-mapping call, the service is mapped to a service account name. This account name is used as the resource access ID, and the pc_username strategy is used. This strategy consists of Cloud API logic that matches, as closely as possible, the user's access as defined in the base configuration's Access Control Lists (ACLs).

For more information on how resource access behaves, see “Resource access” on page 385.

Proxy user access for services with service account mapping

Proxy user access is not applicable for services with service account mapping. The service account is used as the user for the session. Therefore, even though the call is coming from a service, there is no need to assign a proxy user.

JWTs for services with service account mapping

JSON Web Tokens (JWTs) contain token claims. (In standard JWT parlance, these are referred to simply as "claims". To avoid confusion with claims in the property and casualty insurance sense, this documentation sometimes refers to JWT claims as "token claims".) A *token claim* is a piece of information asserted about the bearer of the token, such as the bearer's name. For bearer token authentication, authentication information is stored in token claims.

When a service makes a call with service account mapping, only some of the information in the JWT is specific to Cloud API authentication. The following are the token claims in a JWT for a service-with-service-account-mapping call that pertain to Cloud API authentication.

```
"sub": "<clientId>",
"cid": "<clientId>"
```

- `sub` is the subject of the token. This is set to the service's client ID.
- `cid` is the client ID of the service. This is also set to the service's client ID.

For a service-with-service-account-mapping call, the JWT may contain a `scp` token claim. But, unlike calls from standalone services or services with user context, there is no authorization information specific to Cloud API in the `scp` token claim. All authorization information comes from the service account that the service is mapped to.

Mapping services to service accounts

Service account mapping

The *service account mapping* is a set of name/value pairs that map a service's client ID to the name of a user in the PolicyCenter database. This user is referred to as the *service account*.

When a service call is received, Cloud API checks to see if the client ID is listed in the service account mapping.

- If the client ID is found, the call is treated as a service-with-service-account-mapping call. The service account becomes the session user, and the access associated with the service account defines what the service can do.
- If the client ID is not found, the call is treated as either a service-with-user-context call or a standalone service call.

For example, suppose the following service account mapping exists.

Client ID	User Name
0oaqt9pl1vZK1kybt0h7	acmeDocuments
0oapqkzpmaHfIU0sI0h7	acmeCSRPortaleast
0oaer46gh823d777er0x	acmeCSRPortalwest

Suppose a call is received from a service with a client ID of "0oaqt9pl1vZK1kybt0h7". This client ID exists in the service account mapping. Therefore, the call is treated as a service-with-service-account mapping call. The call is associated with the service account whose user name is acmeDocuments.

Similarly, suppose a call is received from a service with a client ID of "0oa33344455566677788". This client ID does not exist in the service account mapping. Therefore, the call is treated as either a service-with-user-context call or a standalone service call.

Every service has a single client ID. Therefore, for a given service, either all calls are treated as service-with-service-account-mapping calls, or none of the calls are treated as service-with-service-account-mapping calls. If the client ID is in the service account mapping, it is the former. If not, it is the latter.

Storing service account mapping

Service account mapping can be stored in different locations. For every service call, Cloud API checks all locations. Cloud API uses the first mapping it finds. So, if a client ID is mapped in multiple places, only the first mapping is used. If a client ID is not listed in any of these locations, Cloud API treats the call as a either a service-with-user-context call or a standalone service call.

From a technical perspective, service account mappings can be spread out across all of these locations. However, insurers may find it easier to manage service account mappings if a single location is used.

Storing service account mapping: Guidewire Cloud Platform

Cloud API checks the Guidewire Cloud Platform (GWCP) variables. Insurers can make these changes on their own through the Variables app in Guidewire Home.

Service mapping entries in GWCP variables use the following syntax:

```
PLUGIN_AUTHENTICATIONVERIFIER_SUBJECTMAPPINGS_<sub>=<username>
```

where:

- *<sub>* is the value of the JWT's sub token claim (which is set to the client ID).
- *<username>* is the user name of the service account.

To deploy changes to GWCP variables, you must restart the server.

> **Note:** When specifying values through the Variables app, the user interface gives you the ability to specify that a given value applies only to a given **Environment**. Guidewire recommends using a single set of service account mapping values for all environments. In other words, when specifying service account mapping values, Guidewire recommends leaving the **Environment** field blank.

Storing service account mapping: config.properties

Then, Cloud API checks **the `config.properties` file**. Entries in this file use the following syntax:

```
plugin.PLUGIN_AUTHENTICATIONVERIFIER_SUBJECTMAPPINGS_<sub>=<username>
```

where:

- *`<sub>`* is the value of the JWT's sub token claim (which is set to the client ID).
- *`<username>`* is the user name of the service account.

To deploy changes to the `config.properties` file, you must restart the server.

Storing service account mapping is `config.properties` may be appropriate in development instances. But, Guidewire does not recommend storing service account mapping is `config.properties` in production instances.

The service account

A service account is intended to be used only by a single external service making Cloud API calls. Typically, the service account name reflects the service (such as `acmeDocuments`) and does not look like a person account name (such as `aapplegate`).

The service account needs to have the permissions appropriate for the service.

Guidewire recommends to not use service accounts for user activity. In other words, do not have users log on to PolicyCenter using a service account.

Logging

For each call, information about the caller is logged. The following table lists the fields that provide information about who the caller is, and where the logged value comes from.

Field	Value
`sub`	The value of the `sub` token claim from the JWT
`clientId`	The value of the `cid` token claim from the JWT
`user`	The user name of the service account

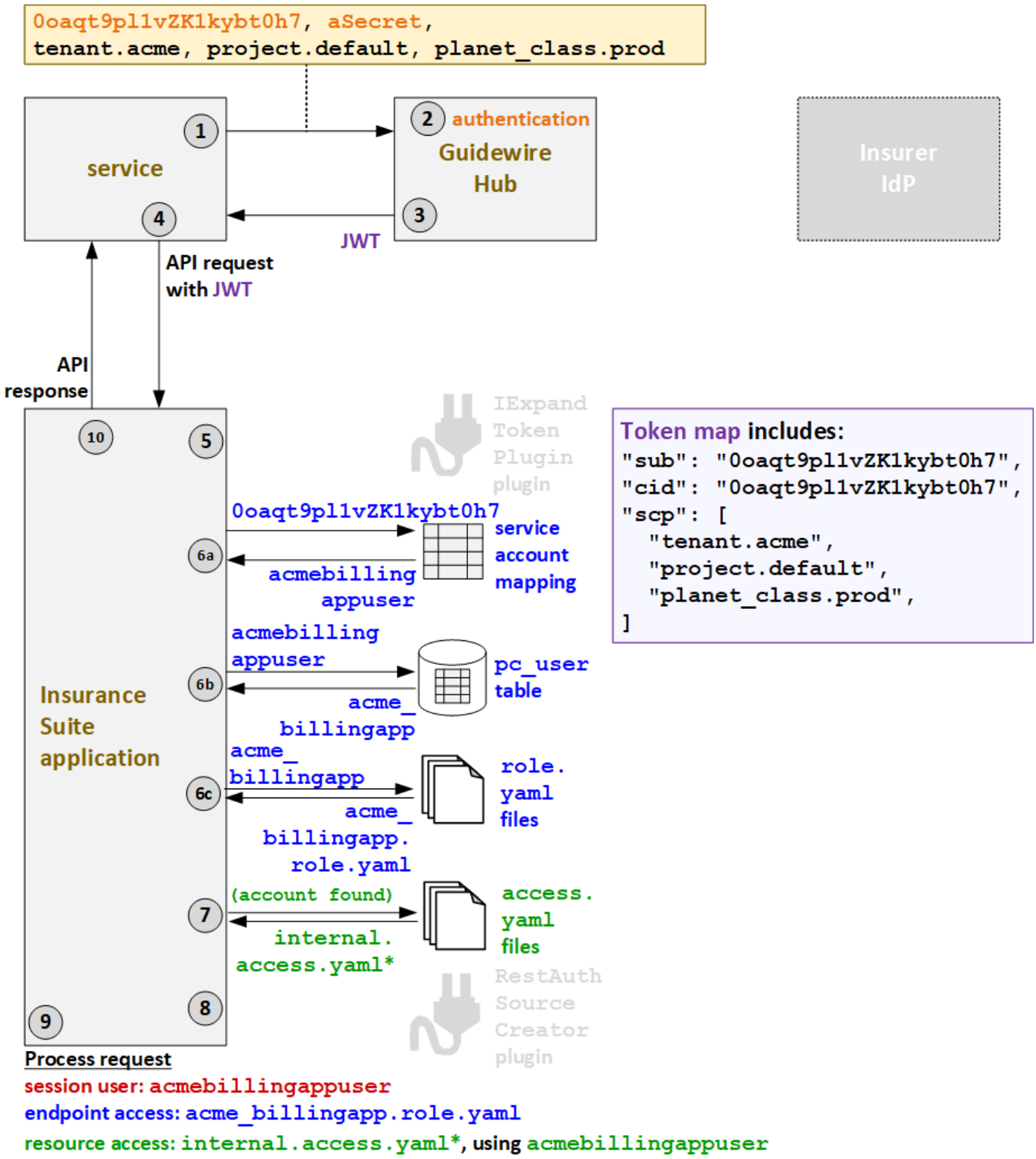
Example flow for services with service account mapping

The following diagram identifies the flow of authentication and authorization information for services with service account mapping. Colors are used in the following ways:

- Orange - credentials information
- Blue - endpoint access information
- Green - resource access information
- Red - proxy user and session user information

Some values are used to determine multiple types of access. These values initially appear as black (when they do not apply to a single type of access), and then later appear in one or more specific colors (to reflect the value is being used at that point in the process for a specific type of access).

In the following example, an API call is triggered by the Acme billing application, BillingApp.



1. When FNOLReporter triggers an API call, it must first request a JWT from Guidewire Hub. The request for the JWT includes the client ID (`0oaqt9pl1vZK1kybt0h7`), the secret (`aSecret`), and additional deployment information (`tenant.acme`, `project.default`, `planet_class.prod`).
2. Guidewire Hub authenticates the services based on the client ID and secret.

3. Guidewire Hub generates a JWT and sends it to the service. This JWT includes the client ID (`cid`) and additional deployment information.
4. The service sends the API request to PolicyCenter along with the JWT.
5. The IExpandTokenPlugin plugin is not relevant for services with service account mapping.
6. PolicyCenter determines the endpoint access.
   a. First, PolicyCenter executes a lookup to map the client ID (`0oaqt9pl1vZK1kybt0h7`) to a service account name (`acmebillingappuser`).
   b. Then, PolicyCenter queries for the user roles that this account name has. One role is returned: `acme_billingapp`.
   c. Based on the returned role, the `acme_billingapp.role.yaml` API role file is used to define the endpoint access.
7. Next, PolicyCenter determines the resource access strategy. Based on the fact that the service account lookup found a valid service account, PolicyCenter grants resource access as defined in the `internal access.yaml` files. (* PolicyCenter starts with `internal_ext-1.0.access.yaml`, but this file references additional `access.yaml` files whose name starts with "`internal`".)
8. Proxy user access is not relevant for services with service account mapping.
9. PolicyCenter processes the request.
   a. The session user is the service account: `acmebillingappuser`.
   b. The endpoint access is defined by `acme_billingapp.role.yaml`.
   c. The resource access is defined by `internal access.yaml` using the resource access ID of `acmebillingappuser`.
10. PolicyCenter provides the response to the initial call.

Implementation checklist for services with service account mapping

To configure Cloud API for authentication for services with service account mapping, you may need to do the following tasks:

Task	More Information
Enable asymmetric encryption	“Enabling bearer token authentication” on page 369
Provide deployment information	“Enabling bearer token authentication” on page 369
Register the caller application with Guidewire Hub	“Enabling bearer token authentication” on page 369
Create the service accounts	“Mapping services to service accounts” on page 353
Create the service account mappings	“Mapping services to service accounts” on page 353
Create or modify API roles	“Endpoint access” on page 373
Review the resource access strategies provided in the base configuration	“Resource access” on page 385

Sending authenticated calls for services with service account mapping

When a caller application wants to make a Cloud API call for a service with service account mapping, the caller application must:

1. Request a JWT from Guidewire Hub
2. Include the JWT with the system API call

Requesting JWTs from Guidewire Hub

For more information on how to request JWTs from Guidewire Hub, refer to *Authentication with Guidewire Identity Federation Hub* in the *Guidewire Cloud Platform* documentation set.

Including JWTs with API calls

Once a JWT has been received from Guidewire Hub, it must be sent to PolicyCenter in the request object's Authorization header. The header must use this format:

```
Authorization: Bearer <token>
```

Authentication failure error messages

For endpoints that return an individual element, when the resource exists but the user lacks authorization to access it, Cloud API throws the following user message. This is the same message that is returned when the resource does not exist.

```
"status": 404,
        "errorCode": "gw.api.rest.exceptions.NotFoundException",
        "userMessage": "No resource was found at path <path>"
```

For endpoints that return collections, Cloud API returns all resources that meet the criteria and for which the user has sufficient resource access. If a resource exists, but the user lacks sufficient authorization, Cloud API omits it from the results.

These approaches are considered to be more secure as they prevent malicious callers from being able to verify the existence of data that they are not authorized to access.

chapter 56

Unauthenticated callers

An *unauthenticated caller* is a user or service who provides no authentication information. Unauthenticated callers can access only metadata endpoints. Unauthenticated callers are typically callers who need information about Cloud API endpoints only.

This topic describes how to implement Cloud API authentication for unauthenticated callers.

> **Note:** Anonymous users start out as unauthenticated callers. This topic focuses on unauthenticated callers who remain unauthenticated and who request Cloud API metadata only. For more information about anonymous users, see "OAuth2 authorization code flow: Anonymous users" on page 313.

Overview of authentication for unauthenticated callers

Typically, authentication includes credentials and authorization. However, unauthenticated callers have no credentials and have a limited set of default authorization.

Credentials

By definition, an unauthenticated user has no credentials.

Authorization

Endpoint access for unauthenticated callers

Endpoint access defines the aspects of an endpoint's behaviors that are available to a caller. This includes:

- What endpoints and resource types are available to the caller?
- What operations can a caller call on the available endpoint?
- What fields can the caller specify in a request payload or get in a response payload?

Endpoint access is controlled by API roles. An *API role* is a list of endpoints, operations, and fields that are available to a set of callers through API calls. API roles act as allowlists. By default, a caller has no endpoint access. When the caller is associated with one or more API roles, they gain access to the endpoints, operations, and fields allowlisted in each of those API roles.

When an unauthenticated caller makes a Cloud API call, Cloud API automatically assigns them the Unauthenticated role. In the base configuration, this role provides two types of access:

- Access to `openapi.json` endpoints

- Access to POST to the `/accounts`, `/accounts/*/contacts`, and `/accounts/*/locations` endpoints. This is part of the authentication flow for anonymous users.
  - Note that these endpoints must be called in a single POST `/accounts` call that uses request inclusion for any contacts and locations. You cannot call the POST `/accounts/*/contacts` or POST `/accounts/*/locations` endpoints on their own.

For more information on how API roles are configured, see "Endpoint access" on page 373.

Resource access for unauthenticated callers

Resource access defines, for a given type of resource, which instances of that resources the caller can access. For example, suppose there is a GET `/claims` endpoint that is available to policyholders, underwriters, adjusters, and service vendors. All of these callers can use the endpoint to access resources whose type is `claim`, but none of the callers can access all of the claims. For example:

- A policyholder may be able to see only the claims associated with the policies they hold.
- An underwriter may be able to see only the claims for policies assigned to them.
- An adjuster may be able to see only the claims assigned to them.
- A service vendor may be able to see only the claims that have a service request assigned to them.

A *resource access strategy* is a set of logic that identifies the meaning of a resource access ID. Unauthenticated callers are automatically assigned the `default` resource access strategy. This strategy does not provide access to any business resources. It provides access to Cloud API metadata only.

Strategy name	Persona using this strategy	The resource access ID is assumed to be...	Grants access to...
`default`	Callers who have presented no resource access strategy	Not applicable	Metadata resources only (information returned by the various `/openapi.json` endpoints)

For more information on how resource access behaves, see "Resource access" on page 385.

Proxy user access for unauthenticated callers

When a caller makes a Cloud API call, the internal PolicyCenter logic may trigger checks that are unrelated to endpoint access or resource access. For example:

- A caller may attempt to assign an activity to themselves. PolicyCenter must check to see if the caller has sufficient permission to own an activity.
- A caller may attempt to create a collision coverage with a deductible less than $1000. PolicyCenter must check to see if the amount of the coverage term is within the caller's authority limit.

Unauthenticated users are not listed in the PolicyCenter operational database, and therefore do not have any system permissions or authority limits tied to them. In the unlikely case that an unauthenticated user triggers an internal check, Cloud API makes use of proxy users. A *proxy user* is an internal user that is assigned to an external user or service when the API call is made. Whenever internal PolicyCenter logic must check to see if the caller has sufficient access, the proxy user is checked.

For more information on how proxy user access behaves, see "Proxy user access" on page 391.

JWTs for unauthenticated callers

An unauthenticated caller has no JWT. They are automatically assigned the Unauthenticated API role, and the `default` resource access strategy. In the base configuration, this provides access to API metadata only.

Logging

For each call, information about the caller is logged. The following table lists the fields that provide information about who the caller is, and where the logged value comes from.

Field	Value
`sub`	The value of the `sub` token claim from the JWT
`clientId`	The value of the `cid` token claim from the JWT
`user`	An empty string

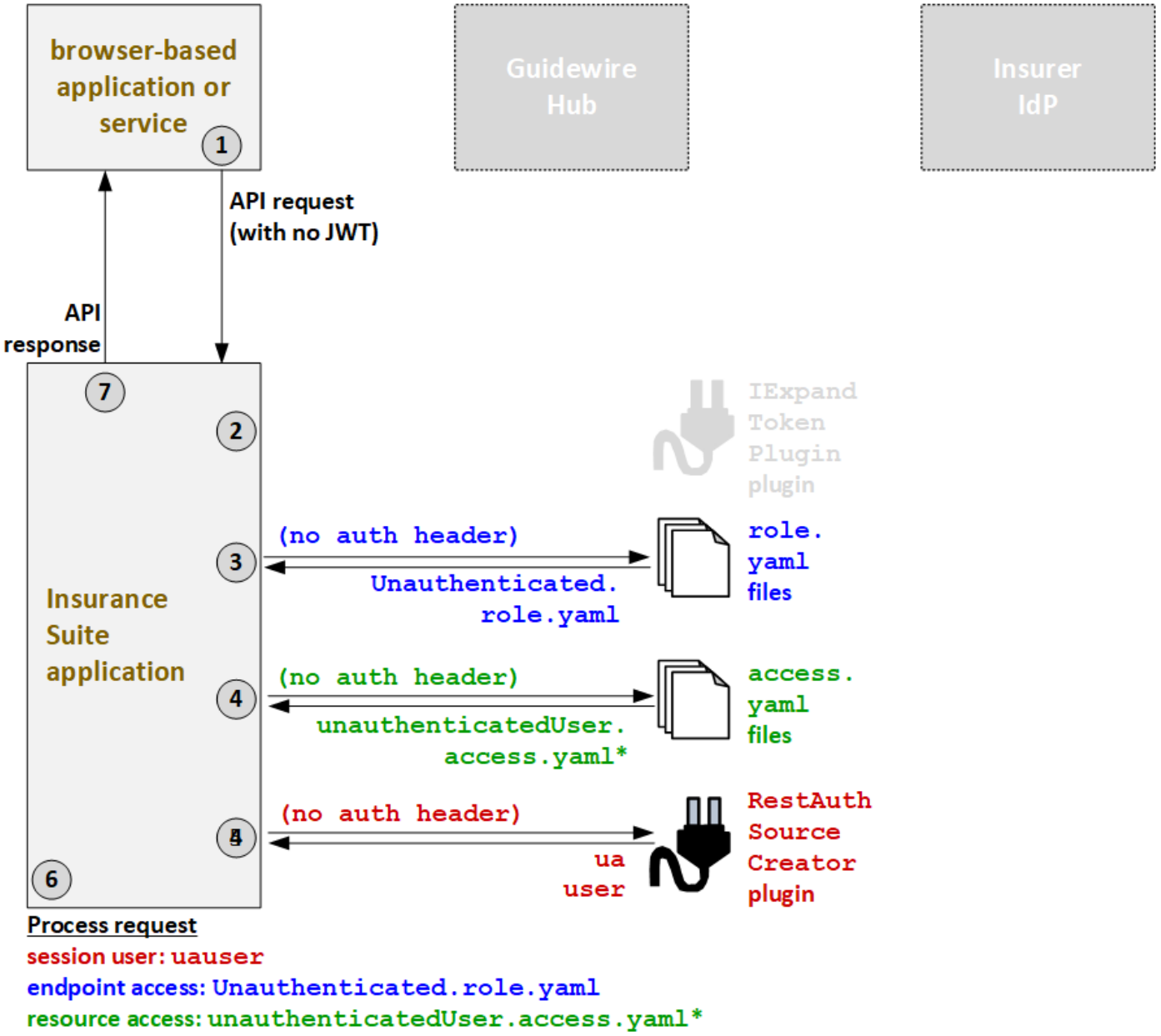
Example flow for unauthenticated callers

The following diagram identifies the flow of authentication and authorization information for unauthenticated callers. Colors are used in the following ways:

- Orange - credentials information
- Blue - endpoint access information
- Green - resource access information
- Red - proxy user and session user information

Some values are used to determine multiple types of access. These values initially appear as black (when they do not apply to a single type of access), and then later appear in one or more specific colors (to reflect the value is being used at that point in the process for a specific type of access).

In the following example, an API call is triggered by an unauthenticated caller.



1. The caller application sends the API request to PolicyCenter. The call includes no JWT, and no authentication information in the header.
2. The IExpandTokenPlugin plugin is not relevant for unauthenticated callers.
3. Because the call has no authentication header, PolicyCenter grants endpoint access as defined in the `Unauthenticated.role.yaml` API role file. (This provides access to metadata endpoints. This also provides access to endpoints that can be used to create a new account. This is part of the anonymous auth flow.)
4. Because the call has no authentication header, PolicyCenter grants resource access as defined in the `unauthenticatedUser.access.yaml` API role file. (This provides no access to existing business resources.)
5. To determine which proxy user to assign to the session, PolicyCenter calls the `RestAuthenticationSourceCreator` plugin. The call has no authentication header. So, the plugin returns the proxy user for unauthenticated users: `uauser`.
6. PolicyCenter processes the request.

a. The session user is the proxy unauthenticated user: `uauser`.

b. The endpoint access is defined by `Unauthenticated.role.yaml`.

c. The resource access is defined by `unauthenticatedUser access.yaml`.

7. PolicyCenter provides the response to the initial call.

Implementation checklist for unauthenticated callers

To configure Cloud API for authentication for unauthenticated callers, you may need to:

1. Review and modify the Unauthenticated API role.
2. Configure the proxy user

To configure Cloud API for authentication for unauthenticated users, you may need to do the following tasks:

Task	More Information
Review and modify the Unauthenticated API role	“Endpoint access” on page 373
Configure the proxy user	“Proxy user access” on page 391

To make a Cloud API call for unauthenticated callers, the caller application does not need to request a code, request a JWT, or include a JWT with the call. The caller simply provides no authentication information.

part 10

Implementing authentication

This section provides information on how to execute each task for the implementation of authentication. This includes:

- How to enable bearer token authentication for a given instance of PolicyCenter
- How to configure endpoint access
- How to configure resource access
- How to configure proxy user access

Configuring access to job types for external account holders

By default, `RestV1JobTypesConfigurationPlugin` provides external account holders access to the following jobs types with the Cloud API for PolicyCenter:

- Submission
- Renewal
- Policy Change
- Cancellation

You can configure external account holders to have access to additional job types.

Provide account holders access to additional job types

Complete the following steps to provide external account holders access to a customized list of job types. The examples in this topic provide external account holders access to the issuance job.

After reading this topic, you'll be able to:

- Create a custom class to implement plugin interface `RestV1JobTypesConfigurationPlugin`
- Add additional endpoint access for account holders
- Add additional resource access for account holders
- Add additional permissions to proxy user `external_user`

Instructions

Step 1: Create a custom class to implement plugin interface RestV1JobTypesConfigurationPlugin

About this task

Create a custom plugin implementation class `RestV1JobTypesConfigurationPluginCustomImpl` to configure the types of jobs that external account holders can access.

Procedure

1. Create a custom plugin implementation class that implements the `RestV1JobTypesConfigurationPlugin` interface.

   For example, create the class named `RestV1JobTypesConfigurationPluginCustomImpl.gs` to implement the `RestV1JobTypesConfigurationPlugin` interface:

   ```
   package gw.rest.ext.pc.security.v1

   uses gw.rest.core.pc.security.v1.RestV1JobTypesConfigurationPlugin

   class RestV1JobTypesConfigurationPluginCustomImpl implements RestV1JobTypesConfigurationPlugin {
     override property get AccessibleJobTypesForAccountHolder() : Set<typekey.Job> {
       // TC_ISSUANCE is added for custom configuration
       // the default supported job types are TC_CANCELLATION, TC_POLICYCHANGE, TC_RENEWAL, TC_SUBMISSION
       return {TC_CANCELLATION, TC_ISSUANCE, TC_POLICYCHANGE, TC_RENEWAL, TC_SUBMISSION}
     }
   }
   ```

   This code provides external account holders access to the issuance job type in addition to the jobs they have access to by default.

2. Update `RestV1JobTypesConfigurationPlugin.gwp` to reference the new implementation class.

   For example, update `RestV1JobTypesConfigurationPlugin.gwp` with the following code:

   ```
   <?xml version="1.0"?>
   ...
   <plugin
     interface="RestV1JobTypesConfigurationPlugin"
     name="RestV1JobTypesConfigurationPlugin">
     <plugin-gosu
       gosuclass="gw.rest.ext.pc.security.v1.RestV1JobTypesConfigurationPluginCustomImpl"/>
   </plugin>
   ...
   ```

Step 2: Update the endpoint access for account holders

About this task

Update the `Account_Holder.role.yaml` file to provide external account holders access to the additional endpoints. In this case, provide account holders access to issue a policy.

Procedure

1. Open the `Account_Holder.role.yaml` file.
2. Add the following code:

   ```
   - endpoint: "/policy/v1/policies/*/issue"
     methods:
     - POST
   ```

Step 3: Update resource access for account holders

About this task

The `accountholder_ext-1.0.access.yaml` defines the resources available to account holders. In this case, update this file to provide account holder access to additional policy information.

Procedure

Update the `config/authorization/gw/core/pc/shared/v1/accountholder_ext-1.0.access.yaml` file with the following code:

```
Issuance:
  permissions:
    view: __inherit
    create: "gw.rest.core.pc.security.v1.AccountHolderSecurityUtil.canAccessPolicy(resource.Parent.Policy)"
```

Step 4: Update the permissions provided to proxy user external_user

About this task

The `roleprivileges.csv` file includes the permissions associated with each PolicyCenter role. In this case, when you provide an account holder access to issuance jobs, you might need to provide the proxy user permissions to issue a policy, quote and bind the Issuance job.

Procedure

1. Open `roleprivileges.csv` and add the following additional items at the end of the file:

   ```
   RolePrivilege,0,sample_data:1698,advanceissuance,external_user
   RolePrivilege,0,sample_data:1699,bindissuance,external_user
   RolePrivilege,0,sample_data:1700,createissuance,external_user
   RolePrivilege,0,sample_data:1701,editissuance,external_user
   ```

2. Verify that the `entityid` values associated with each item are unique and make any updates as needed. For example, if the last item in `roleprivileges.csv` has an `entityid` of sample_data:1697, define the next new row with an `entityid` of sample_data:1698.
3. Use the `import-tools` command to update the permissions. For more information, see the *Administration Guide.*

chapter 57

Enabling bearer token authentication

Insurers must execute several steps to enable bearer token authentication for an instance of PolicyCenter.

1. PolicyCenter must be registered with Guidewire Hub.
2. PolicyCenter must be configured so that it can use asymmetric encryption.
3. PolicyCenter must have its deployment information specified.
4. The IdP must be configured to store and assert information about the users.
   - This is required for internal and external users only.
5. Every caller application must be registered with Guidewire Hub.

This topic details all of the tasks that fall into the category of enabling bearer token authentication.

Enabling asymmetric encryption

Bearer token authentication for Cloud API uses *asymmetric encryption*. To verify a given JWT, PolicyCenter executes an *asymmetric public key lookup*. Periodically, PolicyCenter must request the keys used in these lookups from Guidewire Hub.

When you register PolicyCenter with Guidewire Hub, you are given an auth server URI and a tenant ID. For PolicyCenter to be able to request keys from Guidewire Hub, you must add the auth server URI to your PolicyCenter instance.

Failing to enable asymmetric encryption

If you do not enable asymmetric encryption, then calls that attempt to use bearer token authentication will be rejected with a message similar to the following:

```
JWT verification failed: Encountered JWT issuer '<URL>' that has not been configured in
       'allowedIssuers' by the SignatureKeyProviderPlugin. Allowed issuers are []
```

Enable asymmetric encryption

About this task

Before you can complete this task, you must have the issuer URI. This value is supplied to you by Guidewire.

The following steps identify how to complete this task in your instance of PolicyCenter. It may also be possible to complete this task by storing the `authServerUri` in Guidewire Cloud Property Services. For more information, talk to Guidewire.

Note: The auth server URI is used by the `SignatureKeyProviderPluginV1` plugin. In the base configuration, the plugin registry reads the value from the PolicyCenter `config.properties` file. Therefore, these instructions indicate how to modify the value in the properties file. If you have modified your configuration to read the value from other locations, then you will need to change the value in those locations as needed.

Procedure

1. In Guidewire Studio, navigate to **configuration** > **config**, and open `config.properties`.
2. Add the following line to the file. (Note that this line may already be in the file as a comment. If so, you can simply uncomment the line.) `plugin.signaturekeyprovider.allowedissuers =`
3. Set the value of the `allowedissuers` properties to the value of the `authServerUri` provided to you by Guidewire.
4. Restart the application.

Specifying deployment information

When processing an API call that uses bearer token authentication, Cloud API verifies that the tenant, project, and planet class specified in the JWT match the tenant, project, and planet class of this instance of PolicyCenter.

This check is accomplished using deployment IDs. A *deployment ID* is a string that specifies, for a given instance of PolicyCenter, the tenant, project, and planet class than the instance belongs to. Deployment IDs are formatted using *GRN* (Guidewire Resource Notation).

If you want bearer token authentication to work for a given instance of PolicyCenter, the instance's deployment ID must be specified prior to starting the instance. If the deployment ID is not specified, every call using bearer token authentication is returned with a null pointer exception.

For more information on deployment IDs, see the *Guidewire Cloud Platform* documentation.

The "lower" wild card for planet classes

Within the context of Guidewire Resource Notation, there are several possible values for planet class. However, within the context of Cloud API JWTs, there are only three possible values: "prod", "preprod", and "lower". The token values of "prod" and "preprod" map to the GRN values of "prod" and "preprod". The token value of "lower" maps to all other GRN planet class values. Therefore:

- A JWT that specifies a planet class of "prod" will work with applications whose planet class is "prod".
- A JWT that specifies a planet class of "preprod" will work with applications whose planet class is "preprod".
- A JWT that specifies a planet class of "lower" will work with applications whose planet class is any value other than "prod" or "preprod".

Configuring the IdP

For internal users, the IdP must store:

- The user's credentials (for example, user name and password)

For external users, the IdP must store:

- The user's credentials (for example, user name and password)
- Either:
  - The list of API roles that are to be granted to the user and the user's resource access IDs, OR
  - A lookup value that can be used by the IExpandTokenPlugin plugin to retrieve the user's list of API roles and resource access IDs from an additional authorization application

The IdP must provide its information to Guidewire Hub when it asserts the user's identity.

Note: The IdP is relevant only for the internal user auth flow and external user auth flow. The service auth flows (standalone service, service with user context, and service with service account mapping) do not make use of an IdP. When you implement a service flow, there are no IdP requirements.

Configure the IdP for internal users

Before you begin

This procedure is necessarily only when using the internal user auth flow. It is not necessary for the service with user context flow, even if the user in the user context is an internal user.

Procedure

1. Configure your IdP so that every internal user is associated with their user credentials (such as user name and password).
2. Configure your IdP so that when an internal user is verified, the authorization information is asserted using the following attribute names:
   - User name is asserted as `pc_username`.

Configure the IdP for external users

Before you begin

This procedure is necessarily only when using the external user auth flow. It is not necessary for the service with user context flow, even if the user in the user context is an external user.

For Cloud API to determine a user's endpoint access and resource access, it must know the user's API roles and resource access IDs. Either or both sets of values can be stored in the IdP. Alternately, either or both sets of values can be stored in an additional authorization application. Whenever the values are stored in an additional authorization application, the IdP must still provide some sort of lookup value that the IExpandTokenPlugin plugin can use to retrieve the information from the additional authorization application, such as the user's name as it is known to the additional authorization application.

Procedure

1. Configure your IdP so that every external user is associated with their user credentials (such as user name and password).
2. If you store API roles in the IdP, configure your IdP so that it knows all of the API roles that are assigned to any external user.
   - Typically, this is done with IdP groups.
   - Each group name must be prefixed with "`gwa.<planetclass>.pc.`", where `<planetclass>` is set to either "`prod`", "`preprod`", or "`lower`".
   - After this prefix, each group name must be identical to a Cloud API role name.
   - For example, to assign users to an API role named "Account_Holders" for a production planet, the IdP group must be named "`gwa.prod.pc.Account_Holders`".
3. If you store API roles in the IdP, configure your IdP so that every external user is associated with their API roles.
4. If you store resource access IDs in the IdP, configure your IdP so that every external user is associated with the correct resource access IDs:
   - For account holders, this is an array of one or more account numbers.
   - For producers, this is an array of one or more set of producer codes and roles. numbers.
5. If you store API roles and/or resource access IDs in the IDP, configure your IdP so that when an external user is verified, the authorization information is asserted using the following attribute names:

- API roles are asserted as an array named groups.
- Resource access IDs for account holders are asserted as an array named `pc_accountNumbers`.
- Resource access IDs for producers are asserted as an array named `pc_producerCodes`.

6. If you do not store either API roles and/or resource access IDs in the IDP, configure your IdP so that when an external user is verified, the SAML response includes one or more lookup values that the IExpandTokenPlugin plugin can use to retrieve either API roles and/or resource access IDs from the appropriate additional authorization application. For more information on configuring the IExpandTokenPlugin plugin, see "Configuring the IExpandTokenPlugin plugin" on page 395.

Registering the caller application with Guidewire Hub

Every caller application must be registered with Guidewire Hub. The information provided during the registration process varies based on whether the application is a browser-based application or a service application.

Note: This information applies to applications that are not registered with Integration Gateway. If your application is integrated with Integration Gateway, see *Integration Gateway for Developers* in the *Guidewire Cloud Platform* documentation set for information on how to register the application.

Note: If you are using Jutro Web Apps on the Jutro Development Platform, you do not need to register the application with Guidewire Hub as described in the steps below. Instead, there is a self-service tool you can use to register the application. For more information, see the *Jutro Digital Platform* documentation.

Register an application with Guidewire Hub

About this task

Before an application can request JWTs, the application must be registered with Guidewire Hub.

Procedure

1. Determine which auth flow the application will use. The auth flow must be one of the following:
   - Front end application (auth code flow for internal users and/or external users). If choosing this option, you must also specify whether you plan to use PKCE or a client secret.
   - Standalone service
   - Service with user context
   - Service with service account mapping
2. Contact Guidewire and specify that you need an "InsuranceSuite REST API registration with Guidewire Hub" using the desired auth flow.
3. Guidewire sends a list of required information based on the selected auth flow. Provide this information to Guidewire.
4. Guidewire registers the application for OAuth based on the information provided. They will also send information to you that you need to further configure authentication, such as a client ID and client secret.

Results

Once you have the authorization information from Guidewire, you can proceed with authentication configuration.

For further information on the difference between auth code flow with PKCE and auth code flow with client secret, refer to *Authentication with Guidewire Identity Federation Hub* in the *Guidewire Cloud Platform* documentation set.

chapter 58

Endpoint access

Endpoint access is defined by API roles. An *API role* is a list of endpoints, operations, and fields that are available to a set of callers when triggering Cloud API calls. For example, API roles determine the following:

- What endpoints and resource types are available to the caller?
  - For example, can a given caller access the `/activities` endpoint?
- What methods can a caller call on the available endpoint?
  - For example, can a caller execute both a GET and a POST on the `/activities` endpoint?
- What fields can the caller specify in a request payload or get in a response payload?
  - For example, can a caller include the `priority` field in a POST `/activities` or retrieve the `assignedUser` in a GET `/activities`?

  Note: PolicyCenter includes an "unrestricted user". This user behaves as if it has all permissions. (In the base configuration, the unrestricted user is `su`.) The unrestricted user is not bound by endpoint access. Any authenticated call from the unrestricted user automatically has access to all endpoints.

API role files

API roles are defined in YAML files that are named *`RoleName`*`.role.yaml`. For example:

- `Account_Holder.role.yaml`
- `External_Producer_Code.role.yaml`
- `Producer.role.yaml`
- `Underwriter.role.yaml`

Location of API role files

API roles are declared in Studio in the **Integration** > **roles** directory. When checking for role files, Cloud API looks in this directory only (and not in any subdirectories of the **roles** directory).

Note: In most cases when an insurer adds files to an instance of PolicyCenter, the best practice is to add these files to a subdirectory that is named after the insurer. However, an insurer cannot follow this practice with role files. For a role file to be available to Cloud API, the file must be declared directly in the **roles** directory. It cannot be declared in a subdirectory.

Structure of API role files

API roles are declared in Studio in the **Integration** > **roles** directory. Every file identifies:

- The role name
- The endpoints and endpoint operations for associated users
- The fields that associated users can view or edit
- Any special permission granted to associated callers

Note that these parts can be listed in any order.

API role names

Guidewire recommends using the same string of characters for the role name declared in the file and the role name as it appears in the file name. If an API role name must be multiple words, the file name must use underscores (such as `User_Admin.role.yaml`). Guidewire recommends using the same string in the file's `Role Name` section, but with spaces instead of underscores (such as `name: "User Admin"`).

For example, if a new API role is for Fraud Investigators:

- Name the file `Fraud_Investigator.role.yaml`.
- In the file, declare the name as `name: "Fraud Investigator"`.

For API roles for internal users:

- There must be a user role in PolicyCenter assigned to the appropriate users.
- The API role name and the user role name must be the same.

For API roles for external users:

- The IdP must be able to associate each user with the API role name.
- The IdP must identify the role with a "cc.", "pc.", or "bc." substring followed by the role name.
  - The string in the IdP that identifies the role must start with `cc.`/`pc.`/`bc.`. The substring after the `cc.`/`pc.`/`bc.` must match the role name. For example, "cc.Manager" will be matched with the role named "Manager".)

For API roles for services:

- Guidewire recommends you name the role *insurercode_name* (such as `acme_locationphotos`), where:
  - *insurercode* is an insurer code, such as `acme`.
  - *name* is a meaningful name in lower case.
- When the service is registered with Guidewire Hub, the role must be named with an initial `cc.`/`pc.`/`bc.`. But do not include the prefix in the API role file name or in the `Role Name` section.

Be aware that the naming convention for API roles for services is also used for:

- API roles for Guidewire services.
- API roles for other Guidewire functionality that connect to PolicyCenter.

API role endpoints

The `endpoints` section identifies the endpoints a grantee can use and the methods (GET, POST, PATCH, or DELETE) that a grantee can use on that endpoint. This section acts as an allowlist. By default, a caller cannot use any operation on any endpoint. To enable endpoint use, each endpoint and method must be explicitly allowlisted.

The `endpoints` section contains a list of endpoints in the following pattern:

```
endpoints:
  - endpoint: <endpoint 1>
    methods:
    - <method 1 on endpoint 1>
    - <method 2 on endpoint 1>
  - endpoint: <endpoint 2>
    methods:
    - <method 1 on endpoint 2>
    - <method 2 on endpoint 2>
```

Wildcards in the endpoints section

You can use the asterisk (*) wildcard in the `endpoints` section.

A single * wildcard indicates access is provided for anything one level below the current endpoint level. For example:

- `/common/v1/activities/*` means "anything one level below `/activities`".
- `/common/v1/activities/*/notes` means "the notes for anything one level below `/activities`".

A double ** wildcard indicates access is provided for anything below the current level. For example:

- `/common/v1/activities/**` means "any resource or endpoint that can be accessed from the `/common/v1/activities path`".

If you use a * wildcard to grant specific access to any resource one level below an endpoint, you can still also grant different access to other named resources on that same level. The named resource's access will have priority over the * wildcard. For example:

```
endpoints:
- endpoint: /api/v1/resource1/*
  methods:
  - GET
  - PATCH
- endpoint: /api/v1/resource1/subresource1
  methods:
  - GET
  - DELETE
```

In this example, the role grants GET and PATCH access to any resource one level below `resource1`. However, the role grants GET and DELETE access to `subresource1`, which is also one level below `resource1`, because named resources like `subresource1` have priority over * wildcards. The role would not grant PATCH access to `subresource1` because `subresource1` has its own specified methods.

Exercise caution when using **

Guidewire recommends that insurers exercise caution when using the ** wildcard. This is because later releases of Cloud API may add new endpoints that users will unexpectedly have access to through ** wildcards. For example, suppose in release 1.0 that an insurer creates an API role that provides access to `common/v1/activities/**`. As of release 1.0, this provides access to the following:

- `common/v1/activities/{activityId}`
- `common/v1/activities/{activityId}/assignees`
- `common/v1/activities/{activityId}/notes`

Then, in release 2.0, Guidewire adds the following endpoint:

- `common/v1/activities/{activityId}/confidentialAnalysis`

The API role will automatically have access to the new endpoint, even if this is not what the insurer intended when creating the API role for release 1.0.

API role accessible fields

The `accessibleFields` section identifies the fields of each resource that a grantee can view or edit. This section acts as a allowlist. By default, a caller cannot view or edit any fields on any resource. To enable viewing and editing, each resource, field, and permission must be explicitly allowlisted.

The `accessibleFields` section contains a list of resources in the following pattern:

```
accessibleFields:
  <Resource 1>:
    edit:
    - <fields the grantee can edit on resource 1>
    view:
    - <fields the grantee can view on resource 1>
  <Resource 2>:
```

```
    edit:
    - <fields the grantee can edit on resource 2>
    view:
    - <fields the grantee can view on resource 2>
```

Allowlisting resources

Resources can be named in several ways. You can name the resource explicitly. For example, the following specifies permissions for the Activity resource only:

```
accessibleFields:
  Activity:
    edit:
    - <fields the grantee can edit on this resource>
    view:
    - <fields the grantee can view on this resource>
```

You can also use the "*" wildcard. In this context, it means "all resources available to the endpoints listed in the endpoints section". For example, the following specifies permissions for all resources available to the role's endpoints:

```
accessibleFields:
  "*":
    edit:
    - <fields the grantee can edit on this resource>
    view:
    - <fields the grantee can view on this resource>
```

Allowlisting fields

For every resource, you can specify two field-level permissions: `edit` and `view`. If a permission is not explicitly listed, then callers will not have that permission for any fields on the resource.

Field-level permissions can be named in several ways. You can explicitly name the field and permission. For example, the following grants edit access to the `Activity` resource's `subject` field , and view access to priority `field` and the `subject` field.

```
accessibleFields:
  Activity:
    edit:
    - "subject"
    view:
    - "priority"
    - "subject"
```

You can also use the "*" wildcard. In this context, it means "all fields". For example, the following grants edit access to the `subject` field on the `Activity` resource, and view access to all fields.

```
accessibleFields:
  Activity:
    edit:
    - "subject"
    view:
    - "*"
```

Some resource schemas tag individual fields with a security level of `internal`, `sensitive`, or `public`. When specifying field permissions, you can use the expression "*<*level*>" to indicate "all fields on the resource that have the specified level". For example, the following grants access to fields on the `Job` resource. The grantee can edit and view all fields on the `Job` resource that have a security level of public as well as the `jobFilter` field (which presumably does not have a security level of public).

```
accessibleFields:
  Job:
    edit:
    - "*public"
    - jobFilter
    view:
    - "*public"
    - jobFilter
```

For more information on security levels, see “Security levels” on page 401.

API role special permissions

A *special permission* is a permission that does not operate at the endpoint, endpoint method, or field level. The special permissions supported in this release are defined in the following table.

Permission Name	Applicable Applications	Description
`restcreateautomatedactivity`	ClaimCenter	Provides associated callers the ability to create activities using AutomatedOnly activity patterns.
`restdefervalidation`	PolicyCenter	Provides associated callers the ability to use the `deferValidation` query parameter. For more information, see "Synchronization and deferred validation".
`delintdoc`	PolicyCenter	For external producer authentication, it provides associated callers the ability to delete internal notes on a policy.
`delsensdoc`	PolicyCenter	For external producer authentication, it provides associated callers the ability to delete sensitive documents on a policy.
`editintdoc`	PolicyCenter	For external producer authentication, it provides associated callers the ability to edit internal documents on a policy.
`editintnote`	PolicyCenter	For external producer authentication, it provides associated callers the ability to edit internal notes on a policy.
`editsensdoc`	PolicyCenter	For external producer authentication, it provides associated callers the ability to edit sensitive documents on a policy.
`editsensnote`	PolicyCenter	For external producer authentication, it provides associated callers the ability to edit sensitive notes on a policy.
`viewintdoc`	PolicyCenter	For external producer authentication, it provides associated callers the ability to view internal documents on a policy.
`viewintnote`	PolicyCenter	For external producer authentication, it provides associated callers the ability to view internal notes on a policy.
`viewsensdoc`	PolicyCenter	For external producer authentication, it provides associated callers the ability to view sensitive documents on a policy.
`viewsensnote`	PolicyCenter	For external producer authentication, it provides associated callers the ability to view sensitive notes on a policy.
`restunmasktaxid`	BillingCenter, ClaimCenter, PolicyCenter, ContactManager	In responses that include a contact's `taxId` field, returns the complete and unmasked `taxId`

To add special permissions to a `role.yaml` file, use the following syntax:

```
permissions:
- <permissionName>
```

For example, the following grants the `restunmasktaxid` permission to the role.

```
permissions:
- restunmasktaxid
```

API role example

For example, the following is a portion of the contents of the `Underwriter.role.yaml` file:

```
name: Underwriter
endpoints:
  - endpoint: /account/v1/accounts
    methods:
    - GET
    - POST
  - endpoint: "/account/v1/accounts/*"
    methods:
    - GET
    - PATCH
  - endpoint: "/account/v1/accounts/*/activities"
    methods:
    - GET
    - POST
    ...
accessibleFields:
  "*":
    view: "*"
    edit: "*"
```

Based on this portion, note the following:

- A user with the Underwriter role can use:
  - GET and POST on the `/account/v1/accounts` endpoint
  - GET and PATCH on all endpoints one level below `/account/v1/accounts`
  - GET and POST on the activities for all resource one level below `/account/v1/accounts`
- A user with the Underwriter role can view and edit all fields on the available endpoints.

Assigning API roles to callers

The manner in which API roles are assigned to a caller depends on the type of caller.

Assigning API roles to internal users

An *internal user* is a person who is listed as a user in the InsuranceSuite application's operational database. For example:

- Andy Applegate, a ClaimCenter adjuster
- Alice Applegate, a PolicyCenter underwriter
- Aaron Applegate, a BillingCenter clerk

When an internal user makes a Cloud API call (using either basic authentication or bearer token authentication), PolicyCenter queries the operational database for this internal user's user roles. The user is given endpoint access to all API roles whose names correspond to the names of the user's user roles.

For example, suppose that Alice Applegate is an internal user with two user roles: Underwriter and Reinsurance Manager. Alice Applegate triggers a Cloud API call. When the API call is received, PolicyCenter queries the database for Alice's user roles. Two user roles are returned: Underwriter and Reinsurance Manager. PolicyCenter then grants Alice the endpoint access defined in the API roles named "Underwriter" and "Reinsurance Manager".

API roles and PolicyCenter user roles

For internal users, there are two sets of roles that are used to enable endpoint access. For each logical role, there is a PolicyCenter user role and an API role with the same name. The API role provides endpoint access comparable to the user role.

The following table compares the two types of roles.

Type of role	What does the role specify?	For internal users logging directly in to PolicyCenter...	For internal users who trigger a Cloud API call...	Where is the role configured?
InsuranceSuite user role	A set of system permissions	This specifies what the user can do through the PolicyCenter user interface	This is used to determine which API roles to assign to the user	The **Roles** screen on the PolicyCenter **Admin** tab
API role	A list of accessible endpoints, methods, and fields	Not applicable	This specifies the endpoint access provided to the user	A set of YAML files in Studio

Assigning API roles to external users

An *external user* is a person who is known to the insurer but who is not listed as a user in the PolicyCenter operational database.

When external users make API calls, the call includes a JWT (JSON Web Token). This JWT contains authentication information about the caller, including the API roles to assign to the caller.

Parsing API role information

When PolicyCenter receives a JWT, it extracts the information into a "token map". It then calls the IExpandTokenPlugin plugin, which may add or modify values in the token map.

Then, PolicyCenter looks in the token map for the API roles to grant. For external users, this information is in the `groups` claim. Any value in this claim is assumed to be an API role if it starts with "`gwa.<orbit>.<xc>.`", where `<orbit>` is set to either "`prod`", "`preprod`", or "`lower`", and where <*xc*> is the application code ("cc", "pc", "bc", or "ab" for ContactManager). For each value, PolicyCenter does the following:

1. It confirms that the application and orbit (previously called planet class) referenced in the "`gwa.<orbit>.<xc>.`" substring match the application being accessed.
2. It strips off the "`gwa.<orbit>.<xc>.`" substring.
3. It converts any blanks in the remaining to string to underscores.
4. It then searches for an API role file with the same name.

For example, suppose there is a token map with a `groups` token claim that contains one string: "gwa.prod.pc.Customer Service Representative". PolicyCenter removes the initial "gwa.prod.pc." and converts the spaces to underscores, resulting in the string "Customer_Service_Representative". It then searches for an API role whose file name is "Customer_Service_Representative.role.yaml".

If there are no matches between the resulting strings and the API role names, the caller is given no endpoint access.

If there are multiple matches between the resulting substrings and API role names, the caller is given the union of the access specified in all matching roles. In other words, the API roles are ANDed together.

Assigning API roles to standalone services

A *standalone service* is a service-to-service application that executes the call as itself. It does not execute the call on behalf of a specific person or through a PolicyCenter user account.

When standalone services make API calls, the call includes a JWT (JSON Web Token). This JWT contains authentication information about the caller, including the API roles to assign to the caller.

Parsing API role information

When PolicyCenter receives a JWT, it extracts the information into a "token map". It then calls the IExpandTokenPlugin plugin, which may add or modify values in the token map.

Then, PolicyCenter looks in the token map for the API roles to grant. For standalone services, this information is in the `scp` token claim. Any value in this token claim is assumed to be an API role if it starts with "`scp.<xc>.`", where <*xc*> is the application code ("cc", "pc", or "bc"). For each value, PolicyCenter does the following:

1. It strips off the prefix "`scp.<xc>.`" substring.
2. It converts any blanks in the remaining to string to underscores.
3. It then searches for an API role file with the same name.

For example, suppose there is a token map with a `scp` token claim that contains the following string: "`scp.pc.Document Viewer`". PolicyCenter removes the initial "`scp.pc.`" and converts the spaces to underscores, resulting in the string "Document_Viewer". It then searches for an API role whose file name is "Document_Viewer.role.yaml".

If there are no matches between the resulting strings and the API role names, the caller is given no endpoint access.

If there are multiple matches between the resulting substrings and API role names, the caller is given the union of the access specified in all matching roles. In other words, the API roles are ANDed together.

Assigning API roles to services with user context

A *service with user context* is a service-to-service application that presents information about itself and about a specific user. The call is able to do only the things that both the service by itself could do and the user by itself could do.

When services with user context make API calls, the call includes a JWT (JSON Web Token) and a user context header. The JWT contains authentication information about the caller, including the API roles associated with the service. The user context header contains authorization information about the user, including the API roles associated with the user.

Parsing API role information

When PolicyCenter receives a JWT, it extracts the information into a "token map". It then calls the IExpandTokenPlugin plugin, which may add or modify values in the token map.

Then, PolicyCenter looks in the token map for the API roles to grant. For services with user context, this information is in the `scp` token claim. Any value in the appropriate token claim is assumed to be an API role if it starts with "`scp.<xc>.`", where <*xc*> is the application code ("cc", "pc", or "bc"). For each value, PolicyCenter does the following:

1. It strips off the prefix "`scp.<xc>.`" substring.
2. It converts any blanks in the remaining to string to underscores.
3. It then searches for an API role file with the same name.

Parsing API role information in the user context header (internal users)

When PolicyCenter receives a request with a user context header, it looks for the API roles to grant. If the user context specifies an internal user, PolicyCenter retrieves the user's name from the <*xc*>_username token, where <*xc*> is the application code ("cc", "pc", or "bc"). Then, PolicyCenter queries the operational database for this internal user's user roles. The user is given endpoint access to all API roles whose names correspond to the names of the user's user roles.

Parsing API role information in the user context header (external users)

When PolicyCenter receives a request with a user context header, it looks for the API roles to grant. If the user context specifies an external user, PolicyCenter checks the `groups` token in the user context header. Any value in this token claim is assumed to be an API role if it starts with "`gwa.<orbit>.<xc>.`", where `<orbit>` is set to either "`prod`", "`preprod`", or "`lower`", and where <*xc*> is the application code ("cc", "pc", or "bc"). For each value, PolicyCenter does the following:

1. It strips off the prefix "`gwa.<orbit>.<xc>.`" substring.
2. It converts any blanks in the remaining to string to underscores.
3. It then searches for an API role file with the same name.

Assigning API roles

PolicyCenter assigns the caller endpoint access to all endpoints, methods, and fields listed in both the service's API roles and the user's API roles.

For example, suppose a service with user context sends a call with the following:

- A JWT with a `scp` token claim that contains the following string: "`scp.pc.Document Editor`".
- A user context header that specifies an external user with a `groups` token claim that contains the following string: "`gwa.prod.pc.Document Viewer`".

First, PolicyCenter determines the service's API roles. To do this, it converts the JWT into a token map. Then, it removes the initial "`scp.pc.`" from the token map's `scp` claim and converts the spaces to underscores, resulting in the string "Document_Editor". It then searches for an API role whose file name is "Document_Editor.role.yaml".

Second, PolicyCenter determines the user's API roles. To do this, it removes the initial "`gwa.<orbit>.<xc>.`" from the user context header's `groups` token claim and converts the spaces to underscores, resulting in the string "Document_Viewer". It then searches for an API role whose file name is "Document_Viewer.role.yaml".

Finally, PolicyCenter grants endpoint access to:

- Every endpoint listed in both "Document_Editor.role.yaml" and "Document_Viewer.role.yaml".
- Every method on the endpoints that are listed in both "Document_Editor.role.yaml" and "Document_Viewer.role.yaml".
- Every field on the endpoint resources that are listed in both "Document_Editor.role.yaml" and "Document_Viewer.role.yaml".

Assigning API roles to services with service account mapping

A *service with service account mapping* is a service-to-service application that is mapped to an account in the PolicyCenter database and has access as determined by that account.

When a service with service account mapping makes a Cloud API call, it presents a client ID. PolicyCenter maps that client ID to a *service account* (a user account that is not used by any person, but rather is used exclusively by the service it is mapped to). PolicyCenter then queries the operational database for this service account's user roles. The service is given endpoint access to all API roles whose names correspond to the names of the service account's user roles.

For example, suppose that ACME Documents is a service with service account mapping. It is mapped to a service account with two user roles: Document Viewer and Document Editor. ACME Documents triggers a Cloud API call. When the API call is received, PolicyCenter maps the service to its service account and then queries the database for the service account's user roles. Two user roles are returned: Document Viewer and Document Editor. PolicyCenter then grants ACME Documents the endpoint access defined in the API roles named "Document Viewer" and "Document Editor".

API roles and PolicyCenter user roles

For service accounts, there are two sets of roles that are used to enable endpoint access. For each logical role, there is a PolicyCenter user role and an API role with the same name. The API role provides endpoint access comparable to the user role.

The following table compares the two types of roles.

Type of role	What does the role specify?	If the service were to directly log in to PolicyCenter...	For internal users who trigger a Cloud API call...	Where is the role configured?
InsuranceSuite user role	A set of system permissions	This specifies what the service account could do if it were to log in to the PolicyCenter user interface	This is used to determine which API roles to assign to the service	The **Roles** screen on the PolicyCenter **Admin** tab

Type of role	What does the role specify?	If the service were to directly log in to PolicyCenter...	For internal users who trigger a Cloud API call...	Where is the role configured?
API role	A list of accessible endpoints, methods, and fields	Not applicable	This specifies the endpoint access provided to the service	A set of YAML files in Studio

Assigning API roles to other types of callers

An *unauthenticated caller* is a user or service who provides no authentication information. Unauthenticated callers can access only metadata endpoints and the endpoints to create an account. Unauthenticated callers are automatically assigned the API role named `Unauthenticated`.

An *anonymous user* is a person who is not yet known to the insurer but who may establish a business relationship with the insurer. Typically, an anonymous user can only create an account (and its associated objects), quote a submission, and bind a submission. Once an anonymous user binds a submission, they logically move from being an anonymous user to an external user.

Every anonymous user starts out as an unauthenticated user. After the user creates an account, PolicyCenter generates a self-signed JWT. This JWT grants the user access to the API role named `anonymous`.

Reserved roles

Cloud API has *reserved roles*, which are roles used either by Cloud API or by other Guidewire features or services. Use caution when modifying these roles, as modifications may prevent the relevant Guidewire feature or service from behaving as expected.

Reserved roles for special types of Cloud API callers

Cloud API includes the following reserved roles for special types of callers:

- Unauthenticated.role.yaml - This role defines access for callers who provide no authentication information.
- anonymous.role.yaml - This role defines access for anonymous users. (These are users who were initially unauthenticated but who created an account and may now create, quote, and bind policies for that account.)

Reserved roles for other Guidewire services and features

Cloud API includes a set of reserved roles that start with "gw_". These roles are for Guidewire services and features outside of Cloud API.

Designing API role files

API roles are typically designed either for multiple users or for a single service.

API roles designed for internal users and external users, such as Adjuster, Underwriter, Insured, or Account_Holder, are typically associated with multiple users. This is because there can be hundreds of users that need the same endpoint access. It is more efficient to design API roles that are reusable.

API roles designed for services are typically associated with a single service. This is because each instance of PolicyCenter interacts with a relatively small number of services and each service has its own access requirements. It is more efficient to create one role for each service rather than trying to define multiple, reusable roles.

Configuring API roles

You can modify the base configuration API role files, and you can create new ones.

To deploy changes to API role files (either modifying an existing one or creating a new one):

- In development mode, you can hotswap the files.

- In production mode, you must restart PolicyCenter.

Create an API role file

Procedure

1. In Guidewire Studio, navigate to **configuration** > **config** > **Integration** > **roles**.
2. Right-click the **roles** folder, and then select **New** > **File**.
   - In order for the file to be available to Cloud API, the file must be declared directly in the **roles** folder. It cannot be declared in a subfolder.
3. In the **New File** dialog, enter the name of the role file. Name the file *`RoleName`*`.role.yaml`.
4. Specify the role name as: `name: `*`roleName`*.
5. Specify the endpoints, operations, and fields that the role grants. You can use the base configuration API role files as a reference.
6. Deploy the file by either hotswapping or restarting PolicyCenter.

Modify an API role file

Procedure

1. In Guidewire Studio, navigate to **configuration** > **config** > **Integration** > **roles** and open the file.
2. Modify the file as needed.
3. Deploy the file by either hotswapping or restarting PolicyCenter.

API roles and lookup performance

Whenever Cloud API receives a JWT, it must look up the roles on the JWT and then attempt to match them to `role.yaml` files. For internal users, this lookup involves a query of the `User` table for the user's roles. The lookup may encounter roles that are either in the JWT or returned by the `User` table query for which they are no corresponding `role.yaml` files. The lookup may also have to resolve roles whose names have been translated.

To improve the performance of this lookup, the role names for the default language and US English are now cached by PLDependencies (RoleNameCache). This cache refreshes:

- Any time a Role is updated.
- After the number of minutes specified by the `RoleNameCacheStaleTimeMinutes` application configuration parameter has elapsed.

In the base configuration, `RoleNameCacheStaleTimeMinutes` is set to 60 minutes. The parameter can have a minimum value of 1 and a maximum value of 720.

API roles and localization

If your instance of PolicyCenter uses one or more languages other than English, there are additional behaviors to be aware of.

Internal users and user role queries

When an internal user makes an API call, PolicyCenter queries the database for the user's user roles. Cloud API match a user role to an API role if either (1) the API role has the exact same name as the user role, or (2) the user role has been translated into a non-English language and there is an API role matching one of the translations. Whenever there is a match, the internal user is given the access specified in the API role.

Therefore, if you change the application's default language, to ensure that internal users are granted the correct access, you must also do one of the following:

1. Change the names of any API role files used by internal users. (Guidewire also recommends changing the name of the role within the file itself.) OR
2. Ensure that, for every API role used by internal users, there is a non-English translation for the role name.

For example, suppose there is a user role in PolicyCenter named "Auditor". This user role maps to an API role named "Auditor.role.yaml". The PolicyCenter default language is changed to French. As a result of this change, the query now returns the role name as "Auditeur". To ensure that access to this role is granted appropriately, either the API role file's name must be changed to "Auditeur.role.yaml", or there must be a translation from "Auditor" to "Auditeur".

External users and IdP roles

For external users, the roles associated with each user are stored in either the IdP (and the information is submitted using the JWT) or an additional authorization application (and the information is retrieved by the IExpandTokenPlugin plugin). Regardless of where the roles comes from, each role is prefixed with a "cc.", "pc.", or "bc.". When PolicyCenter receives the call, it converts the JWT to a token map and calls the IExpandTokenPlugin plugin to modify or add tokens based on information from any relevant additional authorization application. Then, it looks for any role names in the token map prefixed with a "cc.", "pc.", or "bc.". It strips off the prefix and then compares the remaining name with the names of the API roles. Whenever there is a match, the external user is given the access specified in the API role.

You can use any language for external roles, even if it is not the default language. But you must ensure that role names match between the IdP/additional authorization application and PolicyCenter.

For example, suppose you wanted to create an external user role for accountants, and you wanted to do this using French. In PolicyCenter, the role could be named "comptable.role.yaml". If roles are stored in the IdP, the IdP would need to assert the appropriate users are associated with "cc.comptable", "pc.comptable", or "bc.compatable". If roles are stored in an additional authorization application, this application would need to respond to the IExpandTokenPlugin plugin's request with "cc.comptable", "pc.comptable", or "bc.compatable".

The prefix for external roles must always be "cc.", "pc.", or "bc.", even if the remainder of the role name uses a different character set, such as Japanese Kanji.

API roles for specific caller types

There are several roles that are designed for specific types of callers:

- All roles whose name is prefixed with "gw_"
- anonymous (used in PolicyCenter only)
- claimautomation_ext (used in ClaimCenter only)
- Unauthenticated

These roles are referenced by internal code or used by other Guidewire services and applications.

WARNING: Do not change the names for the role files in the previous list, regardless of the language you are working in. Doing so will cause Cloud API authorization to not work properly.

chapter 59

Resource access

In order to view and edit information from PolicyCenter, a caller needs to be able to access one or more endpoints. This type of access is known as *endpoint access*. For example, if a caller has access to the GET `/policies` endpoint, that caller can view policies.

However, having access to a given endpoint does not mean a caller can view every resource that endpoint could return. In some cases, callers can access only certain instances of the relevant resource. For example, the GET `/policies` endpoint could be available to a policyholder, an underwriter, and a claims adjuster. But each of these users have access to a different set of policies:

- The policyholder can see only the policies they hold.
- The underwriter can see only the policies assigned to them.
- The claims adjuster can see only the policies associated with claims assigned to them.

This type of access is known as *resource access*. Resource access determines which instances of a given resource are available to a given caller. Resource access is defined by a set of resource access strategies. This topic describes how resource access strategies are assigned to a caller, how they are executed for each call, and how to interpret the base configuration files so that you can understand how resource access is executed.

Overview of resource access strategies

Strategies and IDs

A *resource access strategy* is a set of logic that identifies which resources a caller can access.

A *resource access ID* is a string that identifies either who the caller is or what the caller owns.

Some resource access strategies expect a single resource access ID. Other resource access strategies allow for an array of resource access IDs.

For each call, resource access is determined by executing the resource access strategy using the resource access ID as input. For example, suppose a given resource access strategy states "the caller can access information related to accounts they own". And suppose, for a given call, the resource access ID is account number 464778619. This would mean the following:

- The caller can access resources that are related to account 464778619.
- The caller cannot access resources that are related to accounts other than 464778619.

The list of resource access strategies

The base configuration includes the following resource access strategies:

Strategy name	Persona using this strategy	Resource access ID is...	Grants access to...	More information
pc_accountNumbers	Account holders (including anonymous users who have created an account)	An account number	Resources associated with the account, including its jobs and policies	
pc_producerCodes	Producers	A set of one or more producer codes and roles.	Resources associated with the accounts, jobs, and policies for the given producer and the role.	"The producerCodes resource access strategy" on page 387
pc_username	Internal users	A PolicyCenter user name	Resources this internal user could see in PolicyCenter based on their associated Access Control Lists (ACLs).	
pc.service	Trusted service-to-service application	Not applicable	All resources	"The service resource access strategy" on page 387
default	Callers who have been authenticated but specify no resource access strategy with the call	Not applicable	Typically just metadata resources only (such as API definitions)	
unauthenticated	Callers who have not been authenticated	Not applicable	API definition metadata and the endpoints to create accounts. (The account endpoints are used by anonymous users who may want to quote and potentially bind a policy.)	

The JWT identifies which resource access strategy to use by listing the strategy name in the `scp` token claim. If the given strategy requires resource access IDs, then the JWT also contains a token claim whose name is the strategy name and whose contents are the resource access IDs.

For example, suppose that a given call is using the `pc_accountNumbers` resource access strategy with a resource access ID of 464778619. The JWT would include the following.

```
"scp": [
  "pc_accountNumbers"
],
"pc_accountNumbers": [
  "464778619"
]
```

Determining a call's resource access strategy

Resource access strategies are assigned by internal code as described in the following table. For calls made by services with user context, two resource access strategies are assigned, one at the service level and one at the user level. For all other types of calls, only one resource strategy is assigned.

Strategy name	This is assigned to a call when...
pc_accountNumbers	Any of the following are true: • The JWT's `scp` token claim contains `pc_accountNumbers`, or • The call includes a user context header, and the header includes a `pc_accountNumbers` token claim.
pc_producerCodes	Any of the following are true: • The JWT's `scp` token claim contains `pc_producerCodes`, or • The call includes a user context header, and the header includes a `pc_producerCodes` token claim.
pc_username	Any of the following are true: • The call is using basic authentication, or

Strategy name	This is assigned to a call when...
	• The JWT's `scp` token claim contains `pc_username`, or • The call includes a user context header, and the header includes a `pc_username` token claim, or • The JWT specifies a client ID that was mapped to a service account.
pc.service	The JWT's `scp` token claim contains `pc.service`.
default	The caller has been authenticated, but the JWT specifies no resource access strategy.
unauthenticated	The caller has not been authenticated.

Functionality of specific resource access strategies

This section describes resource access strategies that have unique functionality not found in other resource access strategies.

The producerCodes resource access strategy

The *producerCodes resource access strategy* accepts an array of one or more producer codes and roles. Each item in the array has the following format: `producerCode|producerRole`. This strategy provides callers with access to accounts, jobs, and policies associated with the listed producer codes and it enforces the permissions available to the listed roles.

For example, suppose a caller made a request with the following JWT.

```
{
    "scp": [
        "pc_producerCodes",
        "groups"
    ],
    "pc_producerCodes": [
        "ProducerABC|Producer",
        "ProducerDEF|Producer"
    ],
    "groups": [
        "gwa.lower.pc.External Producer Code"
    ]
}
```

Based on the `pc_producerCodes` token, the caller would have access to accounts, jobs, and policies associated with producer codes ProducerABC and ProducerDEF. The caller would also have permission to perform tasks available to those with the Producer role.

The service resource access strategy

Most of the resource access strategies specify restrictions, which limit the resource and fields that a caller can view.

However, the pc.service resource access strategy specifies almost no restrictions. This is because this resource access strategy is designed to be used by services. Services are expected to be configured such that they access only the resources appropriate for the circumstance. Consequently, JWTs for API calls from services do not typically include resource access IDs.

Note that resource access for the different service-related auth flows behave as described here:

- For **standalone services**, calls use the `pc.service` resource access strategy. Therefore, they have unrestricted resource access.
- For **services with user context**, each call's resource access is the intersection of the service-level resource access and the user-level resource access. The service-level resource access is the `pc.service` resource access strategy, which has no restrictions. Therefore, logically speaking, a service-with-user-context call has resource access equivalent to the user-level resource access.
- For **services with service account mapping**, the service is mapped to an internal service account. The `pc.service` resource access strategy is not used. Rather, the call uses the `pc_username` resource access strategy.

Resource access strategy files

Resource access strategies are defined in a set of resource access files. Each file is a YAML files whose name ends in `access.yaml`. For example, the `internal_core-1.0.access.yaml` file defines the base configuration resource access strategy for internal users. In Studio, access files are located in subdirectories of the `Integration/authorization` directory.

Types of access files

Broadly speaking, there are two types of access files: core and extension.

A *core access file* is an access file that defines one or more base configuration resource access strategies. Core access files either have core in the name, or are located in a package with core in the path.

WARNING: If you need to configure the base configuration resource access behavior, Guidewire recommends that you consult the Guidewire Professional Services team before attempting to modify any extension access files.

An *extension access file* is an access file that provides a location for extensions to base configuration resource access strategy behavior. Extension access files either have `ext` in the name, or are located in a package with `ext` in the path.

Note: If you need to configure the base configuration resource access behavior, Guidewire recommends that you consult your Guidewire account manager before attempting to modify any extension access files.

A strategy is defined across multiple files

Every resource access strategy is defined in multiple files. In the base configuration, all of the files for a given resource access strategy start with the same name.

For example, the `pc_accountNumbers` resource access strategy is defined in the following files:

- The `gw.core.pc.shared.v1` package's `accountholder_core-1.0.access.yaml`
- The `ext.shared.v1` package's `accountholder_ext-1.0.access.yaml` (which references `gw.core.pc.shared.v1.accountholder_core-1.0`)

Sections of a resource access file

Every resource access file contains a list of resources. Resources can be named in the singular (such as `Activity`) or in the plural (such as `Activities`).

When a resource is named in the **singular**, the information that follows applies to endpoints that return individual elements of that resource type. This includes endpoints whose operations are GET (for a single element), POST (for custom business actions), PATCH, and DELETE. For individual elements, there can be only one sections: `permissions`.

When a resource is named in the **plural**, the information that follows applies to endpoints that return collections of that resource type. This includes endpoints whose operations are GET (for a collection) and POST. For collections, there can be two sections: `permissions` and `filters`.

There are different types of access information you can specify for a resource type:

- `permissions`
  - Defines actions that the caller can take on accessible resources, such as `view` and edit
  - Can be specified for both element and collection resources
- `filters`
  - Defines criteria that the resource must meet to be accessible to the caller
  - Can be specified for collection resources only

Resource access files: permissions

In a resource access file, both individual element resources (such as `Activity`) and collection resources (such as `Activities`) can have a `permissions` section. This section defines the actions associated users can take on accessible resources.

The `permissions` section consists of a list of permissions, each of which is followed by a Boolean expression. The permission is granted if and only if the Boolean expression returns true.

For example, the following code defines the permissions for the `Account` resource (for a single account resource) as declared in the `accountholder_core-1.0access.yaml` file. The `view` permission is granted is either Gosu expression returns true. The `freeze` permission is never granted.

```
Account:
  permissions:
    view:     "gw.rest.core.pc.security.v1.AccountHolderSecurityUtil.
               canAccessAccount(resource.Account) || resource.Account.New"
    freeze: false
    purge: false
    unfreeze: false
```

Permissions for element resources

For individual elements, you can specify `view`, `create`, `edit`, and `delete` permissions. You can also specify permissions for custom business actions. For example, if there is a POST `/activities/{activityId}/assign` endpoint, then for an `Activity` resource, you can specify an `assign` permission. For custom actions, the permission name must match the verb used at the end of the endpoint path.

For example, the following specifies permissions for the `Job` entity. Note that is specifies standard `view` and `edit` permissions as well as a custom business action permission, `quote`.

```
Job:
  permissions:
    view: "gw.rest.core.pc.security.v1.AccountHolderSecurityUtil.canView(resource.Job)"
    edit: "gw.rest.core.pc.security.v1.AccountHolderSecurityUtil.canEdit(resource.Job)"
    ...
    quote: "gw.rest.core.pc.security.v1.AccountHolderSecurityUtil.canQuote(resource.Job)"
    ...
```

WARNING: If you create a new endpoint that executes a custom action, do not name the endpoint with a name that would conflict with a base permission, such as `view`, `edit`, `create`, or `delete`. Doing so will result in unexpected permission behaviors.

If a given permission is not specified in an access file, then the permission defaults to the permission of the resource's parent. If a given resource does not have a permissions section, then all permissions default to the permission of the resource's parent.

Possible Boolean expressions

Any Gosu expression that returns true or false can be used as a permission's Boolean expression.

For permissions, the base configuration includes the following types of Boolean expressions:

- A Boolean value
- The keyword `__inherit` (in which case the permission is inherited from the resource's parent, such as `AccountActivities...view: __inherit`)
- A Gosu expression, including:
  - A Gosu system perm expressions (such as `"perm.system.actview"`)
  - A Gosu resource perm expressions (such as `"perm.Activity.view(resource.Activity)"`)
  - A Gosu expression (such as `"!resource.Note.Confidential || resource.Note.Author == entity.User.util.CurrentUser || perm.Claim.viewconfidentialnotes(resource.Note.Claim)"`)

- A Gosu method declared at the system API layer (such as `"gw.rest.core.pl.util.v1.ActivityInternalPermissionUtil.canApprove(resource.Activity)"`)

For more information on writing Gosu expressions that check for system permissions or resource permissions, refer to the *Rules Guide*.

In some cases, multiple expressions are listed on several lines, such as the following example. In this case, the expressions are ANDed together. All expressions must return true for the permission to be granted.

```
Account:
  permissions:
    purge:
    - "perm.Account.edit(resource.Account)"
    - "perm.System.purge"
```

Resource access files: filters

In a resource access file, collection resources (such as `Activities`) can have a `filter` section. This section defines criteria the each member of the collection must meet in order to be accessible. They may be the only criteria that the resource must meet, or they may be in addition to other criteria added by PolicyCenter. (Filters are not used for individual element resources because you can use view and edit permissions to control the availability of an individual resource.)

The `filter` section consists of a list of one or more filter expressions. For a resource to be accessible, all access filter expressions must return true.

The `filter` section consists of a list of one or more filter expressions. A filter expression can be:

- A Gosu expression that returns filter criteria, such as `gw.rest.core.pl.common.v1.activities.AssignedActivitiesFilter`.
- The keyword `__nofilter`, which indicates that there is no filter and all resources are accessible.

For example, the following code defines the filters for the `Accounts` resource (for a collection of accounts) as declared in the `accountholder_core-1.0.access.yaml` file:

```
Accounts:
  filter: gw.rest.core.pc.account.v1.security.AccountHolderAccountsFilter
```

chapter 60

Proxy user access

A *proxy user* is an internal user account in the PolicyCenter database that is assigned to certain types of Cloud API calls made by external users or services. If the call records information or executes a permissions check that requires an internal user account, the proxy user account is used. This type of access is referred to as *proxy user access*.

Proxy user access is defined by the `RestAuthenticationSourceCreatorPlugin` plugin and a set of proxy users. This topic describes how to work with proxy users.

> **Note:** Proxy users do not apply to internal users (using either basic authentication or bearer token authentication). Proxy users are relevant only for external users, anonymous users, standalone services, services with external user context, and unauthenticated callers.

Proxy users

When a caller makes a Cloud API call, Cloud API checks to see if the caller has sufficient endpoint access and resource access. If they do, Cloud API hands processing over to the appropriate internal PolicyCenter logic.

The internal PolicyCenter logic may trigger code that can only be completed using a user account from the `pc_user` table. For example:

- The call may create or modify data. When this occurs, PolicyCenter records the name of the `CreateUser` or `UpdateUser`.
- The call may trigger a domain-level permission check.
  - For example, the call may attempt to assign an activity to the caller. To do this, PolicyCenter must verify that the caller has sufficient permission to own an activity.)
- The call may trigger an authority limit check.
  - For example, the call may attempt to create a collision coverage with a deductible less than $1000. PolicyCenter must check to see if the amount of the coverage term is within the caller's authority limit.

When the caller is an internal user, PolicyCenter uses the internal user account for these types of code.

- The internal user is recorded as the `CreateUser` or `UpdateUser`.
- The internal user's user roles are checked for domain-level permissions as needed.
- The internal user's authority limit profiles are checked for authority limit checks as needed.

However, external users and services are not listed in the `pc_user` table. They cannot be recorded as a `CreateUser` or `UpdateUser`. They also have no system permissions or authority limits assigned to them. So, when a call is made by someone who is not an internal user, PolicyCenter assigns a *proxy user* to the call.

- If the call creates or modifies data, the proxy user is listed as the `CreateUser` or `UpdateUser`.
- If the call triggers a domain-level permissions check, the proxy user's user roles are checked.
- If the call triggers an authority profile check, the proxy user's authority profile limits are checked.

Types of proxy users

Fundamentally, there are three types of proxy users:

- The *external producer proxy user* is a proxy user assigned to calls made by external producer users and services with external producer user context.
- The *external proxy user* is a proxy user assigned to calls made by external users and services with external user context.
- The *service proxy user* is a proxy user assigned to calls made by standalone services.
- The *unauthenticated proxy user* is a proxy user assigned to calls made by unauthenticated callers.

Note that for each type of caller listed above, there is only one proxy user. In other words, all external users and services with external user context make use of a single proxy user, the external proxy user. All standalone services make use of a single proxy user, the service proxy user.

Technically, there is a fourth type of proxy user, the *default proxy user*. This user is used in the unlikely situation that, for some reason, the regular proxy user cannot be assigned to the call.

Proxy users in the base configuration

The base configuration bootstrap data includes the following proxy users. (Bootstrap data is loaded when the product is installed. It is not a part of sample data.)

Proxy user type	Base configuration user	User role	UW authority profile
External producer code proxy user	`extproducercodeuser`	External Producer Code User	External Producer Code User Profile
External proxy user	`extuser`	External User	External User Profile
Service proxy user	`serviceuser`	Service User	Service User Profile
Unauthenticated proxy user	`uauser`	Unauthenticated User	Unauthenticated User Profile
Default proxy user	`defaultuser`	Default User	(none)

To prevent anyone from logging in as one of these users, each of these users is created with a password that makes use of a character that is not valid Base64 encoding.

You can configure the user roles and authority limit profiles referenced by these users. But, Guidewire recommends that you do not configure the users themselves.

WARNING: Do not remove the base configuration proxy users from your database. If these proxy users do not exist, authorization will not behave as expected.

Proxy user assignment

Proxy users are assigned by the `RestAuthenticationSourceCreatorPlugin` plugin. The following table details the conditions that determine which proxy user to assign to a call.

Proxy user type	When this user type is assigned
External producer code proxy user	The call includes the `pc_producerCodes` scope
External proxy user	The call includes the `pc_accountNumbers` scope
Service proxy user	The call includes the `pc.service` scope
Unauthenticated proxy user	The call has no authentication header
Default proxy user	The call requires a proxy user and for some reason no other proxy user could be assigned

WARNING: Do not modify the settings in the `RestAuthenticationSourceCreatorPlugin` plugin that identify the users for each proxy user type. Doing so could make the authorization functionality not behave as expected.

When is proxy user information used?

Proxy users as the "user of record"

Some of the actions that a user can execute in PolicyCenter require the user's name to be recorded in the database. For example:

- When an object (such as an activity or a note) is created, the user who created it is stored in the object's `CreateUser` field.
- When an object (such as an activity or a note) is modified, the user who modified it is stored in the object's `UpdateUser` field.

Actions that require a "user of record" can be triggered by Cloud API calls. When the call is triggered by an internal user, the internal user is noted as the user of record. When the call is triggered by an external user or service, the proxy user is noted as the user of record.

Proxy users and system permissions

A Cloud API call may trigger a check to see if the caller has a specific system permission. When this occurs, PolicyCenter checks to see if the proxy user has a user role that includes the system permission.

- If the proxy user has the permission, processing continues as normal.
- If the proxy user does not have the permission, the action is prevented.

For more information on roles and permissions, refer to the *Application Guide*.

Proxy users and authority limits

A Cloud API call may trigger a check to see if the caller has sufficient financial authority to execute a given action. When this occurs, PolicyCenter checks to see if the proxy user has an authority limit profile with the corresponding authority limit set to a sufficiently high amount.

- If the proxy user has a sufficiently high authority limit, processing continues as normal.
- If the proxy user does not have a sufficiently high authority limit, processing is suspended. PolicyCenter automatically creates an approval activity and assigns it to the appropriate user. If the activity is approved, processing for the underlying transaction continues.

For more information on authority limits, refer to the *Application Guide*.

Configuring proxy users

Modifying system permissions for proxy users

System permissions control a user's ability to access data or take action within a Guidewire application. But, system permissions do not provide the ability to use Cloud API endpoints. The ability to use Cloud API endpoints and access fields on Cloud API resources is controlled by API roles.

If you need to modify the permissions for a proxy user, there are different possible scenarios:

- The proxy user has all required system permissions, but the calling user does not have access to the required endpoints and/or the required fields on the endpoint resources. In this case, you need to modify the appropriate API role files (such as the `Producer.role.yaml` file).
- The calling user has access to the required endpoints and/or the required fields on the endpoint resources, but their proxy user does not have all required system permissions. In this case, you need to modify the user role associated with the proxy user (such as the "External User" role).

- The calling user lacks access to the required endpoints and/or the required fields on the endpoint resources, and their associated proxy user lacks the required system permissions. In this case, you need to modify both the appropriate API role files (such as the `Producer.role.yaml` file) and the user role associated with the proxy user (such as the "External User" role).

For more information on configuring API roles, see “Endpoint access” on page 373.

For more information on configuring user roles, see the *Application Guide*.

Modifying authority limit profiles for proxy users

Authority limits determine whether a user can create a financial object (such as a claim reserve, a coverage, or a goodwill credit) without requiring approval. If the amount of the object is at or below the user's limit, no approval is needed. If the amount is above the user's limit, approval is required.

Unlike permissions, authority limits are controlled exclusively within the authority limit profiles in the Guidewire application. There is no Cloud API analog to authority limit profiles. If you need to modify the authority limits for a proxy user, you must modify the authority limit associated with that proxy user.

For more information on configuring authority limit profiles, see the *Application Guide*.

Do not modify which users are the proxy users

The base configuration provides four proxy users: `extproducercodeuser`, `extuser`, `serviceuser`, `uauser`, and `defaultuser`. Guidewire recommends against modifying these users or the references to these users in the `RestAuthenticationSourceCreatorPlugin`. Doing so could make the authorization functionality not behave as expected.

chapter 61

Configuring the IExpandTokenPlugin plugin

In bearer token authentication, the caller presents a *JSON Web Token (JWT)*. The JWT contains a set of claims. Each claim is a key/value pair that represents information that "the bearer of the token claims to be true". For example, a JWT could contain the following claim, which asserts the identity of the bearer of the token (in the `sub` claim, which identifies the "subject"):

```
[
  "sub": "rnewton@email.com",
 ...
]
```

Cloud API uses information in the JWT to determine the authorization to grant to the caller. This typically involves two types of information:

- Some information in the JWT identifies the API roles to assign to the caller. This determines the level of endpoint access the caller has.
- Some information in the JWT identifies the caller's resource access IDs. This determines the level of resource access the caller has. (In other words, this determines which specific resources that caller can access.)

For example, suppose Ray Newton is an insured making a request to PolicyCenter. The JWT includes the following.

```
[
  "sub": "rnewton@email.com",
  "groups": [
    "gwa.prod.pc.Account_Holder"
  ],
  "pc_accountNumbers": [
    "C000143542"
  ],
 ...
]
```

Cloud API grants authorization in this way:

- Based on the `groups` claim, the caller is given endpoint access as defined in the `Account_Holder` API role.
- Based on the `pc_accountNumbers` claim, the caller is given resource access to resources associated with any account whose account number is C000143542.

The structure and contents of a JWT vary based on the type of caller. But, for every type of caller using bearer token authentication, the JWT either contains the caller's API roles and resource access IDs, or it contains information used to determine the caller's API roles and resource access IDs.

The authorization information that is placed into a JWT can come from the IdP. It can also come from the caller application itself, provided it is a scope value that the caller application has already registered with Guidewire Hub. Cloud API also supports the ability to add PolicyCenter-specific authorization information after the JWT has been received but before authorization is determined. This is done by the IExpandTokenPlugin plugin.

For more overview information on JWTs and how the IExpandTokenPlugin plugin interacts with the information flow, see “Constructing JWTs” on page 275.

Implementing the IExpandTokenPlugin plugin

To implement the IExpandTokenPlugin plugin, you must do the following:

1. Create an implementation class
2. Register the plugin

Creating an IExpandTokenPlugin implementation class

The class that implements the IExpandTokenPlugin plugin must meet the following requirements:

- It must implement `gw.api.modules.rest.framework.v1.plugin.security.IExpandTokenPlugin`
- It must specify an override of the `getTokenExpansions` method

The getTokenExpansions method

When a Cloud API call using bearer token authentication is received, PolicyCenter extracts the information from the JWT into a "token map". Then, it calls the `getTokenExpansions` method. This method returns a "token expansions map" which contains the key/value pairs to add to the initial token map, or to override in the initial token map.

- If the token expansions map specifies a claim that is not in the original token map, Cloud API adds it to the original token map.
- If the token expansions map specifies a claim that is already in the original token map, Cloud API replaces the original claim with the expansions claim. Thus, if a given claim must have both values from the JWT and values retrieved from the external system, this method must merge the two sets into a single claim.

The default implementation of the method, as provided by Studio, is as follows.

```
override function getTokenExpansions(map : Map<String, Object>) : Map<String, Object> {
 return map
}
```

The method receives the original token map in the `map` object, which contains all the key/value pairs from the JWT as received from the caller.

The method must do the following:

1. Extract any appropriate lookup values from the original token map
2. Call the appropriate system of record to retrieve the relevant additional authorization values
3. Construct a "token expansions" map return value that specifies all claims to add to the original token map, and all claims already in the original token map that must be replaced

The method cannot return null. If there are no required changes, return an empty map.

Formats for values in the token expansions map

For internal users, there is no need to specify API roles in either the JWT or the token expansions map. API roles are retrieved by querying the database for the user roles assigned to the user. For more information, see “OAuth2 authorization code flow: Internal users” on page 297.

For external users, API roles must be specified in either the JWT or the token expansions map. When they are specified in the expansions map:

- The key must be set to "groups".
- The value must be of type List<String>, where each string has the following format:
  - It starts with "gwa.<*planetclass*>.pc.", where <*planetclass*> is set to either "prod", "preprod", or "lower".
  - It ends with the Cloud API role name.

For example, to assign a user to an API role named "Account_Holders" for a production planet, the value added to the expansion map must be:

```
"groups": [
  "gwa.prod.pc.Account_Holders"
]
```

Resource access IDs

For both internal and external users, resource access IDs must be specified in either the JWT or the token expansions map. When they are specified in the expansions map:

- The key must be set to the name of the resource access strategy. This must be one of the following:
  - pc_username
  - pc_accountNumbers
- If the resource access strategy requires a single resource access ID, the key's value must be set to that ID.
- If the resource access strategy permits one or more resource access IDs, the key's value must be set to an array storing the IDs.

For example, the pc_username resource access strategy requires a single resource access ID set to the user's username. If the caller is "aapplegate@acme.com", the value added to the expansion map must be:

```
"pc_username": "aapplegate@acme.com"
```

The pc_accountNumbers resource access strategy permits one or more resource access IDs set to the caller's policy numbers. If the caller owns accounts C000123 and C000456, the value added to the expansion map must be:

```
"pc_accountNumbers": [
  "C000123",
  "C000456"
]
```

Performance considerations

This plugin is called during Cloud API authentication. Guidewire recommends that insurers confirm that the SLA with their authorization system of record is sufficient for such a heavily used case.

Guidewire recommends against adding more than 1000 elements in a List or Array that is the value for a key. Such a large number of entries can result in performance issues when GenericCenter passes queries to the database that contain filtering logic with more than 1000 values.

The getTokenExpansions method example: producer codes

For callers using the producerCodes access strategy, suppose you store producer codes in an external producer system. In order to identify the correct producer codes to grant to a caller, the external system must have uniquely identifying information about the caller, such as the value in the caller's `sub` claim.

The method receives the following token map:

```
"sub": "kegerston@allrisk.com",
"scp": [
  "cc_producerCodes",
  "tenant.acme",
  "project.default",
  "planet_class.prod"
],
"groups": [
  "gwa.prod.cc.Producer"
],
```

The method then does the following:

1. Extracts the lookup value (kegerston@allrisk.com)
2. Sends this value to the external producer system
3. Puts the returned values in an expansion tokens map with a `cc_producerCodes` claim.

If the external producer system returned the values 100-002541, 100-002542, and 100-002543, the method's expansion map would be:

```
"cc_producerCodes": [
  "100-002541",
  "100-002542",
  "100-002543"
]
```

After the plugin has been called, the original token map would look like this:

```
"sub": "kegerston@allrisk.com",
"scp": [
  "cc_producerCodes",
  "tenant.acme",
  "project.default",
  "planet_class.prod"
],
"groups": [
  "gwa.prod.cc.Producer"
],
"cc_producerCodes": [
  "100-002541",
  "100-002542",
  "100-002543"
]
```

Example implementation

The following is an example of a complete implementation class for the IExpandtokenPlugin plugin that adds the producer codes discussed in the previous example.

```
package gw.acme.rest.internal.pc.dgc.plugin

uses gw.api.modules.rest.framework.v1.plugin.security.IExpandTokenPlugin

uses javax.annotation.Nonnull

class ExpandTokenDGCPlugin implements IExpandTokenPlugin {

  @Nonnull
  override function getTokenExpansions(map : Map<String, Object>) : Map<String, Object> {

    //Map to be merged to original one
    var dgcMap = new HashMap<String, Object>()
    dgcMap.put("groups", {"gwa.dev.cc.Producer"})
    dgcMap.put("cc_producerCodes", {"100-002541"})
    dgcMap.put("cc_producerCodes", {"100-002542"})
    dgcMap.put("cc_producerCodes", {"100-002543"})

    return dgcMap
  }
}
```

Register the IExpandTokenPlugin plugin

About this task

The base configuration does not include a registry entry for the IExpandTokenPlugin plugin. You must add it manually. For more information on registering plugins, see the *Configuration Guide*.

Procedure

1. In Studio, navigate to **config** > **plugins** > **registry**.
2. Right-click the registry node and select **New** > **plugin**.

3. In the **Plugin** dialog box, provide the following values:
   - Name: IExpandTokenPlugin
   - Interface: IExpandTokenPlugin
4. Click **OK**. Studio opens a new **IExpandTokenPlugin.gwp** tab.
5. In the upper left corner of the tab, click the plus symbol (+). Then select either **Add Gosu Plugin** (to register the plugin with a Gosu implementation class) or **Add Java Plugin** (to register the plugin with a Java implementation class).
6. On the right side, add a value to the **Gosu Class** field or the **Java Class** field that identifies the class that implements the plugin.
7. To deploy your changes, restart the server.

chapter 62

Security levels

API roles specify the resources that callers can access, and the properties on those resources that callers can view or edit. In an API role file, you can explicitly list each property and its view and edit access. However, there may be some situations where it is easier to grant access to a set of properties without explicitly naming all of them. In these situations, you can use security levels.

Security level is a property-level attribute that can be used by API roles to grant view or edit permissions to a set of properties. API roles can grant view or edit permissions using the `"*level"` expression, which means "grant the permission to all properties on this resource with the security level of *level*."

There are three security levels: internal, sensitive, and public. These levels are not hierarchical. Granting access to a specific security level does not inherently include any other security levels. Also, there is no inherent meaning tied to them. They are arbitrary labels that you can use in whatever way is most appropriate.

Specifying a field's security level

Security levels are set in the resource's schema file. The syntax is `"securityLevel": level`.

- `level` must be either `internal`, `sensitive`, or `public`.
- If a property has no specified security level, it defaults to `public`.

For example, the following code snippet, from `admin_pl-1.0.schema.json`, declares two properties for the `User` resource:

```
"User": {
  "properties": {
    "firstName": {
        ...
    },
    "homePhone": {
        ...
        "securityLevel": "internal"
      }
    },
```

The `homePhone` property has a security level of `internal`. The `firstName` property has no defined security level, and therefore defaults to a security level of `public`.

Using security levels in API roles

In an API role file, when specifying view and edit permissions for a given resource, you can use the expression `"*level"` to indicate "all properties on the resource that have the specified level".

For example, the following grants access to fields on the `User` resource. The grantee can edit and view all properties on the `User` resource that have a security level of `public` as well as the `workPhone` property (which presumably is not a

public field). (Based on the previous code snippet, the grantee would be able to view and edit `firstName` and `workPhone`, but not `homePhone`.)

```
accessibleFields:
  User:
    edit:
    - "*public"
    - workPhone
    view:
    - "*public"
    - workPhone
```

Additional behaviors of security levels

Security levels are not hierarchical. To grant access to multiple levels, all levels must be listed explicitly.

For example, the following grants edit access to all properties on the `User` resource that are public or sensitive:

```
User:
  edit:
  - ["*public", "*sensitive"]
```

If the view or edit section of a resource lists both explicit properties and a "*`*Level*`*" expression, the grantee has access to all explicitly listed properties and all properties with the given security level.

For example, the following grants edit access to all properties on the `User` resource that are public as well as the `workPhone` property:

```
User:
  edit:
  - "*public"
  - workPhone
```

chapter 63

Configuring the reauthorize anonymous user flow

An anonymous user can create an account and start a submission, but not bind the submission within the same session. This could happen because:

- The original session expired (the user left the session with the intent to finish the work later).
- The user switched to a different device.

When an anonymous user wants to return and complete a submission, they must first find the incomplete submission, and then obtain a new self-signed JWT to authenticate with PolicyCenter. The POST `/job/v1/recover-new-jobs` endpoint is designed for this use case.

- The request body contains search criteria that Cloud API can use to identify incomplete submissions associated with the anonymous account.
- The response contains the incomplete submissions matching those criteria along with a new self-signed JWT.

With the response, the third-party application can identify which submission the anonymous user wishes to complete, and it can use the self-signed JWT to start a new session.

Base configuration behavior

Submission search criteria varies for each line of business and typically depends on custom aspects of each line of business. Therefore, in the base configuration, the `/recover-new-jobs` endpoint always returns 0 results. The insurer must configure Cloud API to identify the valid search criteria properties and how to construct a query using those criteria. For more information, see “Implementing the /recover-new-jobs endpoint” on page 404.

Once the endpoint has been configured, it has the following additional behaviors:

- A job cannot be recovered if it is more than 7 days old.
- The endpoint can return a maximum of 5 jobs.

These threshold values can also be configured. For more information, see “Configuring general new job recovery behavior” on page 410.

Configured and implicit criteria

The `/recover-new-jobs` endpoint has two levels of criteria: configured and implicit. The *configured criteria* consist of the criteria added by the insurer during implementation. For example, an insurer could configure the endpoint so that searches by first name and last name are supported.

The *implicit criteria* consist of criteria that are automatically applied to all `/recover-new-jobs` searches. This criteria cannot be configured. The implicit criteria are:

- The job type must be Submission.
- The job status must be Draft or Quoted.
- The account cannot have any bound jobs associated with it.
- The criteria cannot match submissions from multiple accounts. (If it does, the endpoint returns 0 results.)

Returning zero results

There are several circumstances under which the `/recover-new-jobs` endpoint returns 0 results.

If there are no jobs matching the specified criteria, the endpoint returns 0 results. This is because no matching job could be found.

If there are one or more submissions matching the specified criteria, but these submissions belong to different accounts, the endpoint returns 0 results. This is because the endpoint is unable to identify which account and which submissions are associated with the caller.

If there are one or more submissions matching the specified criteria, but the account has at least one bound submission or one other non-submission job (such as a renewal), the endpoint returns 0 results. This is because an account with a bound submission or a non-submission job is related to an external user, not an anonymous user. The caller must use the external user auth flow to execute additional calls.

Whenever the `/recover-new-jobs` endpoint returns 0 results, the response header does not contain any JWT.

Risk assessment

By design, the `/recover-new-jobs` endpoint returns account and job information to an anonymous user. For any endpoint of this nature, there is a risk that personal information could be returned to a caller who is not authorized to access that information.

Guidewire recommends that insurers execute a sufficiently rigorous amount of testing and evaluation to ensure that the search criteria and query logic that they configure will not result in returning unintended personal information to an unauthorized caller.

Implementing the /recover-new-jobs endpoint

To implement the `/recover-new-jobs` endpoint, you must do the following:

1. Define search criteria properties in the `RecoverNewJobsWrapperExt` class
2. Define query logic in the `RecoverNewJobsExtResource` class
3. Extend the `RecoverNewJobsRequestAttributes` schema

Sample implementation examples

The following topics contains sample code for a complete implementation for the reauthorize anonymous user flow. In this sample:

- The caller application submits one or more of the following values:
  - Account number
  - Job number
  - First name (of the primary insured)
  - Last name (of the primary insured)
  - Postal code (of the primary insured)
- The endpoint returns any jobs that:

- Meet the implicit criteria for new job recovery (such as the job is an unbound submission)
- Has an exact match with all values submitted

Note that, as this is a sample implementation, none of the properties are required. This includes First Name and Last Name, each of which can be specified without specifying the other.

WARNING: This implementation has not gone through any risk assessment. Guidewire does not recommend using this example in a production environment, as it may expose secure data and it may not have acceptable performance. Guidewire recommends that insurers implement their own search logic and that they ensure their implementation is secure and has sufficient performance.

Define search criteria properties

First, you must define the properties that store the search criteria.

The base configuration includes a class where search criteria properties are to be defined. It is called `RecoverNewJobsWrapperExt`, and it is declared in the `gw.rest.ext.pc.job.v1.security` package. The base configuration version is an empty class that looks similar to this:

```
package gw.rest.ext.pc.job.v1.security

uses gw.rest.core.pc.job.v1.security.RecoverNewJobsWrapper

@Export
class RecoverNewJobsWrapperExt extends RecoverNewJobsWrapper {
}
```

To enable new job recovery, define the required search criteria in this class.

Sample RecoverNewJobsWrapperExt class

The sample implementation supports new job recovery search on the following values:

- Account number
- Job number
- First name (of the primary insured)
- Last name (of the primary insured)
- Postal code (of the primary insured)

This is what the class looks like after configuration.

```
package gw.rest.ext.pc.job.v1.security

uses gw.rest.core.pc.job.v1.security.RecoverNewJobsWrapper

/*
 This class has recovery extension properties used for the Recover New Jobs endpoint. These recovery properties are
examples only and have not gone through Risk Acceptance.
 */

@Export
class RecoverNewJobsWrapperExt extends RecoverNewJobsWrapper {
  var _accountNumber : String as AccountNumber
  var _firstName : String as FirstName
  var _jobNumber : String as JobNumber
  var _lastName : String as LastName
  var _postalCode : String as PostalCode
}
```

Define the query logic

Next, you must define the query that executes the search.

The base configuration includes a class where the query is to be defined. It is called `RecoverNewJobsExtResource`, and it is declared in the `gw.rest.ext.pc.job.v1.security` package.

The class has a single getter named `RecoverNewJobsWrapper`. In the base configuration, it simply returns a new `RecoverNewJobsWrapperExt` instance. The base configuration version looks similar to this:

```
package gw.rest.ext.pc.job.v1.security

uses gw.rest.core.pc.job.v1.security.RecoverNewJobsCoreResource
uses gw.rest.core.pc.job.v1.security.RecoverNewJobsWrapper

@Export
class RecoverNewJobsExtResource extends RecoverNewJobsCoreResource {

  protected override property get RecoverNewJobsWrapper() : RecoverNewJobsWrapper {
    return new RecoverNewJobsWrapperExt()
  }

}
```

To enable new job recovery, add a method override to the class that overrides the `populateRecoverNewJobsQuery` method. The new method must return a PolicyPeriod query that specifies whatever query restrictions are required based on the search criteria properties. The following is a high-level syntax statement for the method.

```
  protected override function populateRecoverNewJobsQuery(recoverNewJobsWrapper :
RecoverNewJobsWrapper) : IQueryBeanResult<PolicyPeriod> {

    // code that defines query with appropriate search criteria

    return <some_value_of_type_IQueryBeanResult<PolicyPeriod>_>
  }
```

For more information on writing Gosu queries, see the *Gosu Reference Guide*. You can also refer to the `makeQueryBuilderForNormalSearches` method in the `PolicyPeriodSearchCriteria.gs` for more example properties.

Sample RecoverNewJobsExtResource class

The sample implementation supports new job recovery search on the following values:

- Account number
- Job number
- First name (of the primary insured)
- Last name (of the primary insured)
- Postal code (of the primary insured)

Thus, the `RecoverNewJobsWrapper` getter does the following for each search criteria property:

- Checks to see if a value was provided for the property
- Adds the corresponding query logic to the query, if a value was provided

This is what the class looks like after configuration. Comments in the code appear in bold.

```
package gw.rest.ext.pc.job.v1.security

uses gw.account.AccountQueryBuilder
uses gw.api.database.IQueryBeanResult
uses gw.contact.ContactQueryBuilder
uses gw.contact.PolicyContactRoleQueryBuilder
uses gw.job.JobQueryBuilder
uses gw.policy.PolicyPeriodQueryBuilder
uses gw.policy.PolicyQueryBuilder
uses gw.rest.core.pc.job.v1.security.RecoverNewJobsCoreResource
uses gw.rest.core.pc.job.v1.security.RecoverNewJobsWrapper

@Export
class RecoverNewJobsExtResource extends RecoverNewJobsCoreResource {

  protected override property get RecoverNewJobsWrapper() : RecoverNewJobsWrapper {
    return new RecoverNewJobsWrapperExt()
  }

// Start of the populateRecoverNewJobsQuery override
  protected override function populateRecoverNewJobsQuery(recoverNewJobsWrapper : RecoverNewJobsWrapper) :
IQueryBeanResult<PolicyPeriod> {

// Create a new PolicyPeriod query.
    var queryBuilder = new PolicyPeriodQueryBuilder()
```

```
// If the job number was specified, add it to the criteria
    if ((recoverNewJobsWrapper as RecoverNewJobsWrapperExt).JobNumber != null) {
      var jobQueryBuilder = new JobQueryBuilder()
          .withJobNumber((recoverNewJobsWrapper as RecoverNewJobsWrapperExt).JobNumber)
      queryBuilder.withJob(jobQueryBuilder)
      }

// If the account number was specified, add it to the criteria
    if ((recoverNewJobsWrapper as RecoverNewJobsWrapperExt).AccountNumber != null) {
      var policyQueryBuilder = new PolicyQueryBuilder()
          .withAccount(new AccountQueryBuilder().withAccountNumber((recoverNewJobsWrapper as
RecoverNewJobsWrapperExt).AccountNumber))
      queryBuilder.withPolicy(policyQueryBuilder)
    }

    if ((recoverNewJobsWrapper as RecoverNewJobsWrapperExt).AccountNumber.NotBlank) {
      queryBuilder.withUseAnyArrayForPolicyContactRoleSearch(false)
    }

// Add First Name, Last Name, and Postal Code criteria
    var contactQueryBuilder = new ContactQueryBuilder()
        .withFirstName((recoverNewJobsWrapper as RecoverNewJobsWrapperExt).FirstName)
        .withLastName((recoverNewJobsWrapper as RecoverNewJobsWrapperExt).LastName)
        .withPostalCodeDenorm((recoverNewJobsWrapper as RecoverNewJobsWrapperExt).PostalCode)

// Limit the query to match First Name, Last Name, and Postal Code only
for contacts whose role on the policy the primary insured
    var policyContactRoleQueryBuilder = new PolicyContactRoleQueryBuilder()
        .withSubtype(TC_POLICYPRINAMEDINSURED)
        .withContactDenorm(contactQueryBuilder)
    queryBuilder.withPolicyContactRole(policyContactRoleQueryBuilder)

// Return the query
    return queryBuilder.build().select() as IQueryBeanResult<PolicyPeriod>
  }
}
```

Extend the RecoverNewJobsRequestAttributes schema

Finally, you must extend the `RecoverNewJobsRequestAttributes` schema.

The schema file

To extend the schema itself, add your extensions to the `policyperiod_ext-1.0.schema.json` file.

Guidewire recommends adding an `_Ext` suffix to all property names you add to a base configuration schema. This is to prevent any conflicts that could arise in future releases with search criteria properties added by Guidewire.

Sample policyperiod_ext-1.0.schema.json file

For the sample implementation, this is what the file looks like after configuration.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "x-gw-combine": [
    "gw.content.pc.policyperiod.v1.policyperiod_content-1.0",
    "ext.common.v1.common_ext-1.0"
  ],
  "definitions": {
    "RecoverNewJobsRequestAttributes": {
      "title": "Recover new jobs request attributes",
      "description": "Recovery properties used to recover new non-complete jobs for an unauthenticated user",
      "type": "object",
      "x-gw-sinceVersion": "1.6.0",
      "properties": {
        "accountNumber_Ext": {
          "title": "Account number",
          "description": "The `accountNumber` of the account",
          "type": "string"
        },
        "firstName_Ext": {
          "title": "First name",
          "description": "The `firstName` of the account's `accountHolder`",
          "type": "string"
        },
        "jobNumber_Ext": {
          "title": "Job number",
          "description": "The number of the job",
          "type": "string"
        },
        "lastName_Ext": {
          "title": "Last name",
```

```
          "description": "The `lastName` of the account's `accountHolder`",
          "type": "string"
        },
        "postalCode_Ext": {
          "title": "Postal code",
          "description": "The `postalCode` of the `primaryAddress` on the account's `accountHolder`. Only applicable in
certain countries.",
          "type": "string",
          "x-gw-extensions": {
            "countryRestricted": true
          }
        }
      }
    }
  }
}
```

The mapper file

To extend the mappings, add your extensions to the `policyperiod_ext-1.0.mapping.json` file.

This file declares search attribute properties, and these properties never store data from PolicyCenter. Therefore, there is no technical requirement to have mappers. However, Cloud API raises a warning if properties are added to a schema without mappers. So, Guidewire recommends adding these mappers solely to suppress this warning.

Sample policyperiod_ext-1.0.mapping.json file

For the sample implementation, this is what the file looks like after configuration.

```
{
  "schemaName": "ext.policyperiod.v1.policyperiod_ext-1.0",
  "combine": [
    "gw.content.pc.policyperiod.v1.policyperiod_content-1.0",
    "ext.common.v1.common_ext-1.0"
  ],
  "mappers": {
    // Added to suppress "no mappers" warnings
    "RecoverNewJobsRequestAttributes": {
      "schemaDefinition": "RecoverNewJobsRequestAttributes",
      "root": "java.lang.Object",
      "properties": {
        "accountNumber_Ext": {
          "path": "null as String"
        },
        "firstName_Ext": {
          "path": "null as String"
        },
        "jobNumber_Ext": {
          "path": "null as String"
        },
        "lastName_Ext": {
          "path": "null as String"
        },
        "postalCode_Ext": {
          "path": "null as String"
        }
      }
    }
  }
}
```

The updater file

To extend the updaters, add your extensions to the `policyperiod_ext-1.0.updater.json` file.

Sample policyperiod_ext-1.0.updater.json file

For the sample implementation, this is what the file looks like after configuration. Note that, for each property, the root is the `RecoverNewJobsWrapperExt` class.

```
{
  "schemaName": "ext.policyperiod.v1.policyperiod_ext-1.0",
  "combine": [
    "gw.content.pc.policyperiod.v1.policyperiod_content-1.0",
    "ext.common.v1.common_ext-1.0"
  ],
  "updaters": {
    "RecoverNewJobsRequestAttributes": {
      "schemaDefinition": "RecoverNewJobsRequestAttributes",
```

```
      "root": "gw.rest.ext.pc.job.v1.security.RecoverNewJobsWrapperExt",
      "properties": {
        "accountNumber_Ext": {
          "path": "RecoverNewJobsWrapperExt.AccountNumber"
        },
        "firstName_Ext": {
          "path": "RecoverNewJobsWrapperExt.FirstName"
        },
        "jobNumber_Ext": {
          "path": "RecoverNewJobsWrapperExt.JobNumber"
        },
        "lastName_Ext": {
          "path": "RecoverNewJobsWrapperExt.LastName"
        },
        "postalCode_Ext": {
          "path": "RecoverNewJobsWrapperExt.PostalCode"
        }
      }
    }
  }
}
```

Calling the /recover-new-jobs endpoint

Once you have implemented the `/recover-new-jobs` endpoint, you can use it in the same way that you would use a base configuration search endpoint. The request includes a body with the appropriate search criteria. The response includes a collection of jobs meeting the implicit and configured search criteria. If the response includes one or more jobs, the response header also contains a self-signed JWT that can be used for subsequent calls.

For example, suppose you configure the `/recover-new-jobs` endpoint according to the sample implementation in the previous topics. Then, suppose you wanted to find all incomplete submissions where the primary insured's first name is "Francine" and last name is "Michaels". The request would look like this.

```
POST /job/v1/recover-new-jobs

{
  "data": {
    "attributes": {
      "firstName_Ext": "Francine",
      "lastName_Ext": "Michaels"
    }
  }
}
```

Suppose there is one job that matches the search criteria. The response would look like this. For the sake of clarity, some properties have been omitted.

```
RESPONSE BODY:
{
    "count": 1,
    "data": [
        {
            "attributes": {
                "account": {
                    "displayName": "C000143542",
                    "id": "pc:Srx-YL_xDmdj455LyQtIu",
                    "type": "Account",
                    "uri": "/account/v1/accounts/pc:Srx-YL_xDmdj455LyQtIu"
                },
                "id": "pc:SVAyPWiBiNuBrYFPjd7ZX",
                "jobNumber": "47586734721",
                "jobStatus": {
                    "code": "Quoted",
                    "name": "Quoted"
                },
                "jobType": {
                    "code": "Submission",
                    "name": "Submission"
                },
                "policy": {
                    "displayName": "P000143542",
                    "id": "pc:SH21ByixaZ0O-2Hteuwc8",
                    "type": "Policy",
                    "uri": "/policy/v1/policies/pc:SH21ByixaZ0O-2Hteuwc8"
                },
                "policyNumber": "P000143542",
                "primaryInsured": {
                    "displayName": "Francine Michaels",
                    "id": "test_pp:2",
                    "type": "PolicyContact",
                    "uri": "/job/v1/jobs/pc:SVAyPWiBiNuBrYFPjd7ZX/contacts/test_pp:2"
```

```
                }
            },
            ...

RESPONSE HEADER:
GW-Access-Token:
eyJhbGciOiJIUzUxMiIsImtpZCI6ImN1cnJlbnRfa2V5IiwidHlwIjoiSldUIn0.eyJleHAiOjE1OTU1NjYzN
jksImdyb3VwcyI6WyJwYy5hbm9ueW1vdXMiXSwiaWF0IjoxNTk1NTU1NTY5LCJpc3MiOiJQQyIsImp0aSI6In
JCMEVDYVdoOVh1Y2U5M3cyYkFETnVXOUszdkZoUGxuS0FpbVR3NVdFNWNueW9VM0FBQUFCQS4uIiwicGNfYWN
jb3VudE51bWJlcnMiOlsiMjU1ODM2MzEzOCJdLCJzdWIiOiJhdXRoIiwidGVuYW50X2lkIjoiTm9UZW5hbnQi
LCJ0eXBlIjoiYWNjb3VudEhvbGRlciJ9.Ix4GCz4nJg_QM3AsC-jVyZU_V8ysGBgWfvIxAIS59t7EN2C6Pi2Q
gJRs09y0ThqFX-_1-ucD58Vunqs5dMivJg
```

Configuring general new job recovery behavior

New job recovery behavior is controlled by the following application configuration parameters in `config.xml`:

- `RestJobRecoveryExpirationDateThreshold`
  - The number of days past the current date when a job becomes too old to be recoverable.
  - The default value is 7
- `RestJobRecoveryMaxResult`
  - This maximum number of jobs that are recoverable
  - The default value is 5

For more information on modifying application configuration parameters, see the *Configuration Guide*.

chapter 64

Configuring calls between InsuranceSuite applications

In the base configuration, InsuranceSuite applications do not make calls to the Cloud API endpoints of other InsuranceSuite applications. For example, the base configuration of ClaimCenter does not make calls to Cloud API for PolicyCenter. However, through configuration, an insurer can enable this type of interaction.

To describe this behavior, this documentation uses the following terminology:

- The calling application is the *caller application*.
- The application whose Cloud API endpoints are being called is the *responder application*.
- `<callerAppCode>` and `<responderAppCode>` are values that represent the two-letter code for the appropriate application (`cc`, `pc`, `bc`, or `ab`).

Configuration is needed in both applications to ensure the caller application has the proper authorization.

API role in the responder application

The responder application must have an API role that defines the endpoint access granted to the caller application. This role must be named `gw_<callerAppCode>_ext`.

Scopes for the caller application

The caller application is considered to be a standalone service. Guidewire automatically registers it with the following scopes:

- `<responderAppCode>.service`
  - This allows the caller application to use the resource access strategy for services as defined in the responder application.
- `scp.<responderAppCode>.gw_<callerAppCode>_ext`
  - This is used to verify that the caller application can be granted the `gw_<callerAppCode>_ext` role.

Example

Suppose that you have instances of both ClaimCenter and PolicyCenter. You want ClaimCenter to make calls to Cloud API for PolicyCenter. To enable this, the following must be true:

- In PolicyCenter, there is an API role named `gw_cc_ext`. This role is declared in a file whose name is `gw_cc_ext.role.yaml`.

- ClaimCenter is registered with the following scopes:
  - `pc.service`
  - `scp.pc.gw_cc_ext`
- When ClaimCenter makes a call to PolicyCenter, the JWT includes this claim:

```
"scp": [
  "pc.service",
  "scp.pc.gw_cc_ext",
  <additional tenant, project, and planet values>
]
```

Enabling cross-application Cloud API calls

Guidewire automatically registers every cloud instance of every InsuranceSuite application with all of the scopes needed to make cross-application calls. Specifically:

- Instances of ClaimCenter have the following scopes registered:
  - `pc.service`
  - `bc.service`
  - `ab.service`
  - `scp.pc.gw_cc_ext`
  - `scp.bc.gw_cc_ext`
  - `scp.ab.gw_cc_ext`
- Instances of PolicyCenter have the following scopes registered:
  - `cc.service`
  - `bc.service`
  - `ab.service`
  - `scp.cc.gw_pc_ext`
  - `scp.bc.gw_pc_ext`
  - `scp.ab.gw_pc_ext`
- Instances of BillingCenter have the following scopes registered:
  - `cc.service`
  - `pc.service`
  - `ab.service`
  - `scp.cc.gw_bc_ext`
  - `scp.pc.gw_bc_ext`
  - `scp.ab.gw_bc_ext`
- Instances of ClaimCenter have the following scopes registered:
  - `cc.service`
  - `pc.service`
  - `bc.service`
  - `scp.cc.gw_ab_ext`
  - `scp.pc.gw_ab_ext`
  - `scp.bc.gw_ab_ext`

However, none of the applications have the corresponding API roles.

To enable cross-application Cloud API calls, you must create one API role in each responder application. The API role's name must adhere to the `scp.<responderAppCode>.gw_<callerAppCode>_ext` naming convention. The API role must also define whatever endpoint access is appropriate for the caller application.

For more information on defining API roles, see “Endpoint access” on page 373.

Making cross-application calls

Whenever any service calls a Cloud API endpoint, the service must acquire a JWT from Guidewire Hub and include that JWT with the call. In the JWT, the `scp` claim must specify the service's resource access strategy and API role.

When the service is a Guidewire application that is calling Cloud API endpoints of another Guidewire InsuranceSuite application (ClaimCenter, PolicyCenter, BillingCenter, or ContactManager), the caller application is already registered with Guidewire. To acquire a JWT, the caller Guidewire application must call platform code in the same as if the caller was making a call to an Integration Gateway API.

For more information on how to do this, refer to the *Integration Gateway* documentation.

chapter 65

Troubleshooting auth issues

When you encounter an auth issue, the single most effective action you can take to troubleshoot the issue is to set the *REST.Config* and *REST.Request* loggers to DEBUG, reproduce the error, and then check the application log for the resulting error messages. Most auth issues trigger log messages that identify the source of the problem.

Be aware of the following:

- In a standalone instance of PolicyCenter, there is only one log. By default, it is located in `/tmp/gwlogs/GenericCenter/logs`. In a Cloud or multiclustered instance, there may be multiple logs.
- Some messages are written to the log only when the *REST.Config* and *REST.Request* loggers are set to DEBUG. When troubleshooting auth issues, Guidewire recommends setting these loggers to DEBUG.

For more information on how to configure general PolicyCenter logging, refer to the *Administration Guide*.

Examples of auth errors in the log

The following section lists auth problems that developers have encountered in the past and the way these errors were reflected in the logs. This is not a comprehensive list of all of the errors that you could encounter. It is provided solely to illustrate some of the ways in which log messages can help identify auth problems.

Failing to list your URL as an allowed issuer when enabling asymmetric encryption

Bearer token authentication for Cloud API uses *asymmetric encryption*. To verify a given JWT, PolicyCenter executes an *asymmetric public key lookup*. Periodically, PolicyCenter must request the keys used in these lookups from Guidewire Hub.

When you register PolicyCenter with Guidewire Hub, you are given an auth server URI and a tenant ID. For PolicyCenter to be able to request keys from Guidewire Hub, you must add the auth server URI to your PolicyCenter instance.

If you do not enable asymmetric encryption, then Cloud API rejects calls that attempt to use bearer token. A message similar to the following is listed in the logs. (For readability, some hard returns have been added to the text below.)

```
error","eventType":"STATUS","eventMessage":"Operation status","contextMap":{"path":"/claim/v1/
claims/demo_sample:1/","query":"","method":"GET","from":"[0:0:0:0:0:0:0:1]","async":false,
"pathTemplate":"/claim/v1/claims/{claimId}","apiFqn":"ext.claim.v1.claim_combined_ext-1.0",
"user":"","sub":"","clientId":"","status":401,"error":"gw.api.modules.rest.framework.v1.plugin
.security.InvalidAuthenticationTokenException","userMessage":"Authorization failed","devMessage":
 "JWT verification failed: Encountered JWT issuer 'http://localhost:8080/cc/rest' that has not
been configured in 'allowedIssuers' by the SignatureKeyProviderPlugin. Allowed issuers are []",
 "auth":"bearer","tokenType":"unrecognized","userContextType":"none","origin":"","userAgent":
"PostmanRuntime/7.26.8","async.1":false,"jutroSdkVersion":""},"elapsedTimeMs":20,"serverId":
"localinstance2us","timestamp":"20230712T144133.394-0700"}
```

For more information on enabling asymmetric encryption, see “Enabling asymmetric encryption” on page 369.

Failing to allowlist an endpoint in an API role file

By default, callers do not have access to any endpoints. In order to use endpoints, a caller must be associated with an API role, and the API role must explicitly list the endpoints that can be used by callers with that role.

If a caller attempts to call an endpoint that is not explicitly listed in their API roles, then Cloud API rejects the call. A message similar to the following is listed in the logs. (For readability, some hard returns have been added to the text below.)

```
localinstance2us aapplegate 60bd8cfe-ee14-4580-99f4-670d1f4d4744 2023-07-17 16:20:50,125 INFO
<RestRequest[ GW.PL ]> {"message":"Client error","eventType":"STATUS","eventMessage":"Operation
status","contextMap":{"path":"/claim/v1/claims/demo_sample:1/exposures","query":"","method":
"GET","from":"[0:0:0:0:0:0:0:1]","async":false,"pathTemplate":"/claim/v1/claims/{claimId}/
exposures","apiFqn":"ext.claim.v1.claim_combined_ext-1.0","user":"aapplegate","sub":"",
"clientId":"","status":404,"error":"gw.api.rest.exceptions.NotFoundException","userMessage":
"No view permission for the requested resource","devMessage":"","auth":"basic","tokenType":
"none","userContextType":"none","origin":"","userAgent":"PostmanRuntime/7.26.8","async.1":
false,"jutroSdkVersion":""},"elapsedTimeMs":5,"serverId":"localinstance2us","timestamp":
"20230717T162050.124-0700"}
```

Cloud API writes a log message for each check it performs to see if paths and operations are accessible. The log message lists the roles that have been checked and whether the roles grant access. For example, suppose a caller attempts to access GET `/claims/{claimId}/exposures`. Also, suppose the caller has two roles, "Adjuster" and "Trusted for Sensitive Claims", but neither role grants access to this endpoint. When the call fails, Cloud API writes the following to the log.

```
pniemeyer2us aapplegate 2b79fe61-75af-4d77-aedb-838864882163 2023-07-17 16:20:44,274 DEBUG
    Checking access to path and operation based off roles.
    Path: '/claim/v1/claims/demo_sample:1/exposures'
    httpMethod: 'GET'
    Roles: {Trusted for Sensitive Claims, Adjuster}
    Roles acquired from: token
    Roles grant access: false
```

When trying to track down "no <action> permission" errors, it may be helpful to search for the error messages containing the string `Roles grant access: false`.

Note that the error message says `Roles acquired from: token`. This means that the roles were retrieved using information from the JWT, but the roles may not have been explicitly listed on the JWT. For internal users, roles come from a database query to the `cc_user`, `pc_user`, or `bc_user` table using information in the JWT. For external users, roles could come from the JWT or from information retrieved by the IExpandTokenPlugin.

For more information on endpoint access, see “Endpoint access” on page 373.

Failing to provide a service role scope in a JWT for a service call

When a call is made by a service, the JWT must contain a `scp` claim with one of the following values:

- `cc.service` (for calls to Cloud API for ClaimCenter)
- `pc.service` (for calls to Cloud API for PolicyCenter)
- `bc.service` (for calls to Cloud API for BillingCenter)

If a service sends a call, but the correct scope is not included in the JWT, then Cloud API rejects the call.

For more information on registering scopes, see “Registering the caller application with Guidewire Hub” on page 372.

Find out why a call failed using correlation IDs

About this task

A *correlation ID* is a unique string that is used to identify a single API request.

Every time Cloud API processes a call, it expects a `X-Correlation-ID` to be included in the header. If there is no correlation ID attached, Cloud API generates one. Correlation IDs are logged and passed along to monitoring tools like Datadog. You can identify where a call failed by gathering all logging statements that share the same correlation ID.

To find log statements by correlation ID, do the following.

Procedure

1. Turn the `REST.Request` and `REST.Config` loggers to DEBUG. For more information on how to temporarily set log levels, see *InsuranceSuite Observability*.
2. Execute the failing call again.
3. If you are working in a local instance using Guidewire Studio, you can access the logs in the console.
   a) Open the console.
   b) Identify a log statement related to the failing call.
   c) Identify the correlation ID in the log statement.

d) Search the log for the correlation ID to find log statements related to the failing call. The logs look something like this:

```
qtp487764004-21  default_data:1 28b788c9-af91-443f-b9ad-15927ffcb2a8 15:04:16,266 INFO REST.Request <
qtp487764004-23  default_data:1 fe94e208-a3d5-41dc-8950-0ba71594730f 15:04:16,309 INFO REST.Request <
qtp487764004-24  default_data:1 822ac002-a53a-4a04-95ff-0e8b1df0b70c 15:04:16,358 INFO REST.Request <
qtp487764004-20           5a888c1d-13cd-40fb-9925-07739ddb1eec 15:04:16,730 DEBUG REST.Request The to
qtp487764004-20           5a888c1d-13cd-40fb-9925-07739ddb1eec 15:04:16,733 DEBUG REST.Request User m
qtp487764004-20  cc:SbyKcVIGz09uW-h69j8EM 5a888c1d-13cd-40fb-9925-07739ddb1eec 15:04:16,747 DEBUG RES
    Path: '/admin/v1/users'
    httpMethod: 'GET'
    Roles: {Adjuster}
    Roles acquired from: token
    Roles grant access: false
qtp487764004-20  cc:SbyKcVIGz09uW-h69j8EM 5a888c1d-13cd-40fb-9925-07739ddb1eec 15:04:16,747 DEBUG RES
gw.api.rest.exceptions.NotFoundException Create breakpoint : No view permission for the requested resource
    at gw.api.rest.auth.ApiRoleApiAuthorizer.checkPathAndMethod(ApiRoleApiAuthorizer.java:33) ~[pl-50
    at com.guidewire.pl.rest.dispatch.handler.SwaggerOperationHandler.authorizeUser(SwaggerOperationH
    at com.guidewire.pl.rest.dispatch.handler.SwaggerOperationHandler.handleOperation(SwaggerOperatio
    at com.guidewire.pl.rest.dispatch.SwaggerRestDispatcher.handleOperation(SwaggerRestDispatcher.jav
    at com.guidewire.pl.rest.dispatch.SwaggerRestDispatcher.handleOperationWithErrorHandling(SwaggerR
    at com.guidewire.pl.rest.dispatch.SwaggerRestDispatcher.handleRequestImpl(SwaggerRestDispatcher.j
    at com.guidewire.pl.rest.dispatch.SwaggerRestDispatcher.handleSynchronousRequest(SwaggerRestDispa
    at com.guidewire.pl.rest.dispatch.SwaggerRestDispatcher.handleRequest(SwaggerRestDispatcher.java:
    at com.guidewire.pl.rest.SwaggerRestServlet.service(SwaggerRestServlet.java:70) ~[pl-50.11.0-alph
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:790) ~[javax.servlet-api-3.1.0.jar:3.1
    at org.eclipse.jetty.servlet.ServletHolder$NotAsync.service(ServletHolder.java:1450) ~[jetty-serv
    at org.eclipse.jetty.servlet.ServletHolder.handle(ServletHolder.java:799) ~[jetty-servlet-9.4.54.
    at org.eclipse.jetty.servlet.ServletHandler$ChainEnd.doFilter(ServletHandler.java:1656) ~[jetty-s
    at org.eclipse.jetty.websocket.server.WebSocketUpgradeFilter.doFilter(WebSocketUpgradeFilter.java
    at org.eclipse.jetty.servlet.FilterHolder.doFilter(FilterHolder.java:193) ~[jetty-servlet-9.4.54.
    at org.eclipse.jetty.servlet.ServletHandler$Chain.doFilter(ServletHandler.java:1626) ~[jetty-serv
    at org.eclipse.jetty.servlet.ServletHandler.doHandle(ServletHandler.java:552) ~[jetty-servlet-9.4
    at org.eclipse.jetty.security.SecurityHandler.handle(SecurityHandler.java:600) ~[jetty-security-9
    at org.eclipse.jetty.servlet.ServletHandler.doScope(ServletHandler.java:505) ~[jetty-servlet-9.4.
    at org.eclipse.jetty.io.AbstractConnection$ReadCallback.succeeded(AbstractConnection.java:311) ~[
    at org.eclipse.jetty.io.FillInterest.fillable(FillInterest.java:105) ~[jetty-io-9.4.54.v20240208.
    at org.eclipse.jetty.io.ChannelEndPoint$1.run(ChannelEndPoint.java:104) ~[jetty-io-9.4.54.v202402
qtp487764004-20  cc:SbyKcVIGz09uW-h69j8EM 5a888c1d-13cd-40fb-9925-07739ddb1eec 15:04:16,763 INFO REST
rocess-startup-bg-thread systemTables:2                                    15:05:00,051 INFO Server
Workflow-1511            525d61a7-2c80-43bb-94b5-11e48f71dc63 15:05:00,063 INFO Server.BatchProcess
```

4. To see the logs for a cloud instance, you can use Datadog to find the logging statements for the call.

a) Open Datadog.

b) Go to the Log Explorer by selecting **Logs** > **Explorer**.

c) Apply a query filter to make it easier to find logs for your environment. For information on query filters in Datadog, see *InsuranceSuite Observability*.

d) Select a log statement related to the failing call. An information pane opens.

e) Under **Event Attributes**, select the correlation id. A pop-up menu opens.

f) Select **Filter by (correlation ID)**.

You can now view and export the log statements for the failing call.

Note: If you do not see DEBUG and INFO statements in the Datadog log, there may be index configurations which prevent these statements from being logged. See Indexes for information on Datadog indexes.

part 11

ContactManager authentication

This section provides information on how ContactManager authentication differs from PolicyCenter authentication.

chapter 66

ContactManager authentication

Authentication for Cloud API for ContactManager is nearly identical to authentication for Cloud API for PolicyCenter. This topic identifies the differences between the two. If a topic is not explicitly discussed here, you can assume that it behaves in the same way for the two applications.

Supported caller types

Cloud API for ContactManager supports the following caller types:

- Basic auth
- Internal users (using bearer token auth)
- Standalone services
- Services with user context (where the user context references an internal user)
- Services with service account mapping
- Unauthenticated callers

Cloud API for ContactManager does not have any resource access files. Therefore, Cloud API for ContactManager does not support:

- External users
- Services with user context where the user named in the context is an external user

Cloud API for ContactManager does not support the anonymous user flow. This flow is supported in PolicyCenter only.

Resource access for ContactManager

The base configuration of Cloud API for ContactManager does not come with any resource access files. As a result, callers are not restricted by resource access.

Tag-based access to contacts

Contacts in ContactManager have contact tags. A *contact tag* is a tag that identifies one or more broad relationships that the contact has with the insurer.

The base application includes three tags:

- Client (a policyholder)
- Claim party (a contact involved in a claim)
- Vendor (a contact that provides service for a claim)

In order to view or edit a contact with a given type of tag, a user must have permissions for that action and for that tag. For example:

- In order to view contacts with the client tag, a user must have a "view client" permission.
- In order to edit contacts with the vendor tag, a user must have a "edit vendor" permission.

There are also two system permissions, `viewanytag` and `editanytag`, that let the associated users view or edit any contact, regardless of the contact's tags.

Base configuration behavior of tag-based permissions

In the ContactManager base configuration, all users have either the `viewanytag`, the `editanytag`, or both. This means that all base configuration users bypass tag-based permissions.

Cloud API access and tag-based permissions

Tag-based permissions apply to callers accessing contacts through Cloud API. In order to view or edit contacts with a given type of tag, the following must be true:

- For internal users and services with service account mapping:
  - Either the caller has an appropriate "view <tag>" permission and/or an appropriate "edit <tag>" permission, OR
  - The caller has the `viewanytag` permission and/or the `editanytag` permission
- For standalone services and services with user context:
  - Either the proxy user has an appropriate "view <tag>" permission and/or an appropriate "edit <tag>" permission, OR
  - The proxy user has the `viewanytag` permission and/or the `editanytag` permission

Note that for internal users and services with service account mapping, the permission check occurs against each caller individually. This means that, for a given "view <tag>" or "edit <tag>" scenario, some callers will be given access to the contact while others will not.

However, for standalone service, and services with user context, the permission check occurs against the one user designated as the proxy service user. This means that, for a given "view <tag>" or "edit <tag>" scenario, either all callers of that caller type will have access to the contact, or none of them will.