Guidewire PolicyCenter™ for Guidewire Cloud

Cloud API Business Flows and Configuration Guide

Release: 2021.11

GUIDEWIRE

Product Name: Guidewire PolicyCenter for Guidewire Cloud
Product Release: 2021.11
Document Name: Cloud API Business Flows and Configuration Guide
Document Revision: 19-March-2023

Contents

Part 2
Business flows: PolicyCenter

Part 3
Business flows: Framework APIs

Part 4
Configuring the Cloud API

Support

For assistance, visit the Guidewire Community.

Guidewire customers

https://community.guidewire.com

Guidewire partners

https://partner.guidewire.com

part 1

Consuming the Cloud API

The *InsuranceSuite Cloud API* is a set of RESTful system APIs that caller applications can use to request data from or initiate action within an InsuranceSuite application. These APIs provide content for the REST API framework that is present in all InsuranceSuite applications. The APIs are built using the Swagger 2.0 Specification. These are also referred to as the *system APIs*.

The following topics discuss how caller applications can consume the system APIs in Cloud API. This includes how to:

- Construct GET requests to query for data
- Construct POST requests to create new data
- Construct PATCH requests to modify existing data
- Construct DELETE requests to remove data
- Use query parameters to refine response payloads
- Reduce the number of calls needed to accomplish a business flow
- Prevent lost updates using checksums

chapter 1

REST API fundamentals in Cloud API

This topic discusses the fundamental concepts of REST APIs and how those concepts are used in Cloud API. This topic is intended primarily for developers with minimal experience using REST APIs.

For information on functionality specific to the APIs in the Cloud API (such as the APIs that exist in the base configuration, the beta APIs, or the `openapi.json` endpoints), see "Overview of the system APIs in Cloud API" on page 23.

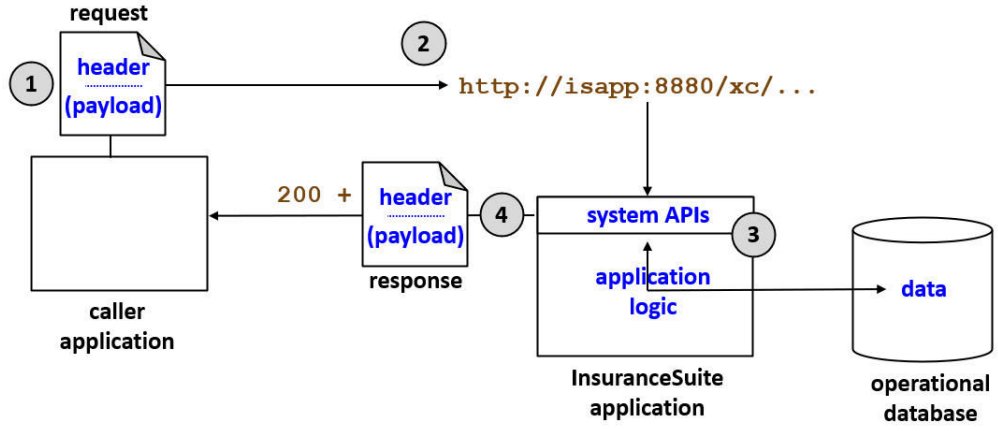
The InsuranceSuite Cloud API

The *InsuranceSuite Cloud API* is a set of RESTful system APIs that caller applications can use to request data from or initiate action within an InsuranceSuite application. These APIs provide content for the REST API framework that is present in all InsuranceSuite applications. The APIs are built using the Swagger 2.0 Specification. These are also referred to as the *system APIs*.

The system APIs can be used by browser-based applications and service-to-service applications. This documentation uses the term *caller application* to generically refer to any application or service calling a system API.

Making system API calls

The following diagram provides a high-level overview of the interaction between the caller application and the system APIs.



1. The caller application constructs a request object. The request object consists of:

- A header, which can contain authentication information and other metadata.
- A payload, when necessary.

2. The caller application sends the request to the system API using an HTTP command.
   - The command calls a specific API endpoint.
   - The command may include query parameters that further identify the data that is desired.
   - The request object is sent with the command.
3. The system API processes the request.
   - This activity uses all of the InsuranceSuite application logic, such as validation logic and pre-update rules.
   - The request is restricted by authorization controls within the system APIs.
4. The system API responds with an HTTP response code (such as 200 for success) and a response object. The response object consists of:
   - A header
   - A payload, when necessary.

System APIs and InsuranceSuite logic

In the software industry, some RESTful APIs are configured to interact directly with the database. The system APIs are not configured to behave this way. The system APIs interact with operational data only through the layer of the application's business logic. Therefore, the system APIs always leverage the existing business logic of the application.

For example:

- Suppose an internal user does not have permission to create an activity. If the internal user attempts to create an activity through the system APIs, the attempt results in an insufficient permissions error.
- Suppose there is a validation rule that requires an activity's due date to be set in the future. If an external system attempts to create an activity with a due date in the past, the attempt results in a validation error.
- Suppose there is a pre-update rule that creates an approval activity whenever a document is marked as "Final". If an external system creates a "Final" document through a system API, the pre-update rule will create an approval activity.

Resources

The primary mechanism for passing information between the caller application and PolicyCenter is the resource. A *resource* is an instance of data that you can create, modify, delete, or query for. Resources are defined in JSON schema files.

Every resource has a type. The type defines the Guidewire data model entities that the resource maps to. For example, `Activity` resources map to the `Activity` data model entity. In most cases, each resource maps to a single data model entity. However, there are some resources which map to multiple data model entities. For example, the `AccountContact` resource maps to three data model entities in PolicyCenter: `AccountContact`, `Contact`, and `AccountContactRole`.

Resources contain a set of fields. Each *field* stores information about the resource. Depending on the context, fields are also referred to as *properties* or *attributes*.

Resources are exchanged in the payloads of the request and response objects. The *payload* is a block of JSON-formatted text that contains fields from the relevant resources and their values. The following is a portion of the response payload for an `Activity` resource.

```
"attributes": {
    "assignedGroup": {
        "displayName": "Auto1 - TeamA",
        "id": "demo_sample:31"
    },
    "assignedUser": {
        "displayName": "Andy Applegate",
        "id": "demo_sample:1"
    },
    "dueDate": "2020-11-16T08:00:00.000Z",
    "id": "xc:20",
    "priority": {
```

```
        "code": "urgent",
        "name": "Urgent"
    "subject": "Contact claimant"
}
```

Note that a field can store:

- A scalar value, such as the `subject` field.
- A set of values, such as the `assignedUser` field. This is referred to as an *inline object*.
- An array of objects. (There is no example of this in `Activity`. If there were, the field name would be followed by square braces ([and]) delimiting the array. Each array member would be listed in curly braces ({ and }).

Every resource can be uniquely defined by its *resource ID*. This value maps to the data model entity's `PublicID` field. The activity in the previous example is activity xc:20.

A single resource is called an *element*. For example, `/contact/xc:203` is an element. (In some REST API literature, this is also referred to as a *singleton*.)

A set of resources is called a *collection*. For example, `/contact/xc:203/addresses` (the addresses associated with contact xc:203) is a collection.

Endpoints

Every API consists of a set of endpoints. An *endpoint* is a command that a caller application can use to request data from or trigger action in PolicyCenter. For example, the `/common/v1/activities` endpoint can be used to either request data about PolicyCenter activities or trigger actions related to PolicyCenter activities. When referenced in documentation, endpoints start with a slash (/), such as the `/activities` endpoint. Endpoints are defined in Swagger schema files.

In Cloud API, the endpoint path (the full name of the endpoint) includes the API and the version. For convenience sake, the documentation often refers to endpoints using only the last part of the endpoint path. For example, the `/rest/common/v1/activities` endpoint is often referred to simply as "the `/activities` endpoint".

Endpoints in Cloud API have four fundamental components: root resources, child resources, operations, and paths.

Root resources

Every endpoint has a root resource. The *root resource* is the resource which the endpoint creates, updates, deletes, or queries for. Every call to an endpoint makes use of the root resource.

For example, the root resource for the `/common/v1/activities` endpoint is `Activity`. This endpoint is used to potentially create, update, delete, or queries for activities.

Child resources

Most endpoints also include child resources. A *child resource* is a resource related to the root resource. Child resources improve the usability of an endpoint by providing access to information related to the root resource. For example, the `/common/v1/activities` endpoint has one child resource - `Notes`. This means you could use the endpoint to:

- Query for a specific activity (and only the activity)
- Query for a specific activity and its related notes

Every call to an endpoint must make use of the root resource. The use of child resources is optional.

Inline and included resources

Child resources can be declared either as inline resources or included resources.

- An *inline resource* is a resource that appears in the `attributes` section of the payload inline with the other root resource fields, such as an `Activity` resource's `assignedUser` field. These resources may be included in a response by default and can be controlled through the `fields` query parameters.

- An *included resource* is a resource that appears in the `included` section at the bottom of the payload, such as an `Activity` resource's `Notes`. These resources are not included in a response by default and must be controlled through the `included` query parameters.

For more information on inline and included resources, see "GETs and response payload structures" on page 35.

Operations

An *operation* is a type of action a caller application can take on a resource through an endpoint. Operations are also referred to as *verbs* or *methods*. The system APIs support the following subset of HTTP operations:

- **GET** - Used to request resources.
- **POST** - Used to create resources. Also used to execute business actions, such as quoting a submission or submitting a claim.
- **PATCH** - Used to update resources.
- **DELETE** - Used to delete resources.

Every endpoint supports one or more of these operations. For example, in the Common API:

- The `notes/{noteId}` endpoint supports GET, PATCH, and DELETE.
- The `/activities` endpoint supports only the GET operation.

The HTTP operations are designed for CRUD operations (Create, Read, Update, Delete). Some business processes in InsuranceSuite applications are available to the system APIs but do not readily map to any of these operations, such as assigning objects, closing objects, or approving objects. As a general rule, the custom actions that trigger these processes use the POST operation.

Operation mapping to elements and collections

In general:

- You can GET either an element or a collection.
- You POST a collection to create an element.
- You POST to a custom action (to execute a business action).
- You PATCH an element.
- You DELETE an element.

For example:

Operation	On endpoint...	Does the following...
GET	`/activities`	Returns all activities assigned to the current user
GET	`/activities/{activityId}`	Returns the details for the specified activity
POST	`/activities/{activityId}/notes`	Adds a new note to the specified activity
POST	`/activities/{activityId}/assign`	Assigns the activity
PATCH	`/activities/{activityId}`	Updates information on the specified activity
DELETE	`/notes/{NoteId}`	Deletes the specified note

Contrasting endpoints and operations

Technically speaking, when an endpoint supports multiple operations, it is still a single endpoint. However, in casual discussion, each operation is sometimes referred to as a separate endpoint. For example, consider the following:

- GET `/common/v1/activities`
- POST `/common/v1/activities`

This is a single endpoint (`/common/v1/activities`) that supports two operations (GET and POST). However, in a casual sense, it is sometimes referred to as two endpoints (the GET `/activities` endpoint and the POST `/activities` endpoint).

The PUT operation

Within REST API architecture, there are two operations that modify existing resources - PATCH and PUT. PATCH is used to modify a portion of an existing resource (while leaving other aspects of it unmodified). PUT is used to replace the entire contents of an existing resource with new data. The system APIs support the PATCH operation, but not the PUT operation. This is because nearly every operation that modifies an InsuranceSuite object modifies only a portion of it while keeping the rest of the object untouched. This behavior maps to PATCH, but not to PUT.

Paths

Every endpoint has a path. The *path* is the portion of the URL used by caller applications to identify the specific endpoint.

For Cloud API, every path consist of the following pattern:

```
rest/<APIname>/<APImajorVersion>/<endpointName>
```

For example, consider the path: `rest/common/v1/activities`:

- `common` is the name of the API to which the endpoint belongs.
- `v1` is the major version number of the API
- `activities` is the endpoint name

The major version number provides information about the backwards compatibility of the endpoint. For more information, see “Cloud API versions” on page 23.

A path can also contain a reference to a specific resource. For example, the path `/activities/xc:20/notes` refers to the notes for activity xc:20. When a path includes a reference to a specific resource, the generic path name is specified using `{typeId}`, where *type* is the resource type. For example, the generic path for `/activities/xc:20/notes` is `/activities/{activityID}/notes`. A reference to a specific resource in a path is known as a *path parameter*.

For most endpoints, the endpoint name is the same as the resource name, with the following conventions and caveats:

- If the endpoint's root resource is an element, the endpoint name ends in a singular noun (such as `/activity`) or a resource reference (such as `/activity/{activityId}`).
- If the endpoint's root resource is a collection, the endpoint name ends in a plural noun (such as `/activities`).
- If the endpoint executes a business action, the endpoint name ends in a verb (such as `/{activityId}/assign`).
- The endpoint name is often close to, but not identical to, the resource name
  - Endpoint names use lower case, whereas resource names use mixed case (for example, the root resource for the `/activity` endpoint is `Activity`)
  - Endpoint names use hyphens to separate words, whereas resource names do not (for example, the root resource for the `/accounts/{accountId}/activity-patterns` endpoint is `ActivityPattern`)
  - In some cases, the endpoint name may differ from the root resource name (for example, the root resource for the `/accounts/{accountId}/contacts` endpoint is `AccountContact`)

Requests and responses

Requests

A *request* is a call from a caller application to an endpoint to either query for data or initiate action.

Requests are made using URLs. Request URLs have the following components:

```
https://iap:8880/xc/rest/common/v1/activities/xc:207?fields=assignedGroup
\_______________/\_____________________________/\__________________/
   application URL          endpoint path             query parameters
```

- **Application URL** - The URL to the InsuranceSuite application.
  - This value is required.

- **Endpoint path** - The path to the specific endpoint that the request is requesting.
  - This value is required.
  - Endpoint paths end either with a resource name (such as `.../activities`) or the ID of a specific element (such as `.../activities/xc:207` in the example above). The ID of a specific element is also referred to as a *path parameter*.
- **Query parameters** - This is a set of query parameters that further defines the data that is desired in the response. For most endpoints, query parameters are optional.
  - For example, when you add `?fields=assignedGroup`, you are specifying that the only field you want returned in the response is the `assignedGroup` field.
  - There are a small number of endpoints that require a query parameter. For example, to prevent resource-intensive calls, the Job API's `/graph-schema` endpoint requires a `product` parameter.

Some requests require a payload. The *payload* is a block of JSON-formatted text that contains information about one or more resources associated with the operation. Typically:

- GETs and DELETEs do not require request payloads.
  - For a GET, you only need to identify the resource you want information about, and this is done in the URL.
  - For a DELETE, you only need to identify the element to delete, and this is done in the URL.
- POSTs and PATCHes do require request payloads.
  - For a POST, you must specify data about the element to create.
  - For a PATCH, you must specify the data about the element that must be updated.

Responses

A *response* is the set of information returned by an API endpoint for a request to the caller application.

Some responses include a payload. The payload contains information about one or more resources that are returned by the operation. For example, for a request to get all open activities assigned to a given user, the response includes a payload with information about the open activities. For more information about the payload structure, see “GETs and response payload structures” on page 35.

The outcome of the operation is specified as an HTTP status code, also referred to as a response code. These codes are three-digit numbers. The general meanings of these codes are defined in the following table:

Status code	Category	Meaning
1xx	Information	Used for transfer protocol-level information
2xx	Success	The server accepted the client request successfully. (The code 200 indicates a successful GET or PATCH. 201 indicates a successful POST. 204 indicates a successful DELETE.)
3xx	Redirection	The client must take some additional action in order to complete its request.
4xx	Errors (client-side)	An error condition occurred on the client side of the HTTP request and response.
5xx	Faults (server-side)	An error condition occurred on the server side of the HTTP request and response.

Testing requests and responses

Developers who work with system APIs typically use a tool that can send requests and get responses within an acceptable amount of time. Guidewire recommends Postman. This tool has the ability to:

- Save API calls, including headers and payloads
- Save collections of calls
- Automatically create a collection of calls for a schema's paths by importing the Swagger schema file
- Share collections with other developers on your team

For more information and to download the tool, see https://www.postman.com/.

Note: Swagger UI is also able to send requests to a working API and show responses. However, the system APIs are significantly robust, and performance time for getting responses to requests can be unacceptably long. Guidewire recommends using Swagger UI only for viewing system API documentation.

Tutorial: Set up your Postman environment

The system API documentation contains a set of tutorials that guide you through examples of how to send requests and review the responses. All of these tutorials assume the following base environment:

- A default instance of PolicyCenter installed on your machine that contains only the Large sample data set.
- An instance of Postman.

This tutorial walks you through the process of setting up this environment.

Note: If your instance of PolicyCenter is installed on a different machine, you will need to adjust the endpoint URLs provided in the tutorials. Also, if you create data in addition to the Large sample data, then your responses may differ from the ones described in the tutorials.

Tutorial steps

1. Install Postman. (For more information, refer to https://www.postman.com/.)
2. Start PolicyCenter and load the Large sample data set.

You can test your environment by sending your first Postman request.

1. Open Postman.
2. Start a new request by clicking the + to the right of the **Launchpad** tab.
3. Under the **Untitled Request** label, make sure that **GET** is selected. (This is the default operation.)
4. In the **Enter request URL** field, enter the following URL: `http://localhost:8180/pc/rest/common/v1/activities`
5. Every tab in Postman requires authorization information to execute the request. To provide sufficient authorization information:
   a. Click the **Authorization** tab.
   b. For the **Type** drop-down list, select *Basic Auth*.
   c. In the **Username** field, enter `aapplegate`.
   d. In the **Password** field, enter `gw`.
6. Click the **Send** button to the right of the request field.

Checking your work

Once a response has been received, its payload is shown in the lower portion of the Postman interface. If your environment has been set up correctly, the first few lines of the response payload are:

```
{
    "count": 13,
    "data": [
        {
        "attributes": {
            "activityPattern": "uw_review_contingency",
            "activityType": {
                "code": "general",
                "name": "General"
            },
            "assignedByUser": {
                "displayName": "Alice Applegate",
                "id": "pc:105"
        },
```

Troubleshooting: No response

Requests can be sent only to running applications. All of the tutorials in this documentation require that PolicyCenter is running. If you send a request when the application is not running, you will see an error similar to the following:

```
Could not get any response
```

```
There was an error connecting to http://localhost:8180/pc/rest/common/v1/activities.
```

Troubleshooting: NotFoundException

Unless it is stated otherwise, all of the tutorials use basic authentication and the `aapplegate` user. If you encounter a `NotFoundException` such as the following example, this could be caused by not providing correct authentication information for this user.

```
"status": 404,
"errorCode": "gw.api.rest.exceptions.NotFoundException",
"userMessage": "No resource was found at path /common/v1/activities/xc:20"
```

chapter 2

Overview of the system APIs in Cloud API

This topic provides an overview of the system APIs in the Cloud APIs. This includes a discussion of the base configuration APIs, the tools available for viewing API information, and the beta APIs.

The base configuration system APIs

The base configuration includes the following system APIs:

Name	Description	Path
Policy	API for policies and policy-specific objects	`/policy/v1`
Account	API for accounts and account-specific objects	`/account/v1`
Job	API for jobs and job-specific objects	`/job/v1`
Product Definition	API for PolicyCenter product definition metadata (This is a Beta API that is available only when enabled.)	`/productdefinition/v1`
Common	API for common InsuranceSuite platform objects like activities and notes	`/common/v1`
API List	Dynamically lists the APIs that are available	`/apis`

You can use the API path to view metadata about the API. This is discussed in detail in the following section.

There are also a minimal set of APIs for ContactManager. For more information, refer to the `<appURL>/rest/apis` endpoint for ContactManager.

Cloud API versions

> **Note:** The following section defines what a *minor release* is. Minor releases are not expected to have "breaking changes". The types of changes that do and do not fall into the definition of "breaking change" are described in the **Schema Backwards Compatibility Contract**. To access a copy of this contract, consult your Guidewire representative.

Every version of Cloud API has a version number. For example, suppose that there were four releases of the system APIs in January, April, July, and October of a given year. Each release could have the following version numbers:

- January: 1.0.0
- April: 1.1.0
- July: 1.2.0
- October: 2.0.0

Minor and major releases

In future releases, system API functionality is expected to change. To define and control these changes, the Cloud API makes use of minor versions and major versions.

- A *minor version* is a version of the Cloud API in which functionality is either identical to the previous release or additive.
- A *major version* is a version of the Cloud API in which functionality has changed from the previous release.

A given release of the Cloud API can have multiple versions of the APIs, some of which are minor and some of which are major.

Major versions are indicated by the "endpoint path number" in API endpoint paths. This is the number that appears after the "/v". (For example, in `/common/v1/activities`, the endpoint path number is 1.) When Guidewire makes a change to an API that is not purely additive, the changed API is considered a major release. Its endpoint number is incremented by 1.

When a release of the Cloud API includes a new major release, the previous minor release is also included. The minor release may be identical to the previous release, but it may also have additive changes.

For example, suppose that for the releases from the previous example:

- The Cloud API in the January release is major version 1.
- The Cloud API in the April release is identical or additive.
- The Cloud API in the July release is identical or additive.
- The Cloud API in the October release includes changes to existing functionality.

In this case, the January, April, and July releases would all include a single version of the APIs whose endpoint included "/v1". The October release would both the "/v1" set of APIs and a new "/v2" set of APIs. This is summarized in the following table.

Release Month	Version #	Compared to the previous release, this release...	Major versions in this release
January	1.0.0	...is identical or additive	/common/v1
April	1.0.1	...is identical or additive	/common/v1
July	1.0.2	...is identical or additive	/common/v1
October	2.0.0	...includes changes to existing functionality	/common/v1 (identical or additive to July's release) and /common/v2 (containing the changed functionality)

Viewing API definitions

An *API definition* is a technical description of the behavior of an API. The API definition typically includes the following:

- The endpoints in the API
- The schemas used by each endpoint, which dictate how payloads are structured
- The resources used by each endpoint
- The fields available in each resource
- The properties that apply to each field

These are the common approaches for viewing API definitions in Cloud API:

- Using Swagger UI
- Calling the `/openapi.json` endpoint (or the `/swagger.json`) through a request tool

Swagger UI

Swagger UI is an open source tool that presents API definitions as interactive web pages. For information on Swagger UI, refer to the Swagger web site: https://swagger.io/tools/swagger-ui/

Swagger UI is automatically bundled with PolicyCenter.

Swagger UI can be helpful when viewing schema information. Swagger UI presents this information hierarchically. Child schemas are indented with respect to parent schemas, and individual nodes of the hierarchy can be expended and collapsed. Searching through a complex schema is relatively straight-forward in Swagger UI.

However, Swagger UI strips out some of the metadata that is present in the source files. For example, Guidewire-proprietary tags are not rendered in Swagger UI. When you need access to all the metadata for an API, it may be better to call the `/openapi.json` endpoint directly.

> **Note:** Be aware that Swagger UI also has the capability to send requests to a working endpoint and observe responses. However, Guidewire recommends using Swagger UI only to view API definitions. The APIs in Cloud API are significantly robust. When sending requests using Swagger UI, the performance time for getting responses can be unacceptably long. For more information on recommended request tools, see “Requests and responses” on page 19.

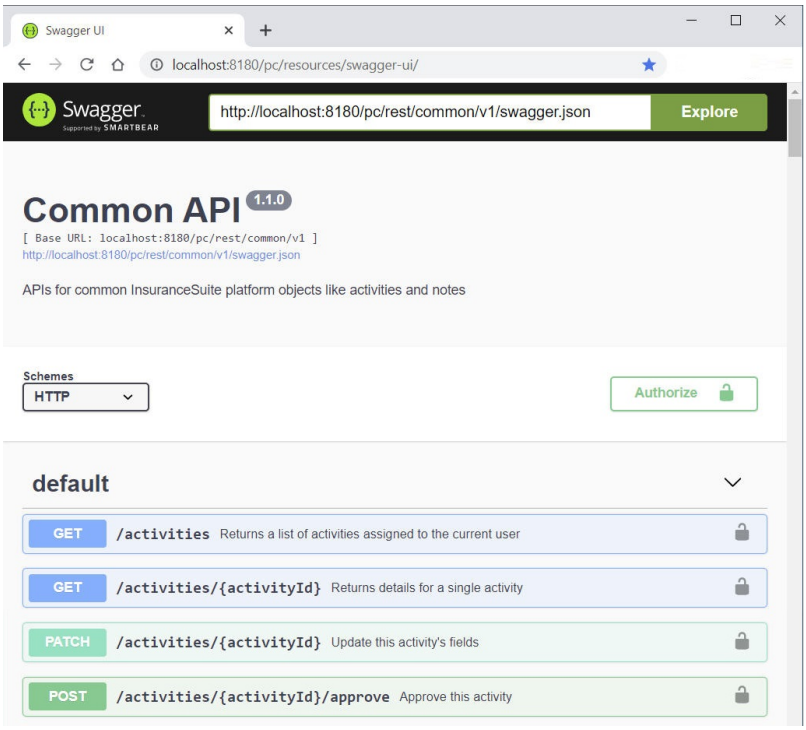
View an API definition using Swagger UI

Procedure

1. Identify the path for the API. (For a list of paths for each API, see “The base configuration system APIs” on page 23.)
2. Start PolicyCenter.
3. In a web browser, enter the URL for Swagger UI. This loads the Swagger UI tool.
   - The format of the URL is `<applicationURL>/resources/swagger-ui/`
   - For example, for a local instance of PolicyCenter, use: `http://localhost:8180/pc/resources/swagger-ui/`
4. In the text field at the top of the Swagger UI interface, enter the URL that points to the desired API's swagger.json file. Then, click **Explore**.
   - The format of the URL is `<applicationURL>/rest<APIpath>/swagger.json`.
   - For example, to view the common API, enter: `<applicationURL>/rest/common/v1/swagger.json`

Results

The following screenshot shows the top of the Swagger UI display of the Common API.



Organization of API information in Swagger UI

The Cloud API version number at the top in a gray bubble after the API name. (Note that individual APIs do not have distinct version numbers. The version numbers the appear in Swagger UI are for the entire Cloud API release.)

Every endpoint in the API appears in a list. For each API, the following information is shown by default:

- The endpoint's operation (such as GET)
- The endpoint's path (such as `/activities`)
- An endpoint summary (such as "Returns a list of activities assigned to the current user")

If you click the operation button, additional information about the endpoint appears. This includes:

- A more detailed endpoint description
- A list of query parameters supported by the endpoint
- A hierarchical list of resources and schemas used by the endpoint (This appears in the **Responses** section on the **Model** tab.)

The API definition endpoints and Postman

In some situations, it is useful to view the raw API definition information. In Cloud API, every API includes two endpoints that return the API definition: `/openapi.json` and `/swagger.json`.

- `/openapi.json` returns the API definition using the OpenAPI 3.0 specification, often referred to as "OpenAPI 3.0"
- `/swagger.json` returns the API definition using the Swagger 2.0 specification, often referred to as "Swagger 2.0"

  Note: Cloud API is built using the Swagger 2.0 Specification. However, the definition for each API can be returned in either the Swagger 2.0 specification (using the `/swagger.json` endpoint) or the OpenAPI 3.0 specification (using the `/openapi.json` endpoint).

The API definition endpoints can be helpful when you want to view all metadata about an endpoint, including metadata that Swagger UI might strip out. However, the API definition endpoints present information in a "raw" format. There is no use of color, font, or placement to help separate information. Schema hierarchies are not as readable as in Swagger UI. When you need to review a schema hierarchy in detail, it may be easier to use Swagger UI.

From a metadata perspective, the OpenAPI 3.0 specification is richer. So whenever either endpoint is an option, Guidewire recommends using the `/openapi.json` endpoint. For example, Guidewire-proprietary tags (such as `x-gw-typelist`) are listed in the `/openapi.json` response, but not in the `/swagger.json` response. However, some tools used to render API metadata may not be robust enough to process information using the OpenAPI 3.0 specification. The `/swagger.json` endpoint is available for these types of circumstances.

In the base configuration, the API definition endpoints are available to any caller, including unauthenticated callers.

Postman

You can call the API definition endpoints using a request tool. Request tools are not automatically bundled with InsuranceSuite applications. You must download and install them on your own.

Postman is a request tool that Guidewire recommends. This tool has the ability to:

- Save API calls, including headers and payloads
- Save collections of calls
- Automatically create a collection of calls for a schema's paths by importing the Swagger schema file
- Share collections with other developers on your team

For more information and to download the tool, see https://www.postman.com/.

View an API definition using Postman

Before you begin

Install Postman. For more information and to download the tool, see https://www.postman.com/.

About this task

This task does not involve authentication information. This is because every type of caller can request API metadata, including unauthenticated callers.

Procedure

1. Identify the path for the API. (For a list of paths for each API, see “The base configuration system APIs” on page 23.)
2. Start PolicyCenter.
3. Start Postman.
4. In Postman, start a new request by clicking the + tab to the right of the **Launchpad** tab.
5. Under the **Untitled Request** label, make sure that **GET** is selected. (This is the default operation.)
6. In the **Enter request URL** field, enter the following URL: `<applicationURL>/rest<APIpath>/openapi.json` (or `<applicationURL>/rest<APIpath>/swagger.json`). For example, to view the Common API on a local instance of PolicyCenter, enter the following:
   - http://localhost:8180/pc/rest/common/v1/openapi.json (OR http://localhost:8180/pc/rest/common/v1/swagger.json)
7. Click the **Send** button to the right of the request field.

Results

The API information appears in the results pane. For example, the following is the first part of the results when calling the previously referenced `openapi.json` endpoint:

```
{
    "components": {
        "parameters": {
            "activityId": {
                "description": "The REST identifier for the activity, as returned via previous requests that return a
list of activities\n",
```

```
                "in": "path",
                "name": "activityId",
                "required": true,
                "schema": {
                    "type": "string"
                }
            },
...
```

Organization of information in the output of API definition endpoints

The output of the API definition endpoints is "raw" JSON.

- General information about the API can be found in the `info` section.
- The list of endpoints can be found in the `paths` section.
  - If an endpoint path has multiple operations, the endpoint path is listed only once. Each operation appears under it.
  - For example, in the Common API, the `/activities/{activityId}` path has listings for GET and PATCH.
- Summaries and descriptions appear inline with the thing they summarize or describe.

Beta APIs

Published APIs and endpoints

In future releases, system API functionality is expected to change. To help insurers manage potential future changes, Guidewire maintains a *Schema Backwards Compatibility Contract*. This document identifies the rules for what Guidewire is allowed to change in an API minor or maintenance release while still having that release considered to be backwards compatible. To request a copy of the Schema Backwards Compatibility Contract, consult your Guidewire representative.

The Schema Backwards Compatibility Contract applies to published APIs and endpoints. These APIs and endpoints have been certified to be stable. By default, schema documentation resources (such as Swagger UI or the `/openapi.json` endpoints) return only published APIs and endpoints.

Beta APIs and endpoints

Each release of Guidewire Cloud API may include one or more beta APIs or beta endpoints. A *beta API* and a *beta endpoint* are an API or endpoint that is not yet covered by the Schema Backwards Compatibility Contract. These APIs and endpoints have not been certified as stable. They may change in the future in ways beyond what is covered by the Schema Backwards Compatibility Contract.

In the base configuration, beta APIs and endpoints:

- Are not enabled
- Are not returned by any schema documentation resources (such as Swagger UI or the `/openapi.json` endpoints)

Guidewire provides beta APIs and endpoints to help insurers with the development of system API functionality that may be available in the future. However, Guidewire recommends using beta APIs and endpoints with caution. They are not certified to be stable, and they are subject to change in future releases.

WARNING: Guidewire does not support the use of beta APIs or beta endpoints in production.

Enabling beta APIs and endpoints

Every beta API or endpoint is controlled by a toggle in the `config.properties` file. A *toggle* is an expression in `config.properties` that turns a feature on when the expression is set to true.

For some beta APIs and endpoints, the toggle is listed in `config.properties`, but it is set to `false`. For other beta APIs and endpoints, there is no toggle in `config.properties`. To enable beta APIs and endpoints, you must either set the

existing toggle to `true`, or add a toggle to `config.properties` and set it to `true`. After you modify `config.properties` in this way, you must restart the server.

For example, suppose that a release of Guidewire Cloud API included a fictitious set of beta endpoints that managed insurance metrics. For these endpoints, the following toggle appears in `config.properties`:

```
feature.InsuranceMetricsApisBeta = false
```

To enable these endpoints, you would need to change the line to the following, and then restart the server:

```
feature.InsuranceMetricsApisBeta = true
```

Identifying beta APIs

The beta APIs and endpoints for this release, if any, are listed in the following section.

Once enabled, beta APIs and endpoints appear in the output of the `/openapi.json` and `/swagger.json` endpoints. All beta APIs endpoints have the following attribute:

```
"x-gw-beta": true
```

For beta APIs, the `x-gw-beta` attribute is listed at the API level. The attribute does not appear at the endpoint level. All endpoints in a beta API are considered beta endpoints.

For beta endpoints in a published API, the `x-gw-beta` attribute is listed with each endpoint.

The `x-gw-beta` attribute appears only for APIs and endpoints that are beta. Published APIs and published endpoints do not have a listing of `"x-gw-beta": false`.

Once enabled, beta APIs also appear in Swagger UI. However, Swagger UI does not indicate whether a given API or endpoint is beta.

WARNING: Swagger UI does not render information from Guidewire proprietary tags, including the `x-gw-beta` tag. This means that, once you enable beta APIs, Swagger UI does not distinguish between published APIs and beta APIs. Guidewire strongly recommends that, if you enable beta APIs on a given development instance, alert all developers using this instance to the fact that beta APIs have been enabled. Without this alert, other developers may not be able to distinguish between published APIs and beta APIs.

Beta APIs for this release

This release contains no APIs that are entirely beta. It also contains no beta endpoints in any of the published APIs.

Guidewire reserves the right to add new beta APIs and endpoints to future releases.

Additional metadata endpoint functionality

Functionality for alternate API tools

Developers using the InsuranceSuite system APIs may want to interact with API metadata using tools other than Swagger UI or Postman. The following functionality may be useful when working with alternate tools. (Note that the `/swagger.json` endpoints do not support the following query parameters. They are supported only by the `/openapi.json` endpoints.)

Alternate options for rendering schemas

A *query parameter* is an expression added to the HTTP request that modifies the default response. The `/openapi.json` endpoints support the following query parameters, which can be used to change the way in which schema metadata is rendered.

- **filterByUser** - Whether to filter endpoints and schema properties by the authorization of this user.
  - Defaults to false
- **prettyPrint** - Whether to "pretty-print" the returned schema, making it larger but more human readable.
  - Defaults to false.

To add a query parameters to an HTTP request, use the following syntax:

```
?<parameterName>=<value>
```

To add additional query parameters to an HTTP request, use the following syntax for each query parameter after the first:

```
&<parameterName>=<value>
```

For example, the following HTTP request retrieves the metadata for the Common API. It also enables `filterByUser` and `prettyPrint`.

```
http://localhost:8180/pc/rest/common/v1/openapi.json?filterByUser=true&prettyPrint=true
```

Converting schema metadata into SDKs

Some tools, such as openapi-generator, provide the ability to consume a Swagger schema and output a Software Development Kit (SDK). The `/openapi.json` endpoints support the following query parameter, which can be used to change the way in which an SDK is rendered.

- **enablePolymorphism** - Whether to use the discriminator/oneOf pattern to output schemas in cases where the valid set of fields can vary based on some attribute of the data such as the country or subtype.
  - Defaults to true.
  - When set to false, the schema in these cases will contain the superset of all valid fields. For example, address schemas will contain all fields for all countries, rather than have separate schemas for different countries.
  - Setting this to false may make the schema output more consumable by tools that do not support that part of the OpenAPI schema.

To add a query parameters to an HTTP request, use the following syntax:

```
?<parameterName>=<value>
```

For example, the following HTTP request retrieves the metadata for the Common API. It also disables polymorphism.

http://localhost:8180/pc/rest/common/v1/openapi.json?enablePolymorphism=false

(For more information on openapi-generator, see https://github.com/OpenAPITools/openapi-generator/.)

The /typelists endpoints

The Common API contains two `/typelist` endpoints:

- `common/v1/typelists` - By default, this returns the names and descriptions of all typelists in PolicyCenter.
- `common/v1/typelists/{typelistName}` - By default, this returns the non-retired typecodes in the named typelist.

These endpoints can be useful when a caller application needs to retrieve typecode information from PolicyCenter. In the base configuration, these endpoints are available only to callers who have been authenticated.

Querying with typekey filters

Some typelists have a parent/child relationship. These typelists make use of typekey filters. A *typekey filter* is a mapping that identifies, for a typecode in one typelist, the valid values in a related typelist. For more information on typekey filters, refer to the *Application Guide*.

For example, the following typelists make use of typekey filters:

- `ActivityType` - The activity's broad type, such as General, Approval, or Assignment Review.
- `ActivityCategory` - An activity's specific category, such as Interview, Reminder, or Approval Denied.

If an activity's `ActivityType` is set to General, then some `ActivityCategory` values (such as Interview and Reminder) are valid, whereas others (such as Approval Denied) are not.

When using the `/typelists/{typelistName}` endpoint, if the typelist is associated with a typekey filter, you can use it to limit the response to only the typecodes that are valid when the parent typelist is set to a given typecode. The syntax for this is:

`/typelists/{typelistName}?typekeyFilter=category:cn:`*`relatedTypelist.Typecode`*

where:

- *`relatedTypelist`* is the name of the related typelist.
- *`Typecode`* is the typecode to use as a filter

For example, this call retrieves all typecodes in the `ActivityCategory` typelist:

GET `/common/v1/typelists/ActivityCategory`

However, this call retrieves only the typecodes in the `ActivityCategory` typelist that are valid when `ActivityType` is General:

GET `/common/v1/typelists/ActivityCategory?typekeyFilter=category:cn:ActivityType.general`

Including retired typecodes

By default, the `common/v1/typelists/{typelistName}` endpoint returns only non-retired typecodes. You can include retired typecodes by adding the following query parameter to the call:

`?includeRetired=true`

Tutorial: Query for typelist metadata

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

In this tutorial, you will query for all typecodes in the `VehicleType` typelist. You will then use a typekey filter to query for all vehicle types that are related to the personal auto policy line.

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `su` and password `gw`.
3. Enter the following call and click **Send**:
   - GET `http://localhost:8180/pc/rest/common/v1/typelists/VehicleType`
4. The response payload contains all non-retired vehicle types. Verify that the first three codes in the payload are: `Commercial`, `PP`, `PublicTransport`.)
5. Modify the call by adding the following query parameter to the end, and then click **Send**:
   - `?typekeyFilter=category:cn:PolicyLine.PersonalAutoLine`
6. The response payload now contains only vehicle types relevant to personal auto policies. Verify that there are only two codes in the payload now: `auto`, `other`. (`Commercial`, `PP`, and `PublicTransport` no longer appear because they are not valid vehicle types for a personal auto policy.)

Routing related API calls in clustered environments

To improve performance and reliability, you can install multiple PolicyCenter servers in a configuration known as a cluster. A PolicyCenter cluster distributes client connections among multiple PolicyCenter servers, reducing the load on any one server. If one server fails, the other servers seamlessly handle its traffic. For more information on clusters, refer to the *Administration Guide*.

When PolicyCenter is running in a cluster, it is possible for related system API calls to be routed to different nodes. This can cause problems, such as Concurrent Data Change Exceptions. Typically, multiple related system API calls need to be routed to the same node.

There are two ways that you can ensure a series of related system API calls are routed to the same instance: session IDs (which is the default behavior) and cookies.

Using session IDs

Within the context of system API calls in a clustered environment, a *session ID* is an arbitrary string generated by the caller application to identify related API calls. The ID is passed in the header of each request. Every request that uses a given session ID will be routed to the same node in the cluster. The header key for a session ID is `x-gwre-session`.

For example, suppose that a caller application makes the following calls to PolicyCenter cluster:

1. A POST to create an activity.
2. A PATCH to update the activity.
3. A POST to create a note on the activity.

All three calls include the following header:

```
x-gwre-session: 09d0fbf0-243c-4337-a582-725df8d33e39
```

Because all three calls specify the same session ID, all three calls will be routed to the same node.

Session IDs is the default behavior of the Guidewire Cloud Platform. If you wish to use this option, you do not need to make any special request to Guidewire Cloud Operations.

Using cookies

Within the context of system API calls in a clustered environment, a *cookie* is an arbitrary string generated by Guidewire Cloud Platform that can be used to identify subsequent related API calls.

If a request header does not contain an `x-gwre-session` header key, the Guidewire Cloud Platform generates a cookie and returns it in the response header with a `Set-Cookie` header key. Subsequent calls can include this cookie in the request header using the `Cookie` header key. Every request that uses a given cookie will be routed to the same node in the cluster that generated the cookie.

For example, suppose that a caller application makes the following calls to PolicyCenter cluster:

1. A POST to create an activity
   - The request header contains no `x-gwre-session` header key.
   - The response header contains the following: `Set-Cookie: gwre=ccd37ca0-f8d3-4a8e-b278-83274d82b355; Path=/`
2. A PATCH to update the activity.
   - The request header contains the following: `Cookie: gwre=ccd37ca0-f8d3-4a8e-b278-83274d82b355`
3. A POST to create a note on the activity
   - The request header contains the following: `Cookie: gwre=ccd37ca0-f8d3-4a8e-b278-83274d82b355`

Because the second and third call specify the cookie returned by the first call, the second and third call are routed to the same node that processed the first call.

Cookies are not the default behavior of the Guidewire Cloud Platform. If you wish to use this option, you must request this from Guidewire Cloud Operations.

Comparing session IDs and cookies

Under most circumstances, it may be easier to use session IDs.

- Session IDs are generated by the caller application.
- Session IDs do not require the caller application to identify information in a response header and then manage the storage that information for later use.

- Session IDs are the default behavior for Guidewire Cloud Platform.

If you wish to use cookies, you must request this from Guidewire Cloud Operations.

chapter 3

GETs and response payload structures

This topic discusses how the GET operation queries for data and the structure of the response payload that contains the query results. For information on how you can add query parameters to a GET to refine the query, see “Refining response payloads” on page 47.

If you want to interact directly with the concepts in this topic, go to the following tutorials:

- “Tutorial: Send a basic Postman request” on page 38
- “Tutorial: Send a Postman request with included resources” on page 43

Overview of GETs

A *GET* is an HTTP method that is used to retrieve data from an InsuranceSuite application.

In its simplest format, a GET consists of the GET operation and the endpoint, such as GET `/activities`. A GET can return either information about a single element (such as GET `/activities/{activityId}`) or information about a collection (such as GET `/activities/{activityId}/notes`).

The response to a GET includes:

- An HTTP response code indicating success or failure.
- A response payload that contains the data that was queried for.

When a developer configures a caller application to query information using a GET, the construction of the API call is fairly straight-forward. The call may require query parameters, but GETs do not require a request payload.

The majority of the work lies in parsing the response payload. The remainder of this chapter discusses how response payloads are structured and how developers can learn about response payload formats.

Standardizing payload structures

Communication between caller applications and system APIs is easier to manage when the information in the payloads follows a standard structure. The system APIs have standard structures for both request payloads and response payloads. The structures are defined by data envelopes, and by request and response schemas.

Standardizing information common to all endpoints

A *data envelope* is a wrapper that wraps JSON sent to or returned from the system APIs. To maintain a standard payload structure, the system APIs use two data envelopes: `DataEnvelope` and `DataListEnvelope`.

`DataEnvelope` is used to standardize the format of information for a single element. It specifies a `data` property with the following child properties:

- `checksum`
- `id`
- `links` (for a single element)
- `method`
- `refid`
- `related`
- `type`
- `uri`

The format of a payload for a single element looks like:

```
{
    "data": {
        "checksum": ...,
        "id":       ...,
        "links":    ...,
        "method":   ...,
        "refid":    ...,
        "related":  ...,
        "type":     ...,
        "uri":      ...
        }
    }
}
```

`DataListEnvelope` is used to standardize the format of information for collections. It specifies the following properties, which are siblings to the data section:

- `count`
- `links` (for a collection)
- `total`

The format of a payload for a collection looks like:

```
{
    "count"    ...,
    "data": [
        { properties_for_element_1 },
        { properties_for_element_2 },
        ...
    ],
    "links":   ...,
    "total":   ...
}
```

Every property does not appear in every payload. There are different reasons why a property may not appear in a given payload. For example:

- Some properties, such as `refid`, apply only to requests and do not appear in response payloads.
- Some properties, such as `count`, apply only to responses and do not appear in request payloads.
- Some properties, such as `related`, do not appear by default and appear only when the request includes certain query parameters.

Standardizing information specific to a given endpoint

`DataEnvelope` and `DataListEnvelope` provide a standard format for information that is applicable to all request and response payloads. But, different endpoints interact with different types of resources. For each endpoint, some portion of the payload must provide information about a specific type of resource.

To address this, the system APIs also use request schemas and response schemas. A *request schema* is a schema that is used to define the valid structure of a request payload for a specific set of endpoints. Similarly, a *response schema* is a schema that is used to define the valid structure of a response payload for a specific set of endpoints.

Request and response schemas are hierarchical. For example, for responses, the GET `/activity/{activityId}` endpoint uses the `ActivityResponse` schema. This schema has two child schemas: `ActivityData` and `ActivityResponseInclusions`.

Request and response schemas extend `DataEnvelope` or `DataListEnvelope`. This ensures that information relevant to all endpoints appears in payloads in a standard way.

Request and response schemas also define an `attributes` property for the payload. This property is associated with a schema that includes resource-specific information for the payload. For example, the GET `/activity/{activityId}` endpoint specifies an `attributes` property in the `ActivityData` child schema. This property is associated with the `Activity` schema, which contains activity-specific fields, such as `activityPattern` and `activityType`. As a result, response payloads for the GET `/activity/{activityId}` endpoint have this structure:

```
{
    "data": {
        "checksum": ...,
        "attributes" : {
            "activityPattern": ... ,
            "activityType": ...,
            ...},
        "id":         ...,
        "links":      ...,
        "method":     ...,
        "refid":      ...,
        "related":    ...,
        "type":       ...,
        "uri":        ...
        }
    }
}
```

Viewing response schemas

You can use Swagger UI to review the structure of a response payload for a given endpoint. This includes the hierarchy of response schemas and the type of information in each schema. The information appears in each endpoint's **Responses** section on the **Model** tab.

View a response schema in Swagger UI

Procedure

1. Start PolicyCenter.
2. In a web browser, navigate to the Swagger UI for the appropriate API.
   - For more information, see “View an API definition using Swagger UI” on page 25.
3. Click the operation button for the appropriate endpoint. Swagger UI shows details about that endpoint underneath the endpoint name.
   - For example, to view the response schema for GET `/activities/{activityID}`, click the GET button for that endpoint.
4. Scroll down to the **Responses** section. The **Model** tab shows the hierarchy of schemas for this endpoint, and the contents defined by each schema.

Sending GETs

You can use a request tool, such as Postman, to ensure GETs are well-formed and to review the structure of the response payloads. For more information, see “Requests and responses” on page 19.

Send a GET using Postman

Procedure

1. Start PolicyCenter and Postman.
2. Start a new request by clicking the + to the right of the **Launchpad** tab.
3. Under the **Untitled Request** label, make sure that GET is selected. (This is the default operation.)

4. In the **Enter request URL** field, enter the URL for the server and the endpoint.
   - For example, to do a GET `/activities` on an instance of PolicyCenter on your machine, enter: `http://localhost:8180/pc/rest/common/v1/activities`
5. On the **Authorization** tab, provide sufficient authorization information to execute the request. For example, to set up basic authentication for `aapplegate`:
   a) Click the **Authorization** tab.
   b) For the **Type** drop-down list, select *Basic Auth*.
   c) In the **Username** field, enter `aapplegate`.
   d) In the **Password** field, enter `gw`.
6. Click the **Send** button to the right of the request field.

Tutorial: Send a basic Postman request

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see "Tutorial: Set up your Postman environment" on page 21.

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Enter the following call and click **Send**: GET `http://localhost:8180/pc/rest/common/v1/activities`

Checking your work

Once a response has been received, its payload is shown in the lower portion of the Postman interface. The first few lines of the response payload are:

```
{
    "count": 13,
        "data": [
            {
            "attributes": {
            "activityPattern": "uw_review_contingency",
            "activityType": {
                "code": "general",
                "name": "General"
            },
            "assignedByUser": {
                "displayName": "Alice Applegate",
                "id": "pc:105"
            },
```

Payload structure for a basic response

The following sections describe the response payload for a basic response. For the purpose of this discussion, a *basic response* is a response that contains information about a specific element or collection, but does not include any included resources. Included resources are discussed in "Payload structure for a response with included resources" on page 42.

Examples of response payloads in Postman

You can use the following Postman calls to load examples of response payloads. All of these calls assume the following:

- Your instance of PolicyCenter is installed on your local machine.
- The Large sample data has been loaded.
- The call uses basic authentication with user `aapplegate` and password `gw`.

Response payload examples

Response payload for a single resource:

1. Activity "Verify coverage" (whose Public ID is pc:101)
   - GET `http://localhost:8180/pc/rest/common/v1/activities/pc:101`
2. Policy 0596932432 (whose Public ID is pc:1)
   - GET `http://localhost:8180/pc/rest/policy/v1/policies/pc:1`

Response payload for a collection:

1. All activities assigned to Alice Applegate
   - GET `http://localhost:8180/pc/rest/common/v1/activities`
2. All policies assigned to Alice Applegate
   - GET `http://localhost:8180/pc/rest/policy/v1/policies`

Structure of a basic response

The high-level structure of a basic response is shown below. The first and last properties (`count` and collection-level `links`) are used only for collection payloads. All other properties are used for both element and collection payloads.

(Note: JSON does not support comments. However, to clarify the code, pseudo-comments have been added. Each pseudo-comment is preceded by a hashtag (#).)

```
{
      "count": N,                     # Number of resources in collection*
      "data": [                       # List of resources
          {                           # Resource 1 begins here
          "attributes": {             # Resource 1 name/value pairs
              "propertyName": "propertyValue",
              ... },
          "checksum": "val",          # Resource 1 checksum value
          "links": { ... }            # Resource 1 links
          },                          # Resource 1 ends here
          {                           # Resource 2 begins here
          "attributes": {             # Resource 2 name/value pairs
              "propertyName": "propertyValue",
              ... },
          "checksum": "val",          # Resource 2 checksum value
          "links": { ... }            # Resource 2 links
          },                          # Resource 2 ends here
      ... ],                          # Resources 3 to N
      "links": { ... }                # Collection-level links*
}
                                      # *-used only for collection responses
```

The count property

The `count` property identifies the number of elements returned in the payload. It is used only in responses that contain collections.

The data section

The `data` section contains information about the resources returned by the endpoint. For each resource, the following subsections appear by default:

- **attributes** - A set of name/value pairs for the fields of each resource.
- **checksum** - A checksum value for each resource.
- **links** - HTTP links that can be used to take action on each resource.

If an endpoint returns a single resource, the data section has a single set of `attributes`, `checksum`, and `links`. If an endpoint returns a collection, the data section has one set of `attributes`, `checksum`, and `links` for each resource.

The attributes section

The `attributes` subsection lists the fields returned for a resource, and the values for those fields. For example:

```
"attributes": {
    "activityPattern": "check_coverage",
    "activityType": {
        "code": "general",
        "name": "General"
    },
    ...
},
```

Each resource has a default set of fields that are returned. This is typically a subset of all the fields that could be returned. You can override the default set of fields returned using the `fields` query parameter. For more information, see “Specifying which fields to GET” on page 50.

Simple values

When a field is a scalar, its value is listed after the colon. For example:

```
"subject": "Verify which coverage is appropriate"
```

ID properties

Every resource has an `id` field. This has the same value as the Public ID of the object in PolicyCenter. This is typically one of the fields returned by default. For example:

```
"id": "xc:20",
```

This value is also used in an endpoint that names a specific element, such as:

```
GET /activities/xc:20/notes
```

Date and datetime values

Date and datetime values appear in payloads as a string with the following format:

- Datetime: `YYYY-MM-DDThh:mm:ss.fffZ`
- Date: `YYYY-MM-DD`

where:

- `YYYY` is the year.
- `MM` is the month.
- `DD` is the day.
- For datetime values:
  - `T` is a literal value that separates the date portion and the time portion.
  - `hh` is the hour.
  - `mm` is the minute.
  - `ss` is the second.
  - `fff` is the second fraction.
  - `Z` is a literal value that means "zero hour offset". It is also known as "Zulu time" (UTC).

For example:

`"dueDate": "2020-03-23T07:00:00.000Z",`

Be aware that, for some fields, the system APIs require a date value as an input. But, PolicyCenter stores the value as a datetime value, and therefore the value in response payloads is a datetime value. The system APIs behave this way in an attempt to closely model the PolicyCenter behavior of adding a time value to entered date values. For example, the `JobEffectiveDate` field is specified in the `CreateSubmissionAttributes` schema as a date value. But once the submission has been created, the `JobEffectiveDate` field in the response payload will have a datetime value.

Inlined resources

Some response payloads contain inlined resources. An *inlined resource* is a resource that is not the root resource, but some of its fields are listed in the attributes section by default along with fields from the root resource. Inlined resources follow the format:

```
"inlinedResourceName": {
    "inlinedResourceField1": value,
    "inlinedResourceField2": value,
    ...
},
```

Inlined resources are declared in the response schema. Similar to scalar values, you can control which inlined resources and inlined resource fields are returned in a response by using the `fields` query parameter. For more information, see “Specifying which fields to GET” on page 50.

Broadly speaking, there are four types of inlined resources: typelists, currency amount values, simple references, and complex references.

Typelists are listed with a code field and a name field. They use the `TypeCodeReferences` schema. For example:

```
"priority": {
    "code": "urgent",
    "name": "Urgent"
},
```

Currency amount values have a currency field and an amount field. For example:

```
"transactionAmount": {
    "amount": "500.00",
    "currency": "usd"
```

Simple references are references to a related object that use the `SimpleReferences` schema. This schema includes only the following fields: `displayName`, `id`, `refId`, `type`, and `uri`. By default, most endpoints return only `displayName` and `id`.

For example, in the following snippet, `assignedGroup` and `assignedUser` are simple references:

```
"assignedGroup": {
    "displayName": "Auto1 - TeamA",
    "id": "demo_sample:31"
},
"assignedUser": {
    "displayName": "Andy Applegate",
    "id": "demo_sample:1"
},
```

Complex references are references to a related object that uses a schema more complex than the `SimpleReferences` schema. For example, when a contact's primary address is added to a response payload, it uses the `Address` schema, which includes a larger number of fields.

Fields with null values are omitted

Response payloads contain only fields whose values are non-NULL. Fields with NULL values are omitted from the response payload.

If a given field is expected in a response payload but it is missing, this is often because the value was NULL.

The checksum field

The `checksum` field lists a value that identifies the "version" of a resource. Whenever a resource is modified at the database level, it is assigned a new checksum value. Processes that modify data can use checksums to verify that a resource has not been modified by some other process in between the time the resource was read and the time the resource is to be modified.

For more information, see “Lost updates and checksums” on page 101.

The links subsection (for an element)

The `links` subsection of the `data` section lists paths that identify actions that can be taken on the specific element, if any. Each link has a name, an `href` property, and a list of methods. Caller applications can use these links to construct HTTP requests for additional actions to take on that resource.

For example, suppose that a given caller application gets activity xc:20. This application has sufficient permission to assign this activity and to view the notes associated with this activity. The following would appear in the links section for activity xc:20:

```
"links": {
    "assign": {
        "href": "/common/v1/activities/xc:20/assign",
        "methods": [
            "post"
        ]

    },
    "notes": {
        "href": "/common/v1/activities/xc:20/notes",
        "methods": [
            "get",
            "post"
        ]
    },
    "self": {
        "href": "/common/v1/activities/xc:20",
        "methods": [
            "get"
        ]
    }
}
```

The `self` link is a link to the resource itself. The `self` link is useful when a caller application receives a list of resources and wants to navigate to a specific resource in the list.

For a given object, links that execute business actions appear only if the action makes sense given the state of the object, and only if the caller has sufficient permission to execute the action. For example, the link to close an activity will not appear if the activity is already closed. Similarly, the link to assign an activity will not appear if the caller lacks permission to assign activities.

The collection-level links section

If a response contains a collection, there is a `links` section at the end of the payload. This section is a sibling of the `data` section. It contains links that are relevant to the entire collection, such as the `prev` and `next` links that let you page through a large set of resources.

Payload structure for a response with included resources

Some endpoints support the ability to query for a given type of resource and for resource types related to that type. For example, by default, the GET `/activities` endpoint returns only activity resources. However, you can use the `include` query parameter to include any notes related to the returned activities in the response payload. These types of resources are referred to as *included resources*. The technique of adding included resources to a GET is sometimes referred to as *response inclusion* or *read inclusion*.

The syntax for adding included resources is:

```
?include=<resourceName>
```

For example GET `/activities?include=notes` returns all activities assigned to the current user, and all notes associated with those activities.

You can include multiple resource types in a single query. To do this, identify the resources in a comma-delimited list. For example, GET `/policies?include=contacts,locations` returns all policies assigned to the current user, and all contacts and locations associated with those policies.

When you execute a call with `include`, the response payload contains information about the primary resources and the included resources. However, most of the information about the included resources do not appear inline with the primary resources. Rather:

- Every primary resource has a `related` section. This section lists the IDs (and types) of included resources related to that resource. However, each `related` section does not include any other details about those resources.
- Details about the included resources appear at the end of the payload in a section called `included`.

The IDs of included objects appear in both the `related` section and the `included` section. You can use these IDs to match a primary resource with details about its included resources.

Contrasting included resources and inlined resources

A response payload can contain two types of resources that have a relationship to the root resources: inlined resource and included resources. The following table contrasts the two types of resources.

Resource type	How many related resources for each primary resource?	Where do their fields appear?	When do they appear?
Inlined resource	Typically one. (For example, every activity has only one related `assignedUser`.)	Entirely in the `attributes` section of the root resource	If the query does not use the `fields` query parameter, then each inlined resource appears only if it is one of the default attributes. If the query does use the `fields` query parameter, then each inlined resource does or does not appear based on whether it is specified in that query parameter.
Included resource	One to many. (For example, every `activity` can have several related `notes`.)	IDs appear in the `related` section of the root resource. The remaining attributes appear in the `included` section at the bottom of the payload.	When the query parameter includes the `?include=resourceName` query parameter

Tutorial: Send a Postman request with included resources

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. To get a response payload for Policy 0596932432 (Public ID pc:1) and its contacts, enter the following and click **Send**:

   GET `http://localhost:8180/pc/rest/policy/v1/policies/pc:1?include=contacts`

   In the response payload:

   - In the data section, line 122 identifies a related contact with the ID of test_pp:2
   - In the related section, lines 132 to 185 provides the details about contact test_pp:2. (The ID is in the related section on line 152.)
4. To get a response payload for Policy 0596932432 (Public ID pc:1) and its primary insured, enter the following and click **Send**:

   GET `http://localhost:8180/pc/rest/policy/v1/policies/pc:1?include=primaryInsured`

   In the response payload:

   - In the data section, line 122 identifies a related contact with the ID of test_pp:2
   - In the related section, lines 132 to 185 provides the details about contact test_pp:2. (The ID is in the related section on line 152.)

Structure of a response with included resources

The high-level structure of a response with included resources is shown below. Information that pertains specifically to included resources appears in bold. (Note: JSON does not support comments. However, to clarify the code, pseudo-comments have been added. Each pseudo-comment is preceded by a hashtag (#).)

```
{
    "count": N,                         # Number of resources is payload
    "data": [                           # Details for each resource
        {                               # Resource 1 begins here
            "attributes": {             # Resource 1 name/value pairs
                "propertyName": "propertyValue",
                ... },
            "checksum": "val",          # Resource 1 checksum value
            "links": {                  # Links relevant to Resource 1
                ... },
            "related": {               # List of resources related to R1
                "resourceType": {       # Related resource type
                    "count": NN,        # Number of related resources for R1
                    "data": [
                        {               # First resource related to R1 starts
                            "id": "relatedResourceID",
                            "type": "resourceType"
                        },              # First resource related to R1 ends
                        ...             # Other resources related to R1
                    ] } }
        },                              # Resource 1 ends here
        {                               # Resource 2 begins here
            "attributes": {             # Recourse 2 name/value pairs
                "propertyName": "propertyValue",
                ... },
            "checksum": "val",          # Resource 2 checksum value
            "links": {                  # Links relevant to Resource 2
                ... },
            "related": {                # List of resources related to R2
                "resourceType": {       # Related resource type
                    "count": NN,        # Number of related resources for R2
                    "data": [
                        {               # First resource related to R2 starts
                            "id": "relatedResourceID",
                            "type": "resourceType"
                        },              # First resource related to R2 ends
                        ...             # Other resources related to R2
                    ] } }
        },                              # Resource 2 ends here
        ... ],                          # Resources 3 to N
    "links": {                          # Links relevant to collection
          ...
        },
    "included": {                       # List of related resources
        "resourceType": [               # First related resource type
            {
                "attributes": {         # Related resource 1 start
                    ...                 # Related resource 1 name/value pairs
                    "id": " relatedResourceID ",
                    ... },
                "checksum": "0",        # Related resource 1 checksum value
                "links": { ... }        # Links relevant to Related resource 1
            },
            ...                         # Related resources 2 to end
    ] }
}
```

The related section (for a resource)

For every resource, there is an additional `related` section that identifies:

- The number of included resources, and
- The IDs of the included resources

For example, the following code snippet is from the response for a query for all activities and related notes. Activity xc:44 has one included note, whose ID is xc:55.

```
{
    "attributes": {
        ...
        "id": "xc:44",
        ...
        "subject": "Check coverage"
    },
    "checksum": "2",
```

```
    "links": {
        ...
    },
    "related": {
        "notes": {
            "count": 1,
            "data": [
                {
                    "id": "xc:55",
                    "type": "Note"
                }
            ]
         }
      }
},
```

If a GET uses the `included` query parameter, but no related resources exist, the `related` section still appears. But, the count is 0 and the data section is empty. For example:

```
"related": {
    "notes": {
        "count": 0,
        "data": []
    }
}
```

If a GET omits the `included` query parameter, the related section is omitted from the response payload.

The included section (for a response)

For every response, there is an `included` section that appears at the end of the response payload. It lists details about every included resource for the primary resources.

For example, the following code snippet is from the `included` section from the previous example.

```
"included": {
    "Note": [
        {
            "attributes": {
                "author": {
                    "displayName": "Betty Baker",
                    "id": "demo_sample:8"
                },
                "bodySummary": "Main contact is on vacation 03/20",
                "confidential": false,
                "createdDate": "2020-03-30T23:11:33.346Z",
                "id": "xc:55",
                "securityType": {
                    "code": "unrestricted",
                    "name": "Unrestricted"
                },
                "subject": "Main contact is on vacation 03/20",
                "topic": {
                    "code": "general",
                    "name": "General"
                },
                "updateTime": "2020-03-30T23:12:08.892Z"
            },
            "checksum": "0",
            "links": {
                "self": {
                    "href": "/common/v1/notes/xc:55",
                    "methods": [
                        "get",
                        "patch"
                    ]
                }
            }
        }
    ]
},
```

Recall that activity xc:44 has one included note. The included note's ID is xc:55. The note shown in the `included` section is the note related to activity xc:44.

Including either a collection or a specific resource

For a given endpoint, some of the `include` options return a collection of resources for each primary resource. Other `include` options return a single resource for each primary resource.

An example of the first case is GET `/policies/{policyId}?include=contacts`. This call returns the policy with the given Policy ID and all contacts related to that policy. There are theoretically several related resources (contacts) for each primary resource (the policy with the given Policy ID).

An example of the second case is GET `/policies/{policyId}?include=primaryInsured`. This call returns the policy with the given Policy ID and the contact who is designated as the primary insured. There is only a single related resources (the contact who is the primary insured) for each primary resource (the policy with the given Policy ID).

You can also specify multiple include options, as in GET `/policies/{policyId}?include=contacts,primaryInsured`. In this case, for each policy, the related section specifies the IDs of all related contacts and it also explicitly identifies the ID of the primary insured.

As a general rule, if an include option is named using a plural, it returns a set of resources for each primary resource. If an include option is named using a singular, it returns a single resource for each primary resource.

Determining which resources can be included

For each endpoint, you can determine the resources that can be included by referring to the Swagger UI model for the endpoint. There will be a data envelope in the model whose name ends with `...Inclusions`. This data envelope lists all the resources that can be included when querying for that type of resource.

For example, in the Common API, the model for GET `/activities` references an `ActivityResponseInclusions` element. This element has a single child element - `Note`. This means that the only type of element you can include on an activity query is notes.

If you attempt to include a resource type that a given endpoint does not support for inclusion, the API returns a 400 error code and message. For example, the following is the message if you attempt to do a GET `/activities?include=users`:

```
"userMessage": "Bad value for the 'include' query parameter - The requested
inclusions '[users]' are not valid for this resource. The valid options are [notes]."
```

chapter 4

Refining response payloads

This topic discusses how to use query parameters to refine a request to customize the response payload. This is most often done with GETs, but query parameters can also be used with POSTs and PATCHes.

If you want to interact directly with the concepts in this topic, go to the following tutorials:

- “Tutorial: Send a GET with the filter parameters” on page 50
- “Tutorial: Send a GET with the fields parameter” on page 53
- “Tutorial: Send a GET with the sort query parameter” on page 54
- “Tutorial: Send a GET with the pageSize and totalCount parameters” on page 57
- “Tutorial: Send a GET with query parameters for included resources” on page 59

Overview of query parameters

When you execute a system API call using only the endpoint (as in GET `/activities`), the response payload has a default set of resources and a default structure.

You may want to refine the response payload beyond the default behavior by:

- Specifying a custom set of properties.
- Filtering out resources that do not meet a given criteria
- Sorting the resources
- Limiting the number of elements returned in each payload
- Retrieving a count of the total number of resources in the database that meet the query's criteria

You can refine the response payload using query parameters. A *query parameter* is an expression added to the HTTP request that modifies the default response payload.

A system API call can include any number of query parameters. The list of query parameters starts with a question mark (?). If there are multiple query parameters, each is separated by an ampersand (&). For example:

- GET `/activities?fields=*all`
- GET `/activities?filter=escalated:eq:false`
- GET `/activities?fields=*all&filter=escalated:eq:false`

Included resources

You can use the `include` query parameter to include resources related to the primary resources of the response. You can also use query parameters to specify a custom set of properties for included resources, filter out included resources that do not meet a given criteria, sort the included resources, and so on. For more information, see “Using query parameters on included resources” on page 57.

Viewing query parameter documentation in Swagger UI

For every endpoint, Swagger UI provides descriptions of the query parameters supported by that endpoint. This information is hidden by default. To show the descriptions, click the endpoint's operation button (such as the **GET** button for GET `/activities`). The query parameter descriptions appear under the endpoint.

Parameter definitions

The **Parameters** section describes each query parameter.

Supported parameters

The **Responses** section include a **Model** tab. This tab provides information about the fields that support particular query parameters. For example, you can sort results on some fields, but not all of them. The fields that support sorting appear in the model with the text `"sortable": true`.

Query parameter error messages

If you attempt to use a query parameter on a field that does not support that parameter, the system API returns a 400 Bad Request error and an error message. For example, if you execute: GET `/activities?sort=escalationDate`, the system API provides the following error message:

```
"message": "The sort column 'escalationDate' is not a valid option. The valid
      sort options are [assignedUser, dueDate, escalated, priority, status, subject],
      optionally prefixed with '-' to indicate a descending sort."
```

Specifying the resources and fields to return

You can use query parameters to:

- Specify filtering criteria to narrow which resources are returned
- Specify which fields you want returned for the resources that are returned

Filtering GETs

You can narrow which resources are returned using the `filter` keyword followed by one or more criteria. The criteria are specified using the following syntax:

```
?filter=field:op:value
```

where:

- *field* is the name of a filterable field
- *op* is one of the comparison operators from the following table
- *value* is the value to compare

op value	Meaning	Example using the GET /activities endpoint	Returns activities where...
`eq`	Equal	`?filter=escalated:eq:true`	...the `escalated` field equals true
`ne`	Not equal	`?filter=escalated:ne:true`	...the `escalated` field equals false
`lt`	Less than	`?filter=dueDate:lt:2020-05-11T07::00::00.000Z`	...the due date is less than (before) May 11, 2020.
`gt`	Greater than	`?filter=dueDate:gt:2020-05-11T07::00::00.000Z`	...the due date is greater than (after) May 11, 2020.
`le`	Less than or equal	`?filter=dueDate:le:2020-05-11T07::00::00.000Z`	...the due date is less than or equal to (on or before) May 11, 2020.

ge	Greater than or equal	?filter=dueDate:ge: 2020-05-11T07::00::00.000Z	...the due date is greater than or equal to (on or after) May 11, 2020.
in	In	?filter=priority:in:urgent,high	...the priority is either urgent or high
ni	Not in	?filter=priority:ni:urgent,high	...the priority is neither urgent nor high
sw	Starts with	?filter=subject:sw:Contact %20claimant	...the subject starts with the string "Contact claimant"
cn	Contains	?filter=subject:cn:Contact %20claimant	...the subject contains the string "Contact claimant"

The query parameter is passed as part of a URL. Therefore, special conventions must be used for certain types of values to ensure the URL can be parsed correctly.

- Specify strings without surrounding quotes.
- If a string has a space in it, use the URL encoding for a space (%20). (For example, "subject starts with 'Contact claimant'" is specified as `?filter=subject:sw:Contact%20claimant`)
- If a string has a colon (:) in it, use two colons (::) in the URL. The first colon acts as an escape character. (For example, "subject starts with 'Urgent: Information needed'" is specified as `?filter=subject:sw:Urgent::Information%20needed`)
- Specify Booleans as either true or false. (For example, "escalated is true" is specified as `?filter=escalated:eq:true`)
- Date and datetime fields must be specified as an ISO-8601 datetime value. All datetime fields can accept either date values or datetime values. For datetime values, the colons in the value must be expressed as "::". For example, "due date is less than 2020-04-03T15:00:00.000Z" is specified as `?filter=dueDate:lt:2020-05-11T07::00::00.000Z`.

References to typekey fields automatically resolve to the field's code. For example, to filter on activities whose priority is set to `urgent`, use: GET `/activities?filter=priority:eq:urgent`.

You can also use the filter query for related resources added through the `include` parameter. For more information, see “Using query parameters on included resources” on page 57.

Determining which values you can filter on

For a given endpoint, you can identify the attributes that are filterable by reviewing the endpoint **Model** in Swagger UI. If a field is filterable, then the schema description of the field includes the text: `"filterable": true`.

For example, the following is the schema description for two fields returned by the Common API's `/activities` endpoint.

```
escalated       boolean
                readOnly: true
                x-gw-extensions: OrderedMap { "filterable": true, "sortable": true }
escalationDate  string($date-time)
                x-gw-nullable: true
```

Note that the `escalated` field includes the `"filterable": true` expression, but the `escalationDate` field does not. This means that you can filter on `escalated`, but not `escalationDate`.

When querying for a specific root resource, do not filter on the id property

When writing a call to GET a single root resource, use an endpoint that returns a single element, such as:

GET `/activities/xc:20`

Do not attempt to retrieve the resource by using an endpoint that returns a collection and filtering on the `id` property, as in:

GET `/activities?filter=id:eq:xc::20`

Typically, the latter approach does not work because collection endpoints do not have filterable `id` properties.

This restriction applies to the root resource only. There may be situations where you need to retrieve a root resource and one or more child resources. When retrieving the child resources, it may be possible and appropriate to filter on the child resource's ID.

Filtering on multiple values

You can include multiple filter criteria. To do this, each criteria requires its own filter expression. Separate the expressions with an ampersand (&). The syntax is:

```
?filter=field:op:value&filter=field:op:value
```

When multiple criteria are specified, the criteria are ANDed together. Resources must meet all criteria to be included in the response. For example, the following GET returns only high priority activities that have not been escalated.

GET `/activities?filter=priority:eq:high&filter=escalated:eq:false`

Endpoints with default filters

Some endpoints have default filters. For example, the `/accounts/{AccountID}/jobs` endpoint has a default filter that returns only open jobs.

You can identify whether an endpoint has a default filter by checking the endpoint summary in Swagger UI. The summary is visible after you click the GET button in Swagger UI for the given endpoint.

For example, the summary for the `/accounts/{AccountID}/jobs` endpoint says: "Retrieve a list of jobs associated with this account. By default this will include only open jobs."

You can use the `filter` query parameter to override default filters. For example, the following GET returns all jobs without regards to the status: GET `/accounts/{AccountID}/jobs?filter=jobFilter:eq:all`

If you add a `filter` query parameter on an endpoint with a default filter, the default filter is discarded. If you want the response payload to reflect both the default filter and a custom filter, you must specify both explicitly.

Tutorial: Send a GET with the filter parameters

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Enter the following and click **Send**:
   GET http://localhost:8180/pc/rest/common/v1/activities
4. Open a second request tab and specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
5. Enter the following and click **Send**:
   GET http://localhost:8180/pc/rest/common/v1/activities?filter=priority:eq:high

Checking your work

Compare the two payloads. Note that the first response payload includes all activities, whereas the second response payload contains only the activities with a high priority.

Specifying which fields to GET

Every endpoint returns a default set of fields. You can override this default set using the `fields` parameter. This is useful when you need properties not returned by default, or when you want to avoid getting properties that are not necessary. (You can also use the `fields` query parameter for related resources added through the `include` parameter. For more information, see “Using query parameters on included resources” on page 57.)

The `fields` parameter can be set to one or more of the following values:

- `*all` - Return all fields

  Note: `*all` is not recommended for production environments as it returns fields that cost additional resources trimmed from default response payloads; instead, use the `*default` filter with any specific fields that do not appear in the default detail or summary list.
- `*default` - Return the default fields (typically used in conjunction with an additional field list)
- *`field_list`* - Return one or more fields specified as a comma-delimited list

The set of fields returned by default

For endpoints that return a single element, the default fields to return are defined in a "detail" list. Similarly, for endpoints that return a collection, the default fields to return are defined in a "summary" list.

For example, the following list compares the detail fields for a contact resource (for example, the default fields for the `/accounts/{accountId}/contacts/{contactId}` endpoint) and the summary fields returned for a contact collection (for example, the default fields for the `/accounts/{accountId}/contacts` endpoint). Fields included in the Detail only are in bold:

- Detail: `companyName`, `contactSubtype`, `displayName`, `emailAddress1`, **`emailAddress2`**, `id`, `industryCode`, **`primaryPhoneType`**, `subtype`, `taxId`
- Summary: `companyName`, `contactSubtype`, `displayName`, `emailAddress1`, `id`, `industryCode`, `subtype`, `taxId`

The `fields` parameter has three options related to these default sets:

- `*detail` - Returns the fields in the detail list
- `*summary` - Returns the fields in the summary list
- `*default` - Returns the fields in the detail list (if the endpoint returns a single element) or the fields in the summary list (if the endpoint returns a collection)

For endpoints that return a single element:

- `?fields=*default` and `?fields=*detail` are logically equivalent.
- You can override the default behavior by using `?fields=*summary`, which returns the summary fields instead of the detail fields.

For endpoints that return a collection:

- `?fields=*default` and `?fields=*summary` are logically equivalent.
- You can override the default behavior by using `?fields=*detail`, which returns the detail fields instead of the summary fields.

Some API calls need a set of fields that is not exactly equivalent to either the detail list or the summary list. These calls can name specific fields, either on their own or in addition to a default list of fields. They can also specify all fields.

Returning the default properties plus additional specific properties

To return the default fields of an endpoint with an additional set of fields, use:

```
?fields=*default,field_list
```

where *`field_list`* is a comma-delimited list of fields.

For example, the following query returns all default fields for activity xc:20 as well as the description and the start date.

GET `/activities/xc:20?fields=*default,description,startDate`

Returning a specific set of properties

To return a specific set of fields, use:

```
?fields=field_list
```

where *`field_list`* is a comma-delimited list of fields.

For example, the following query returns only the description and the start date for activity xc:20:

GET `/activities/xc:20?fields=description,startDate`

Returning a specific set of properties on inlined resources

Some response payloads include inlined resources in the `attributes` section. For example, the following is a snippet of the response for a GET `/activities`. This payload contains two inline resources, `assignedGroup` and `assignedUser`.

```
"attributes": {
    "activityPattern": "contact_claimant",
    "assignedGroup": {
        "displayName": "Auto1 - TeamA",
        "id": "demo_sample:31"
    },
    "assignedUser": {
        "displayName": "Andy Applegate",
        "id": "demo_sample:1"
    },
    "closeDate": "2020-04-06T07:00:00.000Z",
    "dueDate": "2020-04-06T07:00:00.000Z",
    "escalated": false,
    "id": "cc:20",
    ...
}
```

You can use the `fields` query parameter to specify an inlined resource. When you do, all default fields for that resource are returned. For example, you could specify that you want a GET `/activities` to return only the `assignedGroup` and `assignedUser` fields (and all of their default subfields) using the following:

GET `/activities?fields=assignedGroup,assignedUser`

This would return:

```
"attributes": {
    "assignedGroup": {
        "displayName": "Auto1 - TeamA",
        "id": "demo_sample:31"
    },
    "assignedUser": {
        "displayName": "Andy Applegate",
        "id": "demo_sample:1"
        }
}
```

You can also specify specific subfields using the following syntax:

`?fields=`*`inlinedResourceName.fieldName`*

For example, you could specify that you want a GET `/activities` to return only the IDs of the assigned user and group using the following:

GET `/activities?fields=assignedGroup.id,assignedUser.id`

This would return:

```
"attributes": {
    "assignedGroup": {
        "id": "demo_sample:31"
    },
    "assignedUser": {
        "id": "demo_sample:1"
        }
}
```

Returning all properties

To return all of the fields that an endpoint is configured to return, use:

```
?fields=*all
```

For example, the following query returns all the possible fields for activity xc:20.

GET `/activities/xc:20?fields=*all`

Note that the `*all` query parameter returns all fields that the caller is authorized to view. If there are fields on a resource that a caller is not authorized to view, they are excluded from queries using the `*all` query parameter.

Tutorial: Send a GET with the fields parameter

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see "Tutorial: Set up your Postman environment" on page 21.

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Enter the following and click **Send**:
   GET http://localhost:8180/pc/rest/common/v1/activities
4. Open a second request tab and specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
5. Enter the following and click **Send**:
   GET http://localhost:8180/pc/rest/common/v1/activities?fields=id,subject

Checking your work

Compare the two payloads. Note that the first response payload includes the default fields for activities, whereas the second response payload includes only the `id` and `subject` fields.

Getting resources "as of" a certain date

In some situations, a system API call may need to retrieve an effective-dated resource as of a specific date. For example, a GET may need to retrieve policy pc:10 not as it exists today but as it existed on January 1, 2019. To achieve this, several job and policy endpoints support an `asOfDate` parameter. You can use this parameter to retrieve data about a resource as it existed at a given point in time. The syntax is:

```
?asOfDate=dateValue
```

For example, the following query returns policy pc:10 as it existed on January 1, 2019:

GET `/policies/pc:10?asOfDate=2019-01-01`

If you query for a policy at a point in time before its effective date or after its expiration date, you will get a `BadInputException` with the user message similar to the following: `"Bad value for the 'asOfDate' query parameter - No branch found for date 01/01/2019 12:00 AM"`.

Sorting the result set

For endpoints that return collections, you can sort the elements in the collection. To do this, use:

```
?sort=properties_list
```

where *properties_list* is a comma-delimited list of properties that support sorting for that endpoint.

For example, the following query returns all activities assigned to the current caller and sorts them by due date:

GET `/activities?sort=dueDate`

You can specify multiple sort properties. Resources are sorted based on the first property. Any resources with the same value for the first property are then sorted by the second property, and so on. For example, the following query returns all activities assigned to the current caller and sorts them first by escalation status and then by due date:

GET `/activities?sort=escalated,dueDate`

You can also use the sort query for related resources added through the include parameter. For more information, see "Using query parameters on included resources" on page 57.

Sort orders

The default sort order is ascending. To specify a descending sort, prefix the property name with a hyphen (-). For example, the following queries return all activities assigned to the current caller, sorted by due date. The first query sorts them in ascending order. The second sorts them in descending order.

GET `/activities?sort=dueDate`

GET `/activities?sort=-dueDate`

Determining which values you can sort on

For a given endpoint, you can identify the attributes that are sortable by reviewing the endpoint **Model** in Swagger UI. If a field is sortable, then the schema description of the field includes the text: `"sortable": true`.

For example, the following is the schema description for two fields returned by the Common API's `/activities` endpoint.

```
escalated       boolean
                readOnly: true
                x-gw-extensions: OrderedMap { "filterable": true, "sortable": true }
escalationDate  string($date-time)
                x-gw-nullable: true
```

Note that the `escalated` field includes the `"sortable": true` expression, but the `escalationDate` field does not. This means that you can sort on `escalated`, but not `escalationDate`.

Tutorial: Send a GET with the sort query parameter

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see "Tutorial: Set up your Postman environment" on page 21.

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Enter the following and click **Send**:

   GET `http://localhost:8180/pc/rest/common/v1/activities`
4. Open a second request tab and specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
5. Enter the following and click **Send**:

   GET `http://localhost:8180/pc/rest/common/v1/activities?sort=dueDate`

Checking your results

Compare the two payloads. In the first response payload, the activities are not sorted. In the second response payload, the activities are sorted by due date.

Controlling pagination

Some endpoints return collections. However, the entire collection is typically not returned in a single call. Instead, only the first *N* resources are returned in the first payload. A caller can use "previous" and "next" links to access additional payloads with the previous or next "page" of *N* resources. The practice of separating a list of resources into discrete groups that can be paged through is referred to as *pagination*.

Every endpoint that returns a collection has default pagination behaviors. Each payload contains one page of resources. There are several query parameters that refine these behaviors.

Limiting the number of resources per payload

GETs that return collections typically return multiple root resources. You can use the `pageSize` parameter to limit the number of root resources returned in a given payload. This can be useful when a query may return more resources than what is practical for performance and parsing. To limit the number, use the following syntax:

```
pageSize=n
```

where *n* is the maximum number of resources per payload to return. For example:

GET `/activities?pageSize=20`

Every resource type has a default `pageSize`. This value is used when the query does not specify a `pageSize`. You can specify a `pageSize` less than or greater than the default `pageSize`.

Also, every resource has a maximum `pageSize`, and you cannot execute a query with a `pageSize` larger than the maximum.

For example, suppose a given user has 125 activities, and the activities resource has a default `pageSize` of 25 and maximum `pageSize` of 100.

- GET `/activities` returns the first 25 activities (using the default `pageSize` value).
- GET `/activities?pageSize=10` returns the first 10 activities.
- GET `/activities?pageSize=30` returns the first 30 activities.
- GET `/activities?pagesize=120` returns an error because the value for `pageSize` exceeds the maximum for the resource.

The `pageSize` values for a resource defaults to `defaultPageSize=25` and `maxPageSize=100`. Individual resources may override these values in the API's `apiconfig.yaml` file. (For example, in `shared_ext-1.0apiconfig.yaml`, the `AccountActivityPatterns` resource overrides the default values and uses `defaultPageSize=100` and `maxPageSize=500`.)

You can also use the `pageSize` query for related resources added through the `include` parameter. For more information, see “Using query parameters on included resources” on page 57.

Selecting a single resource in a collection

When a response payload contains a collection, every element in the collection is listed in the `data` section of the payload. For every element, there is a `links` section that contains endpoints relevant to that element. One of the links is the `self` link. For example:

```
{
    "attributes": {
        ...
        "id": "cc:32",
        ... },
        ...
    "links": {
        ...
        "self": {
            "href": "/common/v1/activities/cc:32",
            "methods": [
                "get",
                "patch"
            ]
        }
    }
}
```

The `href` property of the `self` link is an endpoint to that specific element. When necessary, you can use this link to construct a call to act on that element.

Paging through resources

Whenever a response payload includes some but not all of the available resources, the payload also include a collection-level links section at the bottom. These links provide operations and endpoints you can use to act on a specific "page" of resources. (In the following descriptions, *N* is the `pageSize` of the query.)

- The `first` link is an endpoint to the first *N* elements.
  - This appears for all collections.
- The `prev` link is an endpoint to the *N* elements before the current set of elements.
  - This appears if there are elements earlier than the elements in the payload.
- The `next` link is an endpoint to the *N* elements after the current set of elements.
  - This appears if there are elements later than the elements in the payload.
- The `self` link is an endpoint to the current set of elements.
  - This always appears (for elements and for collections).

For example, suppose there are 25 activities assigned to the current owner. The response payloads have a `pageSize` of 5, and a specific payload has the second set of activities (activities 6 through 10). The collection-level links for this payload would be:

```
"links": {
    "first": {
        "href": "/common/v1/activities?pageSize=5&fields=id",
        "methods": [
            "get"
        ]
    },
    "next": {
        "href": "/common/v1/activities?pageSize=5&fields=id&pageOffset=10",
        "methods": [
            "get"
        ]
    },
    "prev": {
        "href": "/common/v1/activities?pageSize=5&fields=id",
        "methods": [
            "get"
        ]
    },
    "self": {
        "href": "/common/v1/activities?pageSize=5&fields=id&pageOffset=5",
        "methods": [
            "get"
        ]
    }
}
```

To access a collection that starts with a specific resource, the system APIs use a `pageOffset` parameter. This parameter is used in the `prev` and `next` links for a collection. The `pageOffset` index starts with 0, so a theoretical `pageOffset=0` would start with the first element and `pageOffset=5` skips the first 5 elements and starts with the sixth.

There can be some complexity involved in determining how to construct a link with the correct `pageOffset` value. Therefore, Guidewire recommends that you use the `prev` and `next` provided in response payloads and avoid constructing `pageOffset` queries of your own.

Retrieving the total number of resources

When querying for data, you can get the total number of resources that meet the criteria. To get this number, use the following syntax:

```
includeTotal=true
```

When you submit this query parameter, the payload includes an additional `total` value that specifies the total. For example:

```
"total": 72
```

When the `includeTotal` query parameter is used, the response payload contains two counting values:

- `count` - The number of resources returned in this payload.
- `total` - The total number of resources that meet the query's criteria.

If the total number of resources that meet the criteria is less than or equal to the `pageSize`, then `count` and `total` are the same. If the total number of resources that meet the criteria is greater than the `pageSize`, then `count` is less than `total`. `count` can never be greater than `total`.

For performance reasons, Guidewire will count the total number of items up to 1000 only. If a `total` value is equal to 1000, the actual count could be 1000 or some number greater than 1000.

> **Note:** If the number of resources to total is sufficiently large, using the `includeTotal` parameter can affect performance. Guidewire recommends you use this parameter only when there is a need for it, and only when the number of resources to total is unlikely to affect performance.

In some situations, you may be interested in retrieving only the total number of resources that meet a given criteria, without needing any information from any specific resource. However, a GET cannot return only an included total. If there are resources that meet the criteria, it must return the first *N* set of resources and at least one field for each resource. For calls that are sent to retrieve only the total number of resources, Guidewire recommends using a call with query parameters that return the smallest amount of resource information, such as GET `...?includeTotal=true&fields=id&pageSize=1`.

You can also use the `includeTotal` query for related resources added through the include parameter. For more information, see “Using query parameters on included resources” on page 57.

Tutorial: Send a GET with the pageSize and totalCount parameters

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Enter the following and click **Send**:

   GET http://localhost:8180/pc/rest/common/v1/activities
4. Open a second request tab and specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
5. Enter the following and click **Send**:

   GET http://localhost:8180/pc/rest/common/v1/activities?pageSize=10&includeTotal=true

Checking your work

Compare the two payloads. In the first payload, the count of activities included in the payload is 14. Also, there is no count for the total number of activities the endpoint could return. In the second payload, the count of activities included in the payload is 10. Also, the count for the total number of activities the endpoint could return is 14. (This appears at the end of the payload.)

Using query parameters on included resources

Some endpoints support the ability to query for a given type of resource and for resource types related to that type. For example, by default, the GET `/activities` endpoint returns only activity resources. However, you can use the `include` query parameter to include any notes related to the returned activities in the response payload. These types of resources are referred to as *included resources*. The technique of adding included resources to a GET is sometimes referred to as *response inclusion* or *read inclusion*.

The syntax for adding included resources is:

```
?include=<resourceName>
```

For example GET `/activities?include=notes` returns all activities assigned to the current caller, and all notes associated with those activities.

For more information on the default behavior of `include`, see "Payload structure for a response with included resources" on page 42.

Most query parameters that can be used on primary resources can also be used on included resources.

Specifying query parameters that apply to an included resource

The general pattern for query expressions on included resources is to specify the included resource name somewhere in the expression's value. For example, the following call gets all activities assigned to the current user and any related notes. The number of notes returned is limited to 5.

GET `/activities?include=notes&pageSize=notes:5`

Included resources and primary resources

Query expressions for included resources are independent of query expressions for primary resources. There could be a query expression for primary resources only, for included resources only, or for both. For example, the following three queries all return activities and their related notes. But, the impact of the `pageSize` parameter varies.

- GET `/activities?pageSize=7&include=notes`
  - The response is limited to 7 activities.
  - There is no limit on notes.
- GET `/activities?include=notes&pageSize=notes:5`
  - There is no limit on activities.
  - The response is limited to 5 notes per activity.
- GET `/activities?pageSize=7&include=notes&pageSize=notes:5`
  - The response is limited to 7 activities.
  - The response is limited to 5 notes per activity.

Included resources and other included resources

Query expressions for each included resource are also independent of query expressions for other included resources. If a given GET includes multiple included resources and you want to apply a given query expression to all included resources, you must specify the query expression for each included resource.

For example, suppose you want to GET all accounts. But you want the response payload to include only the account holder and other contacts, and you want only the id and subtype for these contacts. To get this response, you must send the following:

```
GET /accounts?include=accountHolder,contacts
        &fields=accountHolder:id,contactSubtype
        &fields=contacts:id,contactSubtype
```

Note that, in this example, the logic to restrict the returned fields to only id and subtype needs to be specified for each included resource.

Summary of query parameters for included resources

The filter parameter

You can filter out included resources that do not meet a given criteria.

- **Syntax:** `filter=`*`resource:field:op:value`*
- **Example:**

```
GET policy/v1/policies/pc:10?
            include=contacts&
            filter=contacts:maritalStatus:eq:S
```

- **Returns:** Policy pc:10 and its included contacts that have a marital status of "S" (single)

The fields parameter

You can specify which fields you want returned in the included resources.

- **Syntax:** `fields=`*`resource`*`:`*`field`*`_list`
- **Example:**

```
GET policy/v1/policies/pc:10?
          include=contacts&
          fields=contacts:licenseState,licenseNumber
```

- **Returns:** Policy pc:10 and its included contacts. For the contacts, return only `licenseState` and `licenseNumber`.

The sort parameter

You can sort the included resources. This sorting is reflected in both the payload's `related` sections and the `included` section.

- **Syntax:** `sort=`*`resource`*`:`*`properties_list`*
- **Example:**

```
GET policy/v1/policies/pc:10?
          include=locations&
          sort=locations:locationNum
```

- **Returns:** Policy pc:10 and its included locations, sorted by their location number.

The pageSize parameter

You can specify a maximum number of included resources per root resource. Also, when you use `pageSize` on included resources, there are no `prev` and `next` links at the included resource level.

- **Syntax:** `pageSize=`*`resource`*`:`*`n`*
- **Example:**

```
GET policy/v1/policies/pc:10?
          include=contacts&
          pageSize=contacts:5
```

- **Returns:** Policy pc:10 and up to 5 of its included contacts.

The includeTotal parameter

You can include the total number of included resources.

- **Syntax:** `includeTotal=`*`resource`*`:true`
- **Example:**

```
GET policy/v1/policies/pc:10?
          include=contacts&
          includeTotal=contacts:true
```

- **Returns:** Policy pc:10, its included contacts, and the total number of included contacts for pc:10.

Tutorial: Send a GET with query parameters for included resources

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Enter the following and click **Send**:

   GET `http://localhost:8180/pc/rest/common/v1/activities?included=notes`

4. Open a second request tab and specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
5. Enter the following and click **Send**:

   GET `http://localhost:8180/pc/rest/common/v1/activities?included=notes&fields=id,subject&filter=notes:escalated`

Checking your results

Compare the two payloads. Note the following differences:

In the first payload:

- For activities, the default activity fields are returned.
- For notes, all notes are returned.

In the second payload:

- For activities, only the `id` and `subject` fields are returned.
- For notes, only the escalated notes are returned.

chapter 5

POSTs and request payload structures

This topic discusses the POST operation and how to construct a request payload for creating a single resource. For information on how to create request payloads that specify multiple resources, see “Reducing the number of calls” on page 79.

If you want to interact directly with the concepts in this topic, go to the following tutorials:

- “Tutorial: Create a new note that specifies required fields only” on page 67
- “Tutorial: Create a new note that specifies optional fields” on page 68

Overview of POSTs

A *POST* is a system API operation that creates a resource or a set of related resources in PolicyCenter. The POST operation is also used to execute specific business processes, such as assigning an activity.

A POST consists of the POST operation and the endpoint, such as POST `/activities/{activityId}/notes`, and a request payload. The request payload contains data about the resource to create.

The response to a POST includes an HTTP code indicating success or failure. It also includes a response payload. The contents of the response payload is determined by the endpoint's schema.

- For an endpoint used to create data, the response payload contains data from the request payload. It may also contain data generated by PolicyCenter, such as IDs and timestamps.
- For an endpoint used to execute a business action, the response payload is a resource related to the business action. It could be the resource on which the action was executed. For example, when assigning an activity, the response payload contains the assigned activity. It could also be a resource generated by the business action. For example, when canceling a policy the response payload contains a `JobResponse`.

When a developer is configuring a caller application to POST information to a system API, they will need to determine the correct structure for the request payload. They may also need to parse information out of the response payload. The remainder of this topic discusses how request payloads for resources are structured and how developers can learn about request payload formats.

POSTs are also used to execute business actions. For these types of POSTs, request payloads may be unnecessary, optional, or required. For example:

- A POST that completes an activity does not require a request payload.
  - You can optionally provide a request payload to add a note to the completed activity.
- A POST that assigns an activity requires a request payload. The payload specifies how to assign the activity.

Standardizing payload structures

Communication between caller applications and system APIs is easier to manage when the information in the payloads follows a standard structure. The system APIs have standard structures for both request payloads and response payloads. The structures are defined by data envelopes, and by request and response schemas.

Standardizing information common to all endpoints

A *data envelope* is a wrapper that wraps JSON sent to or returned from the system APIs. To maintain a standard payload structure, the system APIs use two data envelopes: `DataEnvelope` and `DataListEnvelope`.

`DataEnvelope` is used to standardize the format of information for a single element. It specifies a data property with the following properties: `checksum`, `id`, `links` (for a single element), `method`, `refid`, `related`, `type` and `uri`. At a high level, the format of a payload for a single element looks like:

```
{
    "data": {
        "checksum": ...,
        "id":       ...,
        "links":    ...,
        "method":   ...,
        "refid":    ...,
        "related":  ...,
        "type":     ...,
        "uri":      ...
        }
    }
}
```

`DataListEnvelope` is used to standardize the format of information for collections. It specifies the following properties, which are siblings to the data section: `count`, `links` (for a collection), and `total`. At a high level, the format of a payload for a single element looks like:

```
{
    "count"    ...,
    "data": [
        { properties_for_element_1 },
        { properties_for_element_2 },
        ...
    ],
    "links":   ...,
    "total":   ...
}
```

Every property does not appear in every payload. There are different reasons why a property may not appear in a given payload. For example:

- Some properties, such as `refid`, apply only to requests and do not appear in response payloads.
- Some properties, such as `count`, apply only to responses and do not appear in request payloads.
- Some properties, such as `related`, do not appear by default and appear only when the request includes certain query parameters.

Standardizing information specific to a given endpoint

`DataEnvelope` and `DataListEnvelope` provide a standard format for information that is applicable to all request and response payloads. But, different endpoints interact with different types of resources. For each endpoint, some portion of the payload must provide information about a specific type of resource.

To address this, the system APIs also use request schemas and response schemas. A *request schema* is a schema that is used to define the valid structure of a request payload for a specific set of endpoints. Similarly, a *response schema* is a schema that is used to define the valid structure of a response payload for a specific set of endpoints.

Request and response schemas are hierarchical. For example, for responses, the GET `/activity/{activityId}` endpoint uses the `ActivityResponse` schema. This schema has two child schemas: `ActivityData` and `ActivityResponseInclusions`.

Request and response schemas extend `DataEnvelope` or `DataListEnvelope`. This ensures that information relevant to all endpoints appears in payloads in a standard way.

Request and response schemas also define an `attributes` property for the payload. This property is associated with a schema that includes resource-specific information for the payload. For example, the GET `/activity/{activityId}` endpoint specifies an `attributes` property in the `ActivityData` child schema. This property is associated with the `Activity` schema, which contains activity-specific fields, such as `activityPattern` and `activityType`. As a result, response payloads for the GET `/activity/{activityId}` endpoint have this structure:

```
{
    "data": {
        "checksum": ...,
        "attributes" : {
            "activityPattern": ... ,
            "activityType": ...,
            ...},
        "id":         ...,
        "links":      ...,
        "method":     ...,
        "refid":      ...,
        "related":    ...,
        "type":       ...,
        "uri":        ...
        }
    }
}
```

Viewing request schemas

You can use Swagger UI to review the structure of a request payload for a given endpoint. This includes the hierarchy of schemas and the type of information in each schema. The information appears in the description of the endpoint's **body** parameter on the **Model** tabs.

View a request schema in Swagger UI

Procedure

1. Start PolicyCenter.
2. In a web browser, navigate to the Swagger UI for the appropriate API.
   - For more information, see “View an API definition using Swagger UI” on page 25.
3. Click the operation button for the appropriate endpoint. Swagger UI shows details about that endpoint underneath the endpoint name.
   - For example, to view the request schema for POST `/activities/{activityID}/notes`, click the POST button for that endpoint.
4. Scroll down to the **Body** entry in the **Parameters** section. The **Model** tab shows the hierarchy of data envelopes for this endpoint, and the contents of each data envelope.

Designing a request payload

Determining the required, optional, and write-only fields

Within the context of a request payload, each field of a given resource is either:

- **Required** - This field must be included.
- **Optional** - This field can be included or omitted.
- **Read-only** - This field cannot be included.

Required fields

A required field must be included in the request payload. A field can be required for one of several reasons:

- The field is marked as required on the associated schema, and therefore must be included on all POSTs using that schema.

- The field is not marked as required on the associated schema, but it is always required by PolicyCenter. (For example, the underlying database column could be marked as non-nullable with no default and the application does not generate a value for it.)
- The field is not required by the API, but it is sometimes required by PolicyCenter. (For example, there could be a validation rule in PolicyCenter that says non-confidential documents do not require an author, but confidential documents do. Therefore, the author field is required only some of the time.)

Whether a field is required or allowed in a POST does not always match the requiredness of the corresponding data model entity or database column. For example:

- A field may be marked as non-nullable in the database (and therefore "required"). But, PolicyCenter always generates a value for it. Therefore, the field is marked as read-only and not required in the API schema.
- A field may be marked as non-nullable in the database (and therefore "required") and required when an object is created. But once the object is created, the value of the field cannot be changed. Therefore, the field is required for creation, but read-only for updates.

Determining that a field is required by the API

If a field is required by the API, the schema specifies the following property for the field:

```
"requiredForCreate": true
```

For example, the Claim API has a POST `claim/{claimId}/contacts` endpoint that creates a contact for a given claim. One of the data envelopes used by this endpoint to define the request schema is the `ClaimContact` schema. It contains the following:

```
contactSubtype    string
                        x-gw-type: typekey.Contact
                        x-gw-extensions: OrderedMap { "createOnly": true,
                             "requiredForCreate": true }
dateOfBirth       string($date)
                        x-gw-nullable: true
                        x-gw-extensions: OrderedMap { "before": "now",
                             "entitySubtype": "Person" }
```

Note that the `contactSubtype` field has the `"requiredForCreate": true` property, whereas the `dateOfBirth` field does not. This means that the API requires a contact subtype for contact creation, but not a date of birth.

Determining that a field is required by the application

If a field is not required by the API but is required by the application, the only way to identify this is to send a request to the application. If there is a required value that is missing from the request payload, you will get a `BadInputException` response with a message identifying the missing fields. For example:

```
{
        "status": 400,
        "errorCode": "gw.api.rest.exceptions.BadInputException",
        "userMessage": "The 'body' field is required when creating notes"
        }
```

Read-only fields

Read-only fields are fields that are set within the application (either by a user or by application logic) and cannot be set or modified by system API calls. Read-only fields are listed in the request schema as `readonly: true`. You can view this information in Swagger UI from the endpoint's **Model** tab.

For example, this is the Model text for the POST `/activity/{activityId}/notes` endpoint's `createDate` field:

```
createdDate      string($date-time)
                       readOnly: true
```

You cannot include read-only values in a request payload. If you do, the API returns a `BadInputException` with an error message such as:

```
"message": "Property 'createdDate' is defined as read-only and cannot be specified on inputs"
```

Optional fields

From a technical perspective, any field that is neither required nor read-only is optional. These fields can be either include or omitted as appropriate.

Request payload structure

The basic structure for a request payload that creates a single resource is:

```
{
      "data":
          {
              "attributes": {
                    <field/value pairs are specified here>
                }
          }
}
```

For example, this request payload could be used to create a note:

```
{
      "data":
          {
              "attributes": {
                  "subject": "Main contact vacation",
                  "body": "Rodney is on vacation for the entire month of June.
                          During this time, direct any questions to Sarah Jackson.",
                  "confidential": false,
                  "topic": {
                      "code": "general"
                  }
              }
          }
}
```

In some situations, you can create an object using an "empty body" (a body that specifies no values). An object created in this way will contain only default values. In these situations, the payload has an empty `attributes` section:

```
{
      "data":
          {
              "attributes": {
               }
          }
}
```

Specifying scalar values in a request payload

Formats for values are the same for request payloads and response payloads. For a given field, you can use its format in a response payload as a model for how to build a request payload.

On a schema, field value types for scalar values are marked using the `type` property. In request payloads, scalar values follow these patterns:

Field value type	Pattern	Example	Notes
String	`"fieldName" : "value"`	`"firstName" : "Ray",` `"id": "demo_date:12"`	IDs are considered strings.
Integer	`"fieldName" : value`	`"numDaysInRatedTerm": 180`	Unlike the other scalar value types, integer, Boolean, and null values are expressed without quotation marks.
Decimal	`"fieldName" : "value"`	`"speed": "60.0"`	
Date	`"fieldName" : "value"`	`"dateReported": "2020-04-09"`	Expressed using the format *YYYY-MM-DD*

Field value type	Pattern	Example	Notes
Datetime	`"fieldName" : "value"`	`"createdDate": "2020-04-09T18:24:57.256Z"`	Expressed using the format *YYYY-MM-DD*T *hh:mm:ss.fff*Z where T and Z are literal values.
Boolean	`"fieldName" : value`	`"confidential": false`	Unlike the other scalar value types, integer, Boolean, and null values are expressed without quotation marks.
Fields with NULL values	`"fieldName" : null`	`"directValue": null`	You can set any scalar value to null. Express it without quotation marks.

IDs

ID values are assigned by PolicyCenter. Therefore, you cannot specify an ID for an object that is being created. However, you can specify IDs when identifying an existing object that the new object is related to.

Specifying objects in a request payload

The syntax for specifying an object is:

```
"objectName": {
      "field1": value_or_"value",
      "field2": value_or_"value",
      ...
      }
```

For example:

```
"assignedUser": {
      "displayName": "Andy Applegate",
      "isActive": true
      }
```

The value of each object's field either uses or does not use quotation marks based on the datatype of the field. (For example, `assignedUser` has a `displayName` field. The value for this field is a string, so the value is specified in quotes. If `assignedUser` also had an `isActive` field, which was a Boolean, the value would be specified as either `true` or `false` without quotes.

Typekeys and money values are expressed in objects. Each of these are specified using a standard pattern.

Typekeys

Typekeys use the following format:

```
"field": {
    "code": "value"
}
```

For example:

```
"priority": {
    "code": "urgent"
}
```

Typekeys also have a `name` field, which is included in responses. But, the name field is not required. If you include it in a request schema, it is ignored.

Monetary amounts

Monetary amounts use the following format:

```
"field": {
    "amount": "amountValue",
```

```
    "currency": "currencyCode"
}
```

For example:

```
"transactionAmount": {
    "amount": "500.00",
    "currency": "usd"
}
```

Related objects

For information on how to specify related objects in a request payload, see “Request inclusion” on page 80.

Sending POSTs

You use a request tool, such as Postman, to ensure POSTs are well-formed and to review the structure of the response payloads. For more information, see “Requests and responses” on page 19.

Send a POST using Postman

Procedure

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Under the **Untitled Request** label, make sure that POST is selected.
4. In the **Enter request URL** field, enter the URL for the server and the endpoint.
   - For example, to create a new note for activity pc:101 on an instance of PolicyCenter on your machine, enter: `http://localhost:8180/pc/rest/common/v1/activities/pc:101/notes`
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 *raw*.
   c) At the end of the row of radio buttons, change the drop-down list value from *Text* to *JSON*.
   d) Paste the request payload into the text field underneath the radio buttons.
6. Click **Send**. The response payload appears below the request payload.

Tutorial: Create a new note that specifies required fields only

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

In this tutorial, you will create a note whose subject is "API tutorial note 1" for an existing activity. The other fields will not be specified and will be assigned default values by the application (such as not being confidential and having a subject of "General").

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Enter the following call and click **Send**:
   GET `http://localhost:8180/pc/rest/common/v1/activities`
4. Identify the `id` of the first activity in the payload. (It is pc:101.)
5. Enter the following call, but do not click **Send** yet:
   POST `http://localhost:8180/pc/rest/common/v1/activities/pc:101/notes`
6. 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": {
                "body": "API tutorial note 1"
                }
        }
}
```

7. Click **Send**. The response payload appears below the request payload.

Checking your work

1. View the new note in PolicyCenter.
   a. Log on to ClaimCenter as `aapplegate` and click **My Activities**.
   b. Activity pc:101 is a "Verify coverage" activity. Click the subject of this activity. PolicyCenter opens an **Activity Detail** worksheet in the bottom pane.
   c. Click **View Notes**.

The API tutorial note should be listed as one of the notes.

Tutorial: Create a new note that specifies optional fields

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

In this tutorial, you will create a note whose subject is "API tutorial note 2" for an existing activity. You will also specify values for two optional fields: confidential (set to true) and subject (set to "Legal").

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Enter the following call and click **Send**:

   GET `http://localhost:8180/pc/rest/common/v1/activities`
4. Identify the `id` of the first activity in the payload. (It is pc:101.)
5. Enter the following call, but do not click **Send** yet:

   POST `http://localhost:8180/pc/rest/common/v1/activities/pc:101/notes`
6. 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": {
                "body": "API tutorial note 2",
                "confidential": true,
                "topic": {
                    "code": "legal"
                }
            }
        }
}
```

7. Click **Send**. The response payload appears below the request payload.

Checking your work

1. View the new note in PolicyCenter.
   a. Log on to ClaimCenter as `aapplegate` and click **My Activities**.
   b. Activity pc:101 is a "Verify coverage" activity. Click the subject of this activity. PolicyCenter opens an **Activity Detail** worksheet in the bottom pane.
   c. Click **View Notes**.

The API tutorial note should be listed as one of the notes. This note is confidential and it has the category specified in the request payload.

Responses to a POST

Every successful POST generates a response object with a response payload. This payload may contain values generated by PolicyCenter during resource creation that are needed by the caller application For example:

- The resource's Public ID (which is also the system API `id` value)
- Generated human-readable ID values, such as the policy number
- Values generated by a business flow, such as:
  - The user and group that the resource was assigned to
  - Activities generated to process the resource

Similarly to request schemas, response schemas follow certain patterns around using data envelopes to wrap the resource schema. In many instances, the request and response schemas will match.

Fields with null values are omitted

Similar to GETs, the response payloads for POSTs contain only fields whose values are non-null. Fields with `null` values are omitted from the response payload.

If a given field is expected in a response payload but it is missing, this is often because the value was `null`.

POSTs and query parameters

You can use the `fields` query parameter with a POST to control the fields that appear in the response payload. For example, the following creates a note for activity xc:20 based on the request payload. The response payload has the default fields.

POST `/activities/xc:20/notes`

The following also creates a note for activity xc:20 based on the request payload. But, the response payload includes only the `id` field.

POST `/activities/xc:20/notes?fields=id`

Postman behavior with redirects

Some servers automatically redirect incoming calls to different URLs. For example, a call that uses a non-secure URL (one starting with `http://`) may get automatically redirected to a secure URL (one starting with `https://`).

When Postman executes a POST or PATCH and is redirected to a new URL, Postman automatically changes the operation to a GET. This changes the outcome of the operation, as a GET will only retrieve data. This behavior can cause confusion during development, as the developer using Postman may not realize the POST or PATCH is being turned into a GET, or they may not realize why Postman is making the change.

You can avoid this behavior by ensuring that you use URLs in Postman that avoid any redirect behavior from the server. Alternately, you can disable the Postman behavior by disabling the "Automatically follow redirects" setting in **File** > **Settings**.

Business action POSTs

True REST APIs focus exclusively on the CRUD operations (Create, Read, Update, Delete). Like other REST APIs, Cloud API exposes these CRUD operations through endpoints that support the POST, GET, PATCH, and DELETE operations.

However, in some circumstances, a system API needs to trigger a business process that does not readily map to a single Create, Read, Update, or Delete operation. For example, the system APIs expose the ability to assign an activity. This action modifies the value of the activity's `assignedUser` and `assignedGroup` fields. But, the assigned user and group can be determined by assignment logic internal to PolicyCenter. Assignment could vary based on the activity itself, on the current workload of each group, or on whether a given user is on vacation or not. Activity assignment cannot be executed through a PATCH because the caller application cannot always determine how to set the `assignedUser` and `assignedGroup` fields.

In standard REST architecture, there is no operation for this type of business action. Therefore, Cloud API has adopted the following conventions:

- Endpoints that execute business actions use the POST operation.
- Endpoints that execute business actions have paths the end in verbs (such as "assign" or "complete").

Examples of endpoints that execute business actions include:

- POST `/common/v1/activities/{activityId}/assign`, which assigns the corresponding activity
- POST `/common/v1/activities/{activityId}/complete`, which marks the corresponding activity as complete
- POST `/job/v1/jobs/{jobId}/quote`, which quotes the job
- POST `/job/v1/jobs/{jobId}/bind-and-issue`, which bind and issues the job

Business action POSTs and request payloads

All POST endpoints that create resources (such as POST `/common/v1/activities/{activityId}/notes`, which creates a note for the given activity) require a request payload. For some endpoints, the payload can be empty. But, a request payload is always required.

For POST endpoints that execute business actions, payload requirements can vary.

- Some business action POSTs require a payload. (For example, `activities/{activityId}/assign` requires a payload that specifies the assignment criteria.)
- Some business action POSTs can optionally have a payload. (For example, `activities/{activityId}/complete` does not require a payload. But you can specify one if you want to attach a note to the activity while you complete it.)
- Some business action POSTs may not permit any payload.

To determine whether a business action POST requires, allows, or forbids a request payload, refer to the relevant section of this guide.

Business action POSTs and lost updates

When a business process spans multiple calls, the first call is typically either a GET that retrieves data, or a POST that creates data. If the business process involves a POST that executes a business action, this POST typically comes after the first call, and it typically acts on a resource that was queried for or created in a previous call.

It is possible for some other process to modify the data after the initial GET/POST, but before the subsequent business action POST. This can cause a lost update. Within the system APIs, a *lost update* is a modification made to a resource that unintentionally overwrites changes made by some other process.

You can prevent lost updates using checksums. For more information, see “Lost updates and checksums” on page 101.

Improving POST performance

The first time a caller application makes a call to a Cloud API endpoint, the call may take longer to process than normal. This is because the Guidewire server may need to execute tasks for the first call that it does not need to re-execute for subsequent tasks, such as:

- Loading Java and Gosu classes
- Parsing and loading configuration files that are lazy-loaded on the first reference
- Loading data from the database or other sources into local caches
- Initializing database connections

A caller application can avoid having this slow processing time occur during a genuine business call by "warming up" the endpoint. This involves sending a dummy "warm-up request" to trigger these initial tasks. The warm-up request helps subsequent requests execute more rapidly. The best way to accomplish this is with a POST that contains the `GW-DoNotCommit` header. The POST triggers the initial endpoint tasks, and the header identifies that data modifications made by the request are to be discarded and not committed.

For more information, see “Warming up an endpoint” on page 110.

chapter 6

PATCHes

This topic discusses the PATCH operation, which modifies existing data.

If you want to interact directly with the concepts in this topic, go to the following tutorials:

- "Tutorial: PATCH an activity" on page 75

Overview of PATCHes

A *PATCH* is a system API operation that modifies an existing resource or a set of related resources in PolicyCenter.

A PATCH consists of the PATCH operation and the endpoint, such as PATCH `/activities/{activityId}`, and a request payload. The request payload contains the data to modify in the specified resource.

The response to a PATCH includes an HTTP code indicating success or failure. It also includes a response payload. The default response for a PATCH consists of a predetermined set of fields and resources. This may or may not include the data that the PATCH modified.

When a developer is configuring a consumer application to PATCH information to a system API, they will need to determine the correct structure for the request payload. They may also need to parse information out of the response payload.

The PUT operation

Within REST API architecture, there are two operations that modify existing resources - PATCH and PUT. PATCH is used to modify a portion of an existing resource (while leaving other aspects of it unmodified). PUT is used to replace the entire contents of an existing resource with new data. The system APIs support the PATCH operation, but not the PUT operation. This is because nearly every operation that modifies an InsuranceSuite object modifies only a portion of it while keeping the rest of the object untouched. This behavior maps to PATCH, but not to PUT.

The PATCH payload structure

Communication between consumer applications and system APIs is easier to manage when the information in the payloads follows a standard structure. The system APIs have standard structures for both request payloads and response payloads. The structures are defined by data envelopes, and by request and response schemas. POSTs and PATCHes use data envelopes, request schemas, and response schemas in the same way. For more information, see "Standardizing payload structures" on page 62.

Designing a request payload

Designing a request payload for a PATCH is almost the same as designing a request payload for a POST. The only differences are:

- Fields that are marked as `requiredForCreate` are required for POSTs but not for PATCHes.
- Fields that are marked as `createOnly` are allowed in POSTs but not in PATCHes.

For more information on designing request payloads for POSTs, see "Designing a request payload" on page 63.

PATCHes and arrays

You can include arrays in a PATCH request payload. Within the system APIs, PATCHing an array does not add the PATCH members to the members already existing in the array. Instead, the PATCH replaces the existing members with the PATCH members.

For example, in the Jobs API, the JobRoles resource has a `roles` array. This is an array of users on the job and the role of each user (such as Creator or Underwriter). The following PATCH payload will set the roles array to a single user (user default_data:1) with the role of Auditor. If there were user/role pairs in this array before the PATCH, those pairs will be removed and the only user/role pair will be user default_data:1/Auditor.

```
{
  "data":
    {
      "attributes": {
        "roles": [
          {
            "group": "systemTables:1",
            "role": {
              "code": "Auditor"
            },
            "user": "default_data:1"
          }
        ]
      }
    }
}
```

If you want a PATCH to be additive to an array, you must first determine the existing members of the array, and then specify an array in the PATCH with the existing members as well as the ones you wish to add.

Sending PATCHes

You can use a request tool, such as Postman, to ensure PATCHes are well-formed and to review the structure of the response payloads. For more information on Postman, see "Requests and responses" on page 19.

Send a PATCH using Postman

Procedure

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Under the **Untitled Request** label, make sure that PATCH is selected.
4. In the **Enter request URL** field, enter the URL for the server and the endpoint.
   - For example, to patch activity pc:2 on an instance of PolicyCenter on your machine, enter: `http://localhost:8180/pc/rest/common/v1/activities/cc:2`
5. Specify the request payload.
   - In the first row of tabs (the one that starts with **Params**), click **Body**.
   - In the row of radio buttons, select *raw*.
   - At the end of the row of radio buttons, change the drop-down list value from *Text* to *JSON*.

- Paste the request payload into the text field underneath the radio buttons.

6. Click **Send**. The response payload appears below the request payload.

Tutorial: PATCH an activity

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see "Tutorial: Set up your Postman environment" on page 21.

In this tutorial, you will find an open activity from the sample data. You will then update the activity's subject and priority.

Tutorial steps

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
3. Query for all open activities by entering the following call and clicking **Send**:
   a. GET `http://localhost:8180/pc/rest/common/v1/activities?filter=status:eq:open`
4. For the first activity in the response payload that is assigned to Andy/Alice Applegate, note the following information:
   a. Activity ID
   b. Priority
   c. Subject
5. On the same tab, enter the following call, but do not click **Send** yet:
   a. PATCH `http://localhost:8180/pc/rest/common/v1/activities/<activityID>`
6. 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. For subject, specify the original subject with an additional "!".

```
{
  "data": {
    "attributes": {
      "subject" : "<originalSubject>!",
      "priority": {
        "code": "low"
        }
    }
  }
}
```

7. Click **Send**. The response payload appears below the request payload.

Checking your work

1. View the modified activity in PolicyCenter.
   a. Log on to PolicyCenter as the user `aapplegate`. In the left-hand side bar, click **My Activities**. PolicyCenter shows the **My Activities** screen, which shows the open activities assigned to Alice.
   b. Click the **Priority** column to sort the activities in reverse priority order.

The patched activity (whose priority is now Low) should be at or near the top of the list. The patched activity will have a subject ending with an "!".

Responses to a PATCH

Every successful PATCH generates a response object with a response payload. Depending on the default fields returned and whether any query parameters have been specified, the response payload may or may not contain the values modified by the PATCH.

Similarly to request schemas, response schemas follow certain patterns around using data envelopes to wrap the resource schema. In many instances, the request and response schemas will match.

Fields with null values are omitted

Similar to GETs and POSTs, the response payloads for PATCHes contain only fields whose values are non-null. Fields with null values are omitted from the response payload.

If a given field is expected in a response payload but it is missing, this is often because the value was null.

PATCHes and query parameters

You can use the `fields` query parameter with a PATCH to control the fields that appear in the response payload. For example, the following PATCHes a note for activity xc:20 based on the request payload. The response payload has the default fields.

PATCH `/activities/xc:20/notes`

The following also PATCHes a note for activity xc:20 based on the request payload. But, the response payload includes only the `id` field.

POST `/activities/xc:20/notes?fields=id`

PATCHes and lost updates

When a business process spans multiple calls, the first call is typically either a GET that retrieves data, or a POST that creates data. If the business process involves a PATCH, this PATCH typically comes after the first call, and it typically acts on a resource that was queried for or created in a previous call.

It is possible for some other process to modify the data after the initial GET/POST, but before the subsequent PATCH. This can cause a lost update. Within the system APIs, a *lost update* is a modification made to a resource that unintentionally overwrites changes made by some other process.

You can prevent lost updates using checksums. For more information, see “Lost updates and checksums” on page 101.

Postman behavior with redirects

Some servers automatically redirect incoming calls to different URLs. For example, a call that uses a non-secure URL (one starting with `http://`) may get automatically redirected to a secure URL (one starting with `https://`).

When Postman executes a POST or PATCH and is redirected to a new URL, Postman automatically changes the operation to a GET. This changes the outcome of the operation, as a GET will only retrieve data. This behavior can cause confusion during development, as the developer using Postman may not realize the POST or PATCH is being turned into a GET, or they may not realize why Postman is making the change.

You can avoid this behavior by ensuring that you use URLs in Postman that avoid any redirect behavior from the server.

chapter 7

DELETEs

This topic provides an overview of DELETEs, which are used to delete resources.

If you want to interact directly with the concepts in this topic, go to the following tutorials:

- Tutorial: “Tutorial: DELETE a note” on page 77

Overview of DELETEs

Within the context of true REST APIs, a DELETE is an endpoint operation that deletes a resource. This typically involves removing the resource from the underlying database.

Within the context of Cloud API, a *DELETE* is a system API operation that "removes" an existing resource from PolicyCenter. What it means to "remove" the resource depends on the resource type. The DELETE operation federates to the PolicyCenter code that matches the functionality most closely tied to deletion. That code could theoretically:

- Delete the corresponding data model instance from the operational database.
- Mark the corresponding data model instance as retired.
- Modify the corresponding data model instance and other related instances to indicate the data is no longer active or available.

Unlike GET, POST, and PATCH, there are only a small number of endpoints in the base configuration that support DELETE. This is because, in most cases, PolicyCenter does not support the removal of data. Several business objects can be approved, canceled, completed, closed, declined, rejected, retired, skipped, or withdrawn. But only a few can be deleted.

A DELETE call consists of the DELETE operation and the endpoint, such as DELETE `/notes/{noteId}`. Similar to GETs, DELETEs are not permitted to have a request payload.

The response to a DELETE includes an HTTP code indicating success or failure. DELETE responses do not have a response payload.

Tutorial: DELETE a note

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

In this tutorial, you will send calls as Amy Clinton (user name `aclinton`). In the base configuration, Amy Clinton is an underwriting supervisor who has permission to delete notes. As Amy Clinton, you will create a note and query for it. You will then delete the note and attempt to query for it a second time.

Tutorial steps

1. In Postman, create an initial request by:
   a. Clicking the + to the right of the **Launchpad** tab.
   b. Specifying *Basic Auth* authorization using user `aclinton` and password `gw`.
2. Enter the following call, but do not click **Send** yet:
   a. POST `http://localhost:8180/pc/rest/common/v1/activities/pc:13/notes`
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": {
        "body": "API tutorial note to be deleted"
      }
    }
}
```

4. Click **Send**. In the response payload, identify the note's id.
5. Create a second request by:
   a. Clicking the + to the right of the **Launchpad** tab.
   b. Specifying *Basic Auth* authorization using user `aclinton` and password `gw`.
6. Verify that the new note exists by entering the following call and click **Send**:
   a. GET `http://localhost:8180/pc/rest/common/v1/notes/<noteID>`
7. Create a third request by:
   a. Clicking the + to the right of the **Launchpad** tab.
   b. Specifying *Basic Auth* authorization using user `aclinton` and password `gw`.
8. Delete the new note by entering the following call and click **Send**:
   a. DELETE `http://localhost:8180/pc/rest/common/v1/notes/<noteID>`
9. Verify the new note no longer exists by returning to the second tab (the one with the GET) and clicking **Send** a second time.

Checking your work

The first GET (which was executed before the DELETE) should return details about the note.

The second GET (which was executed after the DELETE) should return an error message similar to the one below:

```
{
    "status": 404,
    "errorCode": "gw.api.rest.exceptions.NotFoundException",
    "userMessage": "No resource was found at path /common/v1/notes/xc:301"
}
```

DELETEs and lost updates

When a business process spans multiple calls, the first call is typically either a GET that retrieves data, or a POST that creates data. If the business process involves a DELETE, this DELETE typically comes after the first call, and it typically acts on a resource that was queried for or created in a previous call.

It is possible for some other process to modify the data after the initial GET/POST, but before the subsequent DELETE. This can cause a lost update. Within the system APIs, a *lost update* is a modification made to a resource that unintentionally overwrites changes made by some other process.

You can prevent lost updates using checksums. For more information, see “Lost updates and checksums” on page 101.

chapter 8

Reducing the number of calls

Good integration design typically involves writing integration points so that the number of calls between services is as small as possible. Cloud API includes multiple features that let caller applications execute multiple requests in a single call. This topic discusses these features in detail.

Features that execute multiple requests at once

Cloud API has several features that let caller applications execute multiple requests in a single call: request inclusion, batch requests, and composite requests.

Request inclusion is a technique for POSTs and PATCHes in which the call consists of the following:

- A single parent request that creates or modifies a resource
- One or more child requests that create or modify resources related to the parent resource

If either the parent request or any child request fails, the entire request fails.

For details about request inclusion, see “Request inclusion” on page 80.

Batch requests are requests which consist of multiple sibling subrequests, with no parent request. Each subrequest is executed non-transactionally in the order it appears. If a given subrequest fails, other subrequests in the batch might still be attempted. Also, there is no mechanism for passing information from one subrequest to another. Each subrequest is essentially independent from the others.

For details about batch requests, see “Batch requests” on page 85.

Composite requests are requests which consist of multiple sibling subrequests, with no parent request. Each subrequest is executed transactionally in the order it appears. If a given subrequest that attempts to commit data fails, the entire composite request fails. Information can be passed from one subrequest to another.

For details about composite requests, see “Composite requests” on page 89.

Comparing features that execute multiple requests

The following table compares these features.

Feature	Request inclusion	Batch requests	Composite requests
Request architecture	A parent request with one or more child requests	Sibling subrequests (with no parent request)	Sibling subrequests (with no parent request)

Feature	Request inclusion	Batch requests	Composite requests
The endpoint to call	The endpoint that creates or modifies the parent object (though not all endpoints support request inclusion)	The relevant API's `/batch` endpoint	The Composite API's `/composite` endpoint
Behavior when one subrequest that attempts to commit data fails	The entire request fails	Other subrequests may still be attempted	The entire request fails
Passing information between subrequests	Through the use of refids	Not possible	Through the use of variables
Allows GET subrequests?	No	Yes	Yes
Allows DELETE subrequests?	No	Yes	Yes
Allows business action POST subrequests (such as `/assign`)?	No	Yes	Yes
Allows the creation or modification of two unrelated objects?	No	Yes	Yes

Determining which feature to use

There is no simple algorithm for determining the appropriate feature to use. In some situations, it may be possible to use multiple features, but it is easier to write the code using one particular feature. The following guidelines may help you determine the best feature to use:

- Use request inclusion or composite requests if:
  - All subrequests must succeed or fail as a unit.
  - Information must be passed from one subrequest to another.
  - The subrequests must use endpoints from different APIs.
- Use batch requests or composite requests if:
  - At least some of the subrequests are GETs, DELETEs, or business action POSTs

At a high level, a composite request is typically the most robust option. If there is a choice of which feature to use, it may be best or easiest to use composite requests.

Request inclusion

Request inclusion is a technique for POSTs and PATCHes in which the call consists of the following:

- A single parent request that creates or modifies a resource
- One or more child requests that create or modify resources related to the parent resource

If either the parent request or any child request fails, the entire request fails.

The parent resource is often referred to as the *root resource*. The root resource is specified in the payload's `data` section. The related resources are specified in the payload's `included` section.

For example, a caller can use a single POST `/accounts` to create a new account, a set of AccountContacts for that account, a set of AccountLocations for that account, and a set of documents for that account.

In order to use request inclusion, the following must be true:

- There must be a POST or PATCH endpoint for the root resource.
- This endpoint must have the child resource as part of its `included` section.
- There must also be a POST or PATCH endpoint for the child resource.

The syntax for request inclusion varies slightly, depending on whether the relationship between the root resource and the included resource is a "simple parent/child relationship", or a "named relationship".

Syntax for simple parent/child relationships

In most cases, the relationship between the root resource and an included resource is a simple parent/child relationship. Examples of this include:

- An activity and its notes
- An account and its AccountLocations

When using request inclusion for simple parent/child relationships, the JSON has the following structure:

```
{
  "data" : {
    "attributes": {
      ...
    }
  },
  "included": {
    "<resourceType>": [
      {
        "attributes": {
          ...
        },
        "method": "post",
        "uri": "/../this/..."
      }
    ]
  }
}
```

The data section

The `data` section includes information about the root resource, such as its `attributes`. (For PATCHes, the `data` section could also include a checksum value for the root resource.)

The included section

The `included` section consists of one or more subsections of included resources. Each subsection starts with the resource type name. Then, one or more resources of that type can be specified. For each resource, you must specify:

- The resource's `attributes`
- The `method` and `uri` to create or modify the resource.

The method and uri fields

Request inclusion involves a single call to a single endpoint. But internally, the system APIs use multiple endpoints to execute the call. For every included resource, you must specify the operation and uri relevant to that resource.

For example, suppose you are writing a POST /activities call to create an activity and a note for that activity. The note is the included resource. The `included` section would contain code similar to this:

```
"included": {
  "Note": [
    {
      "attributes": {
          ...
      },
      "method": "post",
      "uri": "/common/v1/activities/this/notes"
    }
  ]
}
```

This specifies that in order to create the note, use the POST `/common/v1/activities/{activityId}/notes` endpoint.

The uri must start with the API name, such as "/common/v1".

The uri must also specify the ID of the root resource. When the root resource and the included resources are being created at the same time, the root resource does not yet have an ID. Therefore, the keyword `this` is used in the uri to represent the root resource's ID.

Example of request inclusion for simple parent/child relationships

The following payload is an example of creating an activity on account pc:10, and a note for that activity.

```
POST http://localhost:8180/pc/rest/account/v1/accounts/pc:10/activities

{
  "data": {
    "attributes": {
      "activityPattern": "general_reminder"
    }
  },
  "included": {
    "Note": [
      {
        "attributes": {
            "subject": "Initial phone call",
            "body": "Initial phone call with claimant"
        },
        "method": "post",
        "uri": "/common/v1/activities/this/notes"
      }
    ]
  }
}
```

Syntax for named relationships

In some cases, the relationship between the root resource and an included resource is more than just a parent/child relationship. It is a "named relationship" in which the relationship has a special designation or label.

For example, every account has an "account holder". This is an AccountContact who is the legal holder of the account. An account can have any number of child AccountContacts, but only one of those AccountContacts can be labeled as the *account holder*.

When using request inclusion for named relationships, the JSON has the following structure. The lines that are not required for simple parent/child relationships but are required for named relationships appear in bold:

```
{
  "data" : {
    "attributes": {
      "<relationshipField>": "<arbitraryRefId>"
      ...
    }
  },
  "included": {
    "<resourceType>": [
      {
        "attributes": {
          ...
        },
        "refid": "<arbitraryRefId>",
        "method": "post",
        "uri": "/../this/..."
      }
    ]
  }
}
```

The data section

The `data` section includes information about the root resource, such as its `attributes`. (For PATCHes, the `data` section could also include a checksum value for the root resource.)

The `data` section also includes the field that names the relationship with the child resource. This field is set to some reference ID. The value of this reference ID is arbitrary. It can be any value, as long as the value also appears with the child resource in the `included` section.

The included section

The `included` section consists of one or more subsections of included resources. Each subsection starts with the resource type name. Then, one or more resources of that type can be specified. For each resource, you must specify:

- The resource's `attributes`

- The `method` and `uri` to create or modify the resource.

The refid field

Each `included` resource must include a `refid` field. This field must be set to the same arbitrary reference ID used in the `data` section. The system APIs use refids to identify which child resource in the `included` section has the named relationship with the root resource.

The method and uri fields

Request inclusion involves a single call to a single endpoint, but the system APIs internally use multiple endpoints to execute the call. For every included resource, you must specify the operation and uri relevant to that resource.

For example, suppose you are writing a POST /accounts call to create an account and an AccountContact who is the "accountHolder". The AccountContact is the included resource. The `included` section would contain code similar to this:

```
"included": {
  "AccountContact": [
    {
      "attributes": {
        ...
      },
      "refid": "...",
      "method": "post",
      "uri": "/account/v1/accounts/this/contacts"
    }
  ]
}
```

This specifies that in order to create the AccountContact, use the POST `/account/v1/{accountId}/contacts` endpoint.

The uri must start with the API name, such as "/account/v1".

The uri must specify the ID of the root resource. When the root resource and the included resources are being created at the same time, the root resource does not yet have an ID. Therefore, the keyword `this` is used in the uri to represent the root resource's ID.

Example of request inclusion for named relationships

The following payload is an example of creating an account and an AccountContact for the account whose relationship is "accountHolder". (Accounts also require a primary location, so the payload also create an AccountLocation whose relationship is "primaryLocation".) For more information on creating accounts, see “Creating an account” on page 123.

```
POST http://localhost:8180/pc/rest/account/v1/accounts

{
  "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": "Preston",
          "primaryAddress": {
            "addressLine1": "2850 S Delaware St #400",
            "city": "San Mateo",
            "postalCode": "94403",
            "state": {
              "code": "CA"
            }
          }
        },
        "method": "post",
        "refid": "newperson",
        "uri": "/account/v1/accounts/this/contacts"
      }
    ],
    "AccountLocation": [
      {
        "attributes": {
          "locationCode": "0001",
          "locationName": "Location 0001",
          "nonSpecific": true,
          "postalCode": "94403",
          "state": {
            "code": "CA"
          }
        },
        "method": "post",
        "refid": "newloc",
        "uri": "/account/v1/accounts/this/locations"
      }
    ]
  }
}
```

Additional request inclusion behaviors

PATCHing and POSTing in a single request

When you execute a POST with request inclusion, the operation for each included resource must also be POST.

When you execute a PATCH with request inclusion, the operation for an included resource could be either POST or PATCH.

- If you want to modify an existing resource and create a new related resource, the included resource's operation is POST.
- If you want to modify an existing resource and modify an existing related resource, the included resource's operation is PATCH.

Requests succeed or fail as a unit

When a POST or PATCH uses request inclusion, it is possible that there could be a failure either of the operation on the root resource or the operation on any of the included resources. If any operation fails, the entire request fails and none of the objects are POSTed or PATCHed.

Included resources cannot reference resources other than the root resource

When using request inclusion, each included resource must specify its own operation and uri. The uri is expected to reference the root resource using the keyword `this`. This ensures that when the included resource is POSTed or PATCHed, the included resource is related to the root resource.

For example, suppose a POST is creating an account and an AccountContact. The uri for the Accountcontact would have a value such as "`/account/v1/accounts/this/contacts`".

From a syntax perspective, you could attempt to attach an included resource not to the root resource, but rather to some other existing resource. For example, instead of referencing the root resource, the uri for the AccountContact could reference an existing account, such as "`/account/v1/accounts/pc:20/contacts`". This uri says "create an AccountContact and attach it not to the root resource of this POST, but rather to the existing account pc:20".

The system APIs will not allow this. Any attempt to POST or PATCH an included resource to something other than the root resource will cause the operation to fail.

Batch requests

From a system API perspective, a *batch request* is a set of requests that are executed in a non-transactional sequence. Each call within the batch request is referred to as a *subrequest*. The object that contains all of the subrequests is referred to as the *main request*.

Subrequests are executed serially, in the order they are specified in the request payload. PolicyCenter then gathers the response to each subrequest and returns them in a single response payload. Once again, the subresponses appear in the same order as the corresponding subrequests.

When the response to a batch request contains a response code of 200, it means the batch request itself was well-formed. However, each individual subrequest may have succeeded or failed.

Batch requests are limited to a maximum number of subrequests. The maximum is specified by the `MaximumAllowedNumberOfBatchSubRequests` configuration parameter. In the base configuration, this parameter is set to 100. Batch requests with more than the maximum number of subrequests fail with a `BadInputException`.

Batch requests are limited to a maximum number of subrequests. For more information, see “Configuring the maximum number of subrequests” on page 89.

Optional subrequest attributes

A subrequest can optionally have query parameters that refine the corresponding subresponse payload.

By default, each subrequest inherits the information in the headers of the main request object. The one exception to this is the `GW-Checksum` header. This header is not inherited because it is unlikely that a single checksum value will correspond to multiple sub-requests. You can optionally specify header values for an individual subrequest, which will override the corresponding values in the main request header.

If a subrequest fails, the default is to continue processing the remaining subrequests. For each subrequest, you can optionally specify that if the subrequest fails, PolicyCenter must skip the remaining subrequests.

For a complete list of options and further information on how they work, refer to the `batch_pl-1.0.schema.json` file.

Batch request syntax

Batch request call syntax

The syntax for the batch request call is:

```
POST <applicationURL>/rest/<apiWithVersion>/batch
```

For example, if you were executing a Policy API batch from an instance of PolicyCenter on your local machine, the call would be:

```
POST http://localhost:8180/pc/rest/policy/v1/batch
```

Batch request payload syntax

The basic syntax for a batch request payload is:

```
{
  "requests": [
    {
      "method": "<method>",
      "path": "<path>",
      "query": "<queryParameters>",
      "data":
        {
          "attributes": {
            "<field1>": "<value1>",
            "<field2>": "<value2>",
            ...
          }
        }
```

```
    },
    {
      "method": "<method>",
      "path": "<path>",
      "query": "<queryParameters>",
      "data":
        {
          "attributes": {
            "<field1>": "<value1>",
            "<field2>": "<value2>",
            ...
          }
        }
    },
    ...
  ]
}
```

where:

- *<method>* is the operation in lower case, such as "get", "post", "patch", or "delete".
- *<path>* is the endpoint path.
  - This path starts as if it was immediately following the API path (including the major version, such as "/v1"). For example, suppose the path for a command when executed in isolation is: `http://localhost:8180/pc/rest/policy/v1/policies/cc:22/activities/cc:55`. The path within a batch is: `/policies/cc:22/activities/cc:55`
- *<queryParmaters>* is an optional string of query parameters. Start this string without an initial "?".
- *<field1*/*<value>* are the field and value pairs of the request body.

The following sections provide examples of how to use this syntax.

Simple batch requests

The most simple batch request consist of default GET subrequests. This involves no query parameters and no request payloads.

For this example, the response will consist of three subresponses. Each subresponse will consist of the default fields for each policy.

```
{
  "requests": [
    {
      "method": "get",
      "path": "/policies/demo_sample:1"
    },
    {
      "method": "get",
      "path": "/policies/demo_sample:2"
    },
    {
      "method": "get",
      "path": "/policies/demo_sample:3"
    }
  ]
}
```

Batch requests with query parameters

The following is an example of a batch request with multiple GET subrequests. This example includes query parameters for some of the GETs. As shown in the example, it is possible for some subrequests to use query parameters while others do not. The subrequests that use query parameters can use different query parameters.

The response will consist of three subresponses. The fields in each subresponse will vary based on the query parameters.

```
{
  "requests": [
    {
      "method": "get",
      "path": "/policies/demo_sample:1",
      "query": "sort=createdDate"
    },
```

```
    {
      "method": "get",
      "path": "/policies/demo_sample:2",
      "query": "fields=*all"
    },
    {
      "method": "get",
      "path": "/policies/demo_sample:3"
    }
  ]
}
```

Batch requests with request payloads

The following is an example of a batch request with multiple POST subrequests. This example includes request payloads for each subrequest.

In this example, two notes are POSTed to different activities. But it would also be possible to POST each note to the same activity.

```
{
  "requests": [
    {
      "method": "post",
      "path": "/activities/xc:11/notes",
      "data":
        {
          "attributes": {
            "body": "Batch note 1"
          }
        }
    },
    {
      "method": "post",
      "path": "/activities/xc:73/notes",
      "data":
        {
          "attributes": {
            "body": "Batch note 2"
          }
        }
    }
  ]
}
```

Batch requests with distinct operations

Every subrequest in a batch request is distinct from the other subrequests. There is no requirement for any subrequest to share any attribute with any other subrequest. Thus, the following is an example of a batch request with multiple subrequests where each subrequest uses a different operation.

```
{
  "requests": [
    {
      "method": "post",
      "path": "/activities/xc:21/notes"
      "body": {
        "data": {
          "attributes": {
            "body": "Batch activity 1",
            "subject": "Batch activity 1",
            "topic": {
              "code": "general",
              "name": "General"
            }
          }
        }
      }
    },
    {
      "method": "patch",
      "path": "/notes/xc:22",
      "body": {
        "data": {
          "attributes": {
            "body": "PATCHed note body"
          }
        }
      },
    },
```

```
    {
      "method": "delete",
      "path": "/notes/xc:23"
    },
    {
      "method": "get",
      "path": "/activities/xc:24/notes",
      "query": "sort=subject&fields=id,subject"
    }
  ]
}
```

Specifying subrequest headers

The following is an example of a batch request where each subrequest has a header that overrides the main request header.

```
{
  "requests": [
    {
      "method": "delete",
      "path": "/activities/xc:55",
      "headers": [
        {
          "name": "GW-Checksum",
          "value": "2"
        }
      ]
    },
    {
      "method": "delete",
      "path": "/activities/xc:57",
      "headers": [
        {
          "name": "GW-Checksum",
          "value": "4"
        }
      ]
    }
  ]
}
```

Specifying onFail behavior

The following is an example of a batch request that uses `onFail` to specify that if any of the subrequests fail, the remaining subrequests need to be skipped.

```
{
  "requests": [
    {
      "method": "patch",
      "path": "/activities/xc:93",
      "body": {
        "data": {
          "attributes": {
            "subject": "PATCH body 1"
          }
        }
      },
      "onFail": "abort"
    },
    {
      "method": "patch",
      "path": "/activities/xc:94",
      "body": {
        "data": {
          "attributes": {
            "subject": "PATCH body 2"
          }
        }
      },
      "onFail": "abort"
    },
    {
      "method": "patch",
      "path": "/activities/xc:95",
      "body": {
        "data": {
          "attributes": {
            "subject": "PATCH body 3"
          }
        }
```

```
      }
    }
  ]
}
```

Configuring the maximum number of subrequests

Batch requests are limited to a maximum number of subrequests. The maximum is specified by the `MaximumAllowedNumberOfBatchSubRequests` configuration parameter. In the base configuration, this parameter is set to 100. Batch requests with more than the maximum number of subrequests fail with a `BadInputException`.

The greater the number of subrequests in a batch request, the greater the chances that there will be a compromise in performance. A batch request with the maximum number of subrequests could result in a slow response, depending on what the maximum is and what those subrequests are doing.

In the base configuration, the maximum number of subrequests is 100. This can be raised to a greater value. However, batch requests with a significantly large number of subrequests could have negative consequences, such as:

- The request consuming a significant amount of service resources. This could include both memory and database resources.
- The request taking so long to complete that it times out before a response can be provided to the caller.

Consequently, Guidewire recommends setting the maximum number of subrequests to the lowest value that is needed. If there is a legitimate business need to raise the maximum above 100, Guidewire recommends the limit be raised only as much as is absolutely necessary.

Also, be aware that batch requests are subject to any application server limitations around the maximum size of a request body. Thus, it is possible for a batch request to be too large to process, even if the number of subrequests is at or below the allowed maximum.

Composite requests

From a Cloud API perspective, a *composite request* is a set of requests that are executed in a single InsuranceSuite bundle (which corresponds to a single database transaction).

- A composite request can include one or more *subrequests* that create or modify data. Either all of the subrequests succeed, or none of them are executed. Each subrequest is a separate unit of work.
- A composite request can also include one or more *subselections* that query for data.

Subrequests and subresponses are executed serially, in the order they are specified in the composite request payload. PolicyCenter then gathers the response to each subrequest and subselection and returns them in a single response payload. The responses to each subrequest and subselection appear in the same order as the original composite request.

Composite requests can make use of variables. This allows data created by the execution of one subrequest to be used by later subrequests.

Composite requests are limited to a maximum number of subrequests. For more information, see“Configuring the maximum number of subrequests” on page 98.

Constructing composite request calls

The /composite endpoint

To create a composite request, use the `/composite` endpoint in the Composite API. (This is different than batches. Every API has its own `/batch` endpoint. But in all of Cloud API, there is only one `/composite` endpoint, and it is in the Composite API.)

The syntax for the composite request call is:

```
POST <applicationURL>/rest/composite/v1/composite
```

Sections of a composite request

A composite request can have up to two sections:

- A `requests` section, which contains the subrequests that commit data.
- A `selections` section, which contains the subselections that query for data. These are executed after the subrequests, and only if all the subrequests commit data successfully.

At a high level, the syntax for these sections is as follows:

```
{
  "requests": [
    {
      <subrequest 1>
    },
    {
      <subrequest 2>
    },
    ...
  ],
  "selections": [
    {
      <subselection 1>
    },
    {
      <subselection 2>
    },
    ...
  ]
}
```

The requests section

In the requests section, the only supported operations are POST, PATCH, and DELETE. This includes both POSTs that create data and POSTs that execute business actions (such as POST `/assign`).

The basic syntax for the requests section is shown below.

```
{
  "requests": [
    {
      "method": "<post/patch/delete>",
      "uri": "<path>",
      "body": {
        "data": {
          "attributes": {
            "<field1>": "<value1>",
            "<field2>": "<value2>",
            ...
            }
        }
      }
    },
    {
      <next subrequest>
    },
    ...
    {
      <final subrequest>
    }
  ]
}
```

For example, the following simple composite request creates two notes for activity xc:202.

```
POST <applicationURL>/rest/composite/v1/composite

{
  "requests": [
    {
      "method": "post",
      "uri": "/common/v1/activities/xc:202/notes",
      "body": {
        "data": {
          "attributes": {
            "body": "Cloud API note #1."
          }
        }
      }
    },
```

```
    {
      "method": "post",
      "uri": "/common/v1/activities/xc:202/notes",
      "body": {
        "data": {
          "attributes": {
            "body": "Cloud API note #2."
          }
        }
      }
    }
  ]
}
```

For the complete syntax that includes all composite request features, see “Complete composite request syntax” on page 98.

Using variables to share information across subrequests

Information from one subrequest can be used in later subrequests. You can do this through the use of composite variables.

Declaring variables

Composite variables are declared in a subrequest's `vars` section. Each variable has a `name` and `path`. The `name` is an arbitrary string. The `path` specifies a value from the subrequest's response payload as a JSON path expression.

For example, suppose a subrequest that creates an activity has the following:

```
    "vars": [
      {
        "name": "newActivityId",
        "path": "$.data.attributes.id"
      }
    ]
```

This creates a variable named `newActivityId`, which is set to the value of the data section's `attributes` section's `id` field (which would typically be the ID of the newly created activity).

Referencing variables

To reference a variable, use the following syntax:

```
${<varName>}
```

You can use variables anywhere in the body of a subrequest. The most common uses for variable values are:

- In an `attributes` field
- Within the path of a `uri`
- As part of a query parameter

For example, suppose there is a subrequest that creates an activity, and it is followed by a subrequest that creates a note. The first subrequest creates a `newActivityId` variable as shown previously. The `uri` for the second subrequest is:

```
"uri": "/common/v1/activities/${newActivityId}/notes"
```

This would create the new note as a child of the first subrequest's activity.

The following is the complete code for the previous examples.

```
{
  "requests": [
    {
      "method": "post",
      "uri": "/account/v1/accounts/pc:34/activities",
      "body": {
        "data": {
          "attributes": {
              "activityPattern": "contact_insured",
              "subject": "Cloud API activity"
          }
```

```
        }
      },
      "vars": [
        {
          "name": "newActivityId",
          "path": "$.data.attributes.id"
        }
      ]
    },
    {
      "method": "post",
      "uri": "/common/v1/activities/${newActivityId}/notes",
      "body": {
        "data": {
          "attributes": {
            "body": "Cloud API note #1."
          }
        }
      }
    }
  ]
}
```

Responses to the subrequests

The response to a composite request contains a responses section. This section contains one subresponse for each subrequest. Every subresponse has three sections:

- A `body` section, which by default contains the default response data defined in the corresponding endpoint.
- A `headers` section, which contains any custom headers.
- A `status` field, which indicates the subresponse's status code.

For example, the following is the responses section and the first subresponse for a composite request whose first subrequest created an activity:

```
"responses": [
  {
    "body": {
      "data": {
        "attributes": {
          "activityPattern": "contact_insured",
          "activityType": {
            "code": "general",
            "name": "General"
            },
          "assignedByUser": {
            "displayName": "Andy Applegate",
            "id": "demo_sample:1",
            "type": "User",
            "uri": "/admin/v1/users/demo_sample:1"
          },
          ...
        },
        "checksum": "0",
        "links": {
          "assign": {
            "href": "/common/v1/activities/cc:403/assign",
            "methods": [
              "post"
            ]
          },
          ...
        }
      }
    },
    "headers": {
      "GW-Checksum": "0",
      "Location": "/common/v1/activities/xc:403"
    },
    "status": 201
  },
```

Fields whose values are generated when data is committed

The individual subresponses to each subrequest specify data that has technically not been committed yet. However, some fields contain values that are not generated until the data is committed.

When a subresponse includes a value that is generated as part of the commit, Cloud API makes effort to match the data that will be committed as closely as possible. For example, the composite request reserves ID values so that these IDs can be provided in subresponses and committed to the database.

But, there are some fields for which Cloud API cannot match the value. For example, the values for `createTime` and `updateTime` cannot be determined prior to the commit. Fields of this type are always omitted from a subrequest's subresponse. But, they can be retrieved through a subselection.

Suppressing subresponse details

In some cases, a given object may be modified by multiple subrequests. This makes the intermediate subresponses unnecessary, and those subresponses can increase the size of the composite response unnecessarily and make the composite response harder to parse.

You can simplify the composite response by suppressing the amount of information returned for one or more subrequests. To do this, include the following with each relevant subrequest:

```
"includeResponse": false
```

For example:

```
{
  "requests": [
    {
      "method": "post",
      "uri": "/common/v1/activities/xc:202/notes",
      "body": {
        "data": {
          "attributes": {
            "body": "Cloud API note #1."
          }
        }
      },
      "includeResponse": false
    },
    ...
```

The composite response still includes a subresponse for the subrequest. But instead of providing the endpoint's default response, the subresponse appears as:

```
{
    "responseIncluded": false
},
```

The `responseIncluded` field defaults to true. If you want a detailed response for a given subrequest, simply omit the `responseIncluded` reference.

Specifying which fields to return

For a POST or PATCH subrequest, you can also refine which fields are returned. To do this, use the fields query parameter. The syntax for this is:

```
{
  "requests": [
    {
      "method": "<post/patch>",
      "uri": "<path>",
      "body": {
        "data": {
          "attributes": {
            "<field1>": "<value1>",
            "<field2>": "<value2>",
            ...
            }
        }
      },
      "parameters" : {
        "fields" : "<value>"
      }
    },
    ...
  ]
}
```

For example, the following code snippet creates an activity. For the subresponse, it specifies to include only the activity's ID and the assigned user.

```
{
  "requests": [
    {
      "method": "post",
      "uri": "/account/v1/accounts/pc:34/activities",
      "body": {
        "data": {
          "attributes": {
              "activityPattern": "contact_insured",
              "subject": "Cloud API activity"
          }
        }
      },
      "parameters" : {
        "fields" : "id,assignedUser"
      }
    },
  ...
```

The selections section

The `selections` section contains subselections that query for data. These are executed after the subselections in the requests section, and only if all the subrequests commit data successfully.

The basic syntax for the selections section is shown below. You do not need to specify a method for each subselection, as the only valid method in the selections section is GET.

```
  "selections": [
    {
      "uri": "<pathForFirstSubselection>"
    },
    {
      "uri": "<pathForSecondSubselection>"
    },
    ....
  ]
```

For example, the following code creates a new activity and a note for that activity. It then queries for the newly created activity.

```
{
  "requests": [
    {
      "method": "post",
      "uri": "/account/v1/accounts/pc:34/activities",
      "body": {
        "data": {
          "attributes": {
              "activityPattern": "contact_insured",
              "subject": "Cloud API activity"
          }
        }
      },
      "vars": [
        {
          "name": "newActivityId",
          "path": "$.data.attributes.id"
        }
      ]
    },
    {
      "method": "post",
      "uri": "/common/v1/activities/${newActivityId}/notes",
      "body": {
        "data": {
          "attributes": {
            "body": "Cloud API note #1."
          }
        }
      }
    }
  ],
  "selections": [
    {
      "uri": "/common/v1/activities/${newActivityId}
    }
```

```
]
}
```

For the complete syntax that includes all composite request features, see “Complete composite request syntax” on page 98.

Using query parameters in the selections section

You can use certain query parameters for each subselection. This includes:

- `fields`
- `filter`
- `includeTotal`
- `pageOffset`
- `pageSize`
- `sort`

Each subselection is independent from the others. You can use different query parameters for each subselection, and you can have some subselections with query parameters and others without query parameters.

The syntax for adding query parameters to a subselection is as follows:

```
"selections": [
  {
    "uri": "<pathForFirstQuery>",
    "parameters" : {
      "fields" : "<value>",
      "filter" : [<value>],
      "includeTotal" : <value>,
      "pageOffset" : <value>,
      "pageSize" : <value>,
      "sort" : [<value>]
    }
  },
  ....
]
```

Note the following:

- `fields` is specified as a single string of one or more fields, delimited by commas. The entire string is surrounded by quotes.
  - For example, `"assignedUser,dueDate,priority,subject"`
- `filter` and `sort` are stringified arrays consisting of one or more expressions. Each individual expression is surrounded by quotes. The list of expressions is then surrounded by [and].
  - For example: `["dueDate:gt:2022-12-20","status:in:open,complete"]`
- `includeTotal` , `pageOffset`, and `pageSize` are either Boolean or integer values, and therefore do not use quotes.

For example, when querying for activities, to return only the assigned user, due date, priority and subject fields:

```
{
  "uri": "/common/v1/activities",
  "parameters" : {
    "fields" : "assignedUser,dueDate,priority,subject"
  }
```

To return only open and complete activities with due dates after 2022-12-20:

```
{
  "uri": "/common/v1/activities",
  "parameters" : {
    "filter" : ["dueDate:gt:2022-12-20","status:in:open,complete"]      }
```

To return activities based on multiple criteria:

```
{
  "uri": "/common/v1/activities",
  "parameters" : {
    "fields" : "assignedUser,dueDate,priority,subject",
    "filter" : ["dueDate:gt:2022-12-20","status:in:open,complete"],
```

```
        "includeTotal" : true,
        "pageSize" : 5,
        "sort" : ["dueDate"]
      }
```

Composite requests that execute only queries

You can create a composite request that does not create or modify data and instead only queries for data. To do this, create a composite request with only a `selections` section and no `requests` section. In this case, the GETs in the `selections` section are always executed.

Responses to the selections subrequests

When a composite request contains a `selections` section, the response also contains a `selections` section. This section has the same structure as the `responses` section. It contains one subresponse for each subselection. Every subresponse has three sections:

- A `body` section, which by default contains the default response data defined in the corresponding endpoint.
- A `headers` section, which contains any custom headers.
- A `status` field, which indicates the subresponse's status code.

Error handling

If any of the subrequests in the `requests` section fail, Cloud API does the following:

- Does not commit any of the data
- Does not execute any GETs in the `selections` section
- Returns a 400 error

Cloud API also returns a response.

- The response begins with the following: `"requestFailed": true`. This is to make it easy to identify that the composite request did not commit data.
- For the initial subrequests that did not fail (if any), the response is returned.
  - This is either the response as specified by the associated endpoint (and any query parameters), or the `"responseIncluded": false` text.
  - The standard response can be useful for debugging purposes, as you can see the state of objects that succeeded before the subrequest that failed.
- For the subrequest that failed, the error message is returned.
- For the subrequests after the failed subrequest, the text `"skipped": true` is returned.
- If a `selections` section was included, the text `"skipped": true` is returned for each subselection.

For example, the following is the response for a composite request with five subrequests and a set of queries. All subrequests have `responseIncluded` set to false. The third subrequest failed because the `dueDate` attribute was incorrectly spelled as `ueDate`.

```
{
  "requestFailed": true,
  "responses": [
    {
      "responseIncluded": false
    },
    {
      "responseIncluded": false
    },
    {
      "requestError": {
        "details": [
          {
            "message": "Schema definition 'ext.common.v1.common_ext-
                1.0#/definitions/Note' does not define any property
                named 'ueDate'",
            "properties": {
              "lineNumber": 1,
              "parameterLocation": "body",
              "parameterName": "body"
            }
```

```
          }
        ],
        "developerMessage": "The request parameters or body had issues.
           See the details elements for exact details of the problems.",
        "errorCode": "gw.api.rest.exceptions.BadInputException",
        "status": 400,
        "userMessage": "The request parameters or body had issues"
      },
      "status": 400
    },
    {
      "skipped": true
    },
    {
      "skipped": true
    }
  ],
  "selections": [
    {
      "skipped": true
    }
  ]
}
```

If there is an error in the `selections` section only, the subrequests in the `requests` section will execute, the data will be committed, and the composite request will return with a 200 response code, indicating success. Cloud API also attempts to execute each subselection as best as possible.

Logging

Cloud API logs information about composite requests at both the composite request level and the subrequest/subselection level. This information appears in the logs under the `CompositeSubRequest` marker, which is a child of the normal `RestRequest` marker.

Each subrequest and subselection log message details information for that subrequest/subselection, such as the actual path, the HTTP method, and the apiFqn. The same log parameter names and conventions are used for both the parent request logging and the subrequest logging, such as whether a given value is logged with an empty string or omitted. However:

- Some parameters are always omitted at the subrequest because they only make sense for the parent request.
- Some additional composite-specific parameters are added.

For example suppose you executed the composite request listed earlier, in which a composite request creates two notes for activity xc:202. The corresponding log entries could appears as follows. Note that the first message is for the parent request and the next two messages are for the two subrequests:

```
server-id-207          749bdc4c-34b9-4f63-9b54-d1b0442af2c5 2021-12-13
14:40:30,803 INFO <RestService[ GW.PL ]> Operation started
{path="/composite/v1/composite", from="[0:0:0:0:0:0:0:1]", method="POST",
query="", startTime=1227026673066900, message="", eventType="START",
serverID="server-id-207"}

server-id-207 aapplegate 749bdc4c-34b9-4f63-9b54-d1b0442af2c5 2021-12-13
14:40:30,894 INFO <CompositeSubRequest[ RestRequest ]> Operation status
{path="/common/v1/activities/xc:202/notes", query="", method="POST",
pathTemplate="/activities/{activityId}/notes",
apiFqn="ext.common.v1.common_ext-1.0", status=201, error="", userMessage="",
devMessage="", elapsedTimeMs=53, message="Composite API subrequest
succeeded", eventType="STATUS", serverID="server-id-207"}

server-id-207 aapplegate 749bdc4c-34b9-4f63-9b54-d1b0442af2c5 2021-12-13
14:40:30,899 INFO <CompositeSubRequest[ RestRequest ]> Operation status
{path="/common/v1/activities/xc:202/notes", query="", method="POST",
pathTemplate="/activities/{activityId}/notes",
apiFqn="ext.common.v1.common_ext-1.0", status=201, error="", userMessage="",
devMessage="", elapsedTimeMs=4, message="Composite API subrequest
succeeded", eventType="STATUS", serverID="server-id-207"}
```

Each subrequest or subselection always generates a log statement, whether it succeeded, failed, or was skipped due to prior failures. A separate log statement is also added specifically for the commit of the composite request, whether that committed succeeded, failed, or was skipped.

Composite request limitations

Composite requests have the following general limitations:

- The number of subrequests and subselections in a single composite request must be less than or equal to the value of the `MaximumAllowedNumberOfCompositeSubRequests` configuration parameter. (In the base configuration, this is set to 100.)
- The subrequests can make use of other endpoints that are part of Cloud API. However, they cannot make use of endpoints outside of Cloud API, such as custom endpoints created by an insurer.
- You cannot include a subrequest that uses a content type other than application/json.
  - For example, you cannot work with document resources in composite requests, as documents use multipart/form-data.
- There is no mechanism for iterating over a set of things.
  - For example, you cannot start with a list of elements and include related resources for each item in that list.

There are also specific business requirements where you cannot use a composite request. For example:

- The only types of jobs you can create in a composite request are Submissions.
- You can modify other job types (such as Policy Change or Renewal) only if the job has already been created outside of the composite request.
- You cannot create new job versions.
- You cannot quote a job unless that job is a Submission created in the same Composite API request.
- You cannot bind-only or bind-and-issue in a composite request.

Many of the examples in the previous list pertain to situations where there must be an intermediate data commit, which composite requests do not allow by design. However, the previous list is not intended to be exhaustive. Refer to the section of the documentation that discusses each business requirement for more information on requirements or limitations related to composite requests.

Configuring the maximum number of subrequests

Composite requests are limited to a maximum number of subrequests and subselections. The maximum is specified by the `MaximumAllowedNumberOfCompositeSubRequests` configuration parameter. In the base configuration, this parameter is set to 100. Composite requests with more than the maximum number fail with a `BadInputException`. (The maximum applies to the sum of the number of subrequests and the number of subselections.)

The greater the number of subrequests in a composite request, the greater the chances that there will be a compromise in performance. A composite request with the maximum number of subrequests could result in a slow response, depending on what the maximum is and what those subrequests are doing.

In the base configuration, the maximum number of subrequests is 100. This can be raised to a greater value. However, composite requests with a significantly large number of subrequests could have negative consequences, such as:

- The request consuming a significant amount of service resources. This could include both memory and database resources.
- The request taking so long to complete that it times out before a response can be provided to the caller.

Consequently, Guidewire recommends setting the maximum number of subrequests to the lowest value that is needed. If there is a legitimate business need to raise the maximum above 100, Guidewire recommends the limit be raised only as much as is absolutely necessary.

Also, be aware that composite requests are subject to any application server limitations around the maximum size of a request body. Thus, it is possible for a composite request to be too large to process, even if the number of subrequests is at or below the allowed maximum.

Complete composite request syntax

The following is the complete syntax for a composite request:

```
{
  "requests": [
    {
      "method": "<post/patch/delete>",
      "uri": "<path>",
      "body": {
```

```
        "data": {
          "attributes": {
            "<field1>": "<value1>",
            "<field2>": "<value2>",
            ...
            }
        }
      },
      "parameters" : {
        "fields" : "<value>"
      },
      "vars": [
        {
          "name": "<name>",
          "path": "<path-starting-with-$>"
        },
        <next variable>,
        ...
      ],
      "includeResponse": false
    },
    {
      <next subrequest>
    },
    ...
    {
      <final subrequest>
    }
  ],
  "selections": [
    {
      "uri": "<pathForFirstQuery>",
      "parameters" : {
        "fields" : "<value>",
        "filter" : [<value>],
        "includeTotal" : <value>,
        "pageOffset" : <value>,
        "pageSize" : <value>,
        "sort" : [<value>]
      }
    },
    {
      <next subselection>
    },
    ...
    {
      <final subselection>
    }
  ]
}
```

chapter 9

Lost updates and checksums

This topic defines lost updates and discusses how you can prevent them through the use of checksums.

If you want to interact directly with the concepts in this topic, go to the following tutorials:

- “Tutorial: PATCH an activity using checksums” on page 103
- “Tutorial: Assign an activity using checksums” on page 104
- “Tutorial: DELETE a note using checksums” on page 105

Lost updates

Business processes often span multiple system API calls. When this occurs, the first call is typically either a GET that retrieves data or a POST that creates data. A later API call may attempt to modify the resource queried for or created in the initial GET or POST.

Some other process could potentially modify the resource between the GET/POST and the subsequent attempt to modify it. For example, suppose:

1. A caller application GETs activity xc:20. The activity's subject is "Contact additional insured" and the priority is Normal.
2. An internal user manually changes the subject of activity xc:20 to "Contact primary insured" and sets the priority to Urgent.
3. The caller application PATCHes activity xc:20 and sets the priority to Low.

The caller application's PATCH overwrites some of the changes made by the internal user. This could be a problem for several reasons:

- The caller application's change may be based on the data it initially retrieved. The caller application may not have initiated the change if it had known the subject or priority had later been changed by someone else.
- The internal user may not be aware that some of their changes were effectively discarded.
- The activity may now be in an inconsistent state (such as having a subject that is normally used for urgent activities and a priority of Low).

This type of modification is referred to as a lost update. Within the system APIs, a *lost update* is a modification made to a resource that unintentionally overwrites changes made by some other process. You can prevent lost updates through the use of checksums.

Checksums

A *checksum* is a string value that identifies the "version" of a particular resource. The system APIs calculate checksums as needed based on information about the underlying entities in the PolicyCenter database.

When a process modifies data, either through user action, system APIs, or other process, the system APIs calculate a different checksum for the resource. You can prevent lost updates by checking a resource's checksum before you modify the resource to see if it matches a previously retrieved checksum.

By default, checksums are provided in the response payloads of all GETs, POSTs, and PATCHes.

Checksums can be included in:

- Request payloads, which is appropriate for:
  - PATCHes
  - Business action POSTs that allow request payloads (such as POST `/{activityID}/assign`)
- Request object headers for:
  - DELETEs
  - Business action POSTs that do not allow request payloads

When you submit a request with a checksum, PolicyCenter calculates the checksum and compares that value to the submitted checksum value.

- If the values match, PolicyCenter determines the resource has not been changed since the caller application last acquired the data. The request is executed.
- If the values do not match, PolicyCenter determines the resource has been changed since the caller application last acquired the data. The request is not executed, and PolicyCenter returns an error similar to the following:

```
{
    "message": "The supplied checksum '1' does not match the current checksum '2' for the resource with uri '/
common/v1/notes/xc:101'",
    "properties": {
        "uri": "/common/v1/notes/xc:101",
        "currentChecksum": "2",
        "suppliedChecksum": "1"
    }
}
```

In many cases, checksums are simple integer values that are incremented with each update. However, this is not always the case. For some resources, the checksum is a more complicated string value, such as "fwer:3245-11xwj". Also, when a checksum is an integer, there is also no guarantee that the next checksum will be the integer value incremented by one. Guidewire recommends against caller applications attempting to predict what the next checksum value will be. Limit checksums in system API requests to only the checksums returned in previous responses.

Checksums for PATCHes and business action POSTs

For operations that have a request payload, checksums can be specified in the request payload. This applies to PATCHes and to most POSTs that execute business actions. (If a business action POST does not allow a request payload, you can still specify a checksum. But, you must do this in the request header. For more information, see “Checksums for DELETEs” on page 104.)

The `checksum` property is a child of the `data` property and a sibling of the `attributes` property. It uses the following syntax:

```
"checksum": "<value>"
```

For example, the following payload is for a PATCH to an activity. The payload specifies a new attribute value (setting `priority` to urgent) and a checksum value (2).

```
{
  "data": {
    "attributes": {
      "priority": {
```

```
        "code": "urgent"
      }
    },
    "checksum": "2"
  }
}
```

Checksums can be specified on the root resource and on any included resource. Specifying a checksum for any one resource does not require you to specify checksums for the others. For example:

- You could specify a checksum for only the root resource.
- You could specify a checksum for only one of the included resources.
- You could specify a checksum for the root resource and some of the included resources, but not all of the included resources.

Tutorial: PATCH an activity using checksums

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

In this tutorial, you will attempt to PATCH an activity twice. Both PATCHes will include a checksum value. The first PATCH will succeed, and the second will fail.

Tutorial steps

1. In Postman, start a new request by:
   a. Clicking the + to the right of the **Launchpad** tab
   b. Specifying *Basic Auth* authorization using user `aapplegate` and password `gw`.
2. Query for all activities by entering the following call and clicking **Send**:
   a. GET `http://localhost:8180/pc/rest/common/v1/activities`
3. Note the ID, subject, and checksum of the first activity returned in the response payload. (These values are referred to in later steps as "<ActivityID>", "<originalSubject>", and "<originalChecksum>".)
4. Start a second request by:
   a. Clicking the + to the right of the **Launchpad** tab
   b. Specifying *Basic Auth* authorization using user `aapplegate` and password `gw`.
5. Enter the following call, but do not click **Send** yet:
   a. PATCH `http://localhost:8180/pc/rest/common/v1/activities/<ActivityID>`
6. 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. For subject, specify the original subject with an additional "`-1`".

```
{
  "data": {
    "attributes": {
      "subject" : "<originalSubject>-1"
    }
  },
  "checksum": "<originalChecksum>"
}
```

7. Click **Send**. The checksum value in the payload matches the checksum value for the activity stored in PolicyCenter. So, the PATCH should be successful and the response payload should appear below the request payload.
8. Click **Send** a second time. Now, the checksum value in the payload does not match the checksum value for the activity calculated by PolicyCenter. So, the second PATCH is unsuccessful and an error message appears.

Tutorial: Assign an activity using checksums

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see “Tutorial: Set up your Postman environment” on page 21.

In this tutorial, you will attempt to execute a business action (assigning an activity) twice. Both attempts will include a checksum value. The first attempt will succeed, and the second will fail.

Tutorial steps

1. In Postman, query for all open activities by:
   a. Clicking the + to the right of the **Launchpad** tab.
   b. Specifying *Basic Auth* authorization using user `aapplegate` and password `gw`.
   c. Entering the following call and clicking **Send**:
      - GET `http://localhost:8180/pc/rest/common/v1/activities?filter=status:ne:complete`
2. Note the ID and checksum of the first activity returned in the response payload. (These values are referred to in later steps as "<ActivityID", and "<originalChecksum>".)
3. Start a second request by:
   a. Clicking the + to the right of the **Launchpad** tab
   b. Specifying *Basic Auth* authorization using user `aapplegate` and password `gw`.
4. Enter the following call, but do not click **Send** yet:
   a. PATCH `http://localhost:8180/pc/rest/common/v1/activities/<ActivityID>/assign`
5. The POST `/{activityId}/assign` endpoint requires a request payload that specifies how the assignment is to be done. 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. For subject, specify the original subject with an additional "-1".

```
{
  "data": {
    "attributes": {
        "autoAssign": true
    },
    "checksum": "<originalChecksum>"
  }
}
```

6. Click **Send**. The checksum value in the payload matches the checksum value for the activity stored in PolicyCenter. So, the POST `/assign` should be successful and the response payload should appear below the request payload.
7. Click **Send** a second time. Now, the checksum value in the payload does not match the checksum value for the activity calculated by PolicyCenter. (The successful POST `/assign` from the previous step will have modified the checksum value.) So, the second POST `/assign` is unsuccessful and an error message appears.

Checksums for DELETEs

For operations that do not permit a request payload, checksums can be specified in the request header. This applies to DELETEs and a small number of business action POSTs that do not permit request payloads.

The header key for a checksum is `GW-Checksum`. A checksum specified in the header applies only to the root resource.

Send a checksum in a request header using Postman

About this task

You can send checksums in request headers executed from Postman.

Procedure

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify authorization as appropriate.
3. Add the checksum to the header.
   - In the first row of tabs (the one that starts with **Params**), click **Headers**.
   - Scroll to the bottom of the existing key/value list.
   - In the blank row at the bottom of the key/value list, enter the following:
     - **KEY:** GW-Checksum
     - **VALUE:** <checksum value>
4. Enter the request operation and URL.
5. Click **Send**.

Results

The response appears below the request. Depending on the checksum value provided, the response will either include a success code or an error message.

Tutorial: DELETE a note using checksums

This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see "Tutorial: Set up your Postman environment" on page 21.

In this tutorial, you will send calls as Amy Clinton (user name `aclinton`). In the base configuration, Amy Clinton is an underwriting supervisor who has permission to delete notes. As Amy Clinton, you will create a note. You will then attempt to DELETE the note twice. Both DELETEs will include a checksum value. The first DELETE will fail, and the second will succeed.

Tutorial steps

1. In Postman, create an initial request by:
   a. Clicking the + to the right of the **Launchpad** tab.
   b. Specifying *Basic Auth* authorization using user `aclinton` and password `gw`.
2. Enter the following call, but do not click **Send** yet:
   a. POST `http://localhost:8180/pc/rest/common/v1/activities/pc:13/notes`
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": {
        "body": "API tutorial note to be deleted with a checksum"
      }
    }
}
```

4. Click **Send**. In the response payload, identify the note's id and checksum value.
5. Create a second request by:
   a. Clicking the + to the right of the **Launchpad** tab.
   b. Specifying *Basic Auth* authorization using user `aclinton` and password `gw`.
6. Enter the following call, but do not click **Send** yet:
   a. DELETE `http://localhost:8180/pc/rest/common/v1/notes/<noteID>`

7. Add the checksum to the header
   a. In the first row of tabs (the one that starts with **Params**), click **Headers**.
   b. Scroll to the bottom of the existing key/value list.
   c. In the blank row at the bottom of the key/value list, enter the following:
      - KEY: GW-Checksum
      - VALUE: 99
8. Click **Send**. The checksum value in the header does not match the checksum value for the note calculated by PolicyCenter. So, the DELETE is unsuccessful and an error message appears.
9. Change the checksum value so that it matches the one from the POST payload.
10. Click **Send** a second time. Now, the checksum value in the header matches the checksum value for the note calculated by PolicyCenter. So, the DELETE is successful.

chapter 10

Cloud API headers

This topic describes the Guidewire-proprietary headers supported by Cloud API.

HTTP headers

Request and response objects are used by REST APIs to send information between application. These objects contain HTTP headers. An *HTTP header* is a name/value pair included with a request or response object that provides metadata about the request or response. An HTTP header can specify information such as:

- The format used in the request object (such as whether it is JSON or XML)
- The format to use in the response object
- Session and connection information
- Authorization information

Overview of Cloud API headers

Cloud API supports standard HTTP headers, such as `Authorization` and `Content-Type`.

Cloud API also supports the following Guidewire-proprietary headers. Every Guidewire-proprietary header is optional.

Header	Datatype	Description
`GW-Checksum`	String	This can prevent lost updates. When specified, if the call would result in a database commit, then the API allows the commit only if the checksum in the header matches the checksum value from PolicyCenter. For more information, see "Checksums" on page 102.
`GW-DBTransaction-ID`	String of up to 128 characters	This can prevent duplicate requests. When specified, this is used as the database transaction ID for this request. The system API allows the commit only if the header's value has not be submitted by any prior request. The value is stored in the PolicyCenter database and must be globally unique across all clients, APIs and web services. For more information, see "Preventing duplicate database transactions" on page 109.

Header	Datatype	Description
`GW-DoNotCommit`	Boolean	This can be used to warm up endpoints. Typically, a caller application specifies this on a dummy POST that is sent prior to any genuine business requests. The POST triggers "warm up" activities for the endpoint, such as loading of Java and Gosu classes. But the header prevents any data from being committed. This request can improve the performance of subsequent requests to that endpoint. For more information, see "Warming up an endpoint" on page 110.
`GW-FailOnValidationWarnings`	Boolean	This can cause certain Job actions to fail if the action throws any validation warnings. If set to true on a request to quote, bind-only, or bind-and-issue a Job, this will cause the action to fail if there are any validation warnings. (Normally, these actions fail only if there are validation errors.) The default is false. For more information on Job actions, see "Policy transactions" on page 131.
`GW-IncludeSchemaProperty`	Boolean	This can modify the format of a JSON payload. When this is set to true, if the operation returns JSON with a defined schema, the `$GW-Schema` property is added to the root JSON object of the response with the fully-qualified name of the JSON Schema definition for that object. The default is false.
`GW-Language`	String	This sets the language used when processing the request. For more information, see "Globalization" on page 113.
`GW-Locale`	String	This sets the locale used when processing the request. For more information, see "Globalization" on page 113.
`GW-UnknownPropertyHandling`	One of these string values: • `log` • `reject` • `ignore`	This specifies the behavior for handling request payloads with unknown properties. The default behavior is `reject`. For more information, see "Handling a call with unknown elements" on page 110.
`GW-UnknownQueryParamHandling`	One of these string values: • `log` • `reject` • `ignore`	This specifies the behavior for handling URLs with unknown query parameters. The default behavior is `reject`. For more information, see "Handling a call with unknown elements" on page 110.
`GW-User-Context`	String	This provides information about the represented user when a service makes a service-for-user or service-for-service call. For more information, see the *Cloud API Authentication Guide*.
`GW-ValidateResponseHandling`	Boolean	Requests that the server performs additional validation of REST API responses against constraints such as `maxLength` that are declared in the schema. Disabled by default, but may be useful in some contexts for testing or debugging of custom APIs.

Header	Datatype	Description
		For more information, see “Validating response payloads against additional constraints” on page 111.
`x-gwre-session`	String	This controls how related calls are routed on instances of PolicyCenter running in a cluster. (Note: This header is not exclusive to Cloud API and therefore does not follow the convention of using "GW-" at the start of header names.) For more information, see “Routing related API calls in clustered environments” on page 31.
X-Correlation-ID	String	This permits a customer to trace a request from its initial reception through all of the subsequent applications that were invoked to handle that request. The actual traceability ID present in the MDC and logs (and returned in the response) is dependent on the implementation of `TraceabilityIDPlugin` plugin. The default implementation uses this value, if specified, or a generated UID. However, another implementation may always generate a unique ID and log the relationship between these incoming values and the generated UID. This header can be repeated, but the resulting string will just be the comma separated values. (Note: This header predates the REST API Framework and was created prior to the convention of using "GW-" at the start of header names.)

Send a request with a Cloud API header using Postman

About this task

You can include Cloud API headers in calls executed from Postman.

Procedure

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify authorization as appropriate.
3. Add the header and header value.
   - In the first row of tabs (the one that starts with **Params**), click **Headers**.
   - Scroll to the bottom of the existing key/value list.
   - In the blank row at the bottom of the key/value list, enter the header name in KEY column and its value in the VALUE column.
4. Enter the request operation and URL.
5. Click **Send**.

Preventing duplicate database transactions

In some circumstances, when a caller application is making a request that involves a commit to the database, the application may want to ensure that the request is processed only once. The caller application can do this using the `GW-DBTransaction-ID` header.

The `GW-DBTransaction-ID` header identifies a transaction ID (a string of up to 128 characters). When submitted with a system API call, the system API attempts to insert the value into the database's `TransactionID` table.

- If the value does not already exist in the table, the insert is successful. The system API assumes the transaction has not already been committed, and the call is processed as normal.
- If the value does exist in the table, the insert fails. The system API assumes the transaction has already been committed, and the call is rejected. The system API returns a 400 status code with an `gw.api.webservice.exception.AlreadyExecutedException` error.

For the call to success, the transaction ID specified in the header must be globally unique across all clients, APIs and web services.

The `GW-DBTransaction-ID` header has the following limitations:

- It only works with system APIs that commit data to the database.
- It only works when the system API commits a single time only. (System APIs that commit multiple times are rare.)
- It only works if the commit is either the only side effect of the call, or if the commit happens before any other side effects, such as the sending of notifications to external systems.

Duplicate requests do not return identical responses. The first request will succeed, but subsequent requests will fail. It is the responsibility of the caller application to decide how or if to handle this situation.

Warming up an endpoint

The first time a caller application makes a call to a Cloud API endpoint, the call may take longer to process than normal. This is because the Guidewire server may need to execute tasks for the first call that it does not need to re-execute for subsequent tasks, such as:

- Loading Java and Gosu classes
- Parsing and loading configuration files that are lazy-loaded on the first reference
- Loading data from the database or other sources into local caches
- Initializing database connections

A caller application may want to avoid having this slow processing time occur during a genuine business call. Therefore, the caller application may want to "warm up" the endpoint. This involves sending a dummy "warm-up request" to trigger these initial tasks. The warm-up request helps subsequent requests execute more rapidly.

Warm-up requests are not supposed to create or modify data. Theoretically, a caller application could use a GET as a warm-up request. However, GETs do not trigger as wide a range of start-up tasks as POSTs. The better option is to send a POST that does not commit any changes to the database. The best way to accomplish this is with a POST that contains the `GW-DoNotCommit` header. This header identifies that data modifications made by the request are to be discarded and not committed.

Best practices for warming up endpoints

Every endpoint makes use of different resources. Therefore, to warm up multiple endpoints, you need multiple requests. In general, the most effective warm-up request is a composite request with a large number of subrequests that POST to each endpoint you want to warm up.

For example, this could be a composite request where you create an account, and then a submission for the account, which you then quote and bind. This would include POSTs to other child objects as well, such as contacts, locations, coverages, and documents.

When executing a `GW-DoNotCommit` request, the response code will be the same as normal, such as 200 or 201, even though no data is committed. Caller applications need to be careful to ensure that there are no other undesired side effects from the warm-up request, such as integration points that might inadvertently send the dummy data downstream.

Handling a call with unknown elements

A system API call may include a payload that includes a property that is not defined in the associated schema. By default, the system APIs reject unknown properties. You can override the default behavior by including the `GW-UnknownPropertyHandling` header. The header must be set to one of the following string values:

- `ignore` - Ignore all unknown properties. Do not log any messages or return any validation errors.
- `log` - Log a service-side info message, but then process the call, ignoring any unknown properties.
- `reject` - Do not process the call. Return a validation error specifying there are unknown properties.

Similarly, a system API call may include a URL with a query parameter that is not defined in the associated schema. By default, the system APIs reject calls with unknown query parameters. You can override the default behavior by including the `GW-UnknownQueryParamHandling` header. The header must be set to one of the following string values:

- `ignore` - Ignore all unknown query parameters. Do not log any messages or return any validation errors.
- `log` - Log a service-side info message, but then process the call, ignoring any unknown query parameters.
- `reject` - Do not process the call. Return a validation error specifying there are unknown query parameters.

Validating response payloads against additional constraints

Serialization of the HTTP response is one of the final steps in handling a request. Both the response body and response headers need to be serialized, with the response body written to the `HttpServletResponse` output stream and the response headers turned into Strings that the servlet container is responsible for writing to the response. The system APIs support serialization of a number of different Java object types that can be returned directly from an API handler method, set as the value of the body of a Response object, or added as the value of a header on the Response object.

There are several types of response objects whose serialized format is JSON. This includes `JsonObject`, `JsonWrapper`, and `TransformResult`. By default, a `JsonObject` or `JsonWrapper` is validated only against the declared response schema to ensure that all properties on the object are declared in the schema and have the correct data type. `TransformResult` objects are "implicitly validated", given that the mapping file that produces them must conform to the associated JSON schema.

It is possible to request that the framework also validate a `JsonObject`, `JsonWrapper`, or `TransformResult` against additional constraints defined in the schema, such as `minLength`, the set of required fields, or any custom validators that have been defined. These additional validations are not done by default because they can potentially be an unnecessary expense in a production situation where the assumption is that the API has been implemented correctly and will only return valid data. It is also possible that the constraints defined in the schema are intended to only apply to inputs, and that the response may violate some of them.

You can use the `GW-ValidateResponseHandling` header to have the system API validate its responses against the declared schema. To do this, include the header and set its value to true.

chapter 11

Globalization

In the context of Guidewire InsuranceSuite applications, globalization refers to the internationalization and localization aspects of system configuration. The system APIs can work with the globalization settings of your system. For details on how Guidewire InsuranceSuite applications support globalization, refer to the *Globalization Guide*.

Specifying language and locale in API requests

By default, system API calls return data in the format of the default language and locale of your PolicyCenter instance, as specified by the `DefaultApplicationLanguage` and `DefaultApplicationLocale` system parameters in `config.xml`. If your instance supports additional languages and locales, then you can construct API calls to request data in those alternative formats.

Callers can specify a preferred language and locale in the request header. Guidewire provides two header fields for this purpose, `GW-Language` and `GW-Locale`. The `GW-Language` field accepts an ISO 639-1 code designating the language, while the `GW-Locale` field takes the ISO 639-1 language code along with the ISO 3166-1 alpha-2 locale code, separated by an underscore.

For example, the ISO 639-1 language code for Japanese is `ja`, and the ISO 3166-1 alpha-2 locale code for Japan is `JP`. The following code block displays a request header with the `GW-Language` and `GW-Locale` fields set to Japanese language and locale, respectively:

```
GET /pc/rest/account/v1/accounts/pc:102? HTTP/1.1
Host: localhost:8180
GW-Language: ja
GW-Locale: ja_JP
Authorization: Basic c3U6Z3c=
```

Addresses and locales

The formatting of postal addresses can vary by country. PolicyCenter provides a flexible way to format addresses using the Address entity along with the State and Country typelists. In the system APIs, this address data is mapped to the Address schema found in the Common API.

The following table lists each Address property with its associated Guidewire Address entity field:

Address property	GW entity mapping	Description
`addressLine1`	Address.AddressLine1	First line of a street address
`addressLine1Kanji`	Address.AddressLine1Kanji	First line of a street address (in Japanese)

Address property	GW entity mapping	Description
addressLine2	Address.AddressLine2	Second line of a street address
addressLine2Kanji	Address.AddressLine2Kanji	Second line of a street address (in Japanese)
addressLine3	Address.AddressLine3	Third line of a street address
area	Address.State	Country-specific administrative area, as defined in the State typelist
city	Address.City	City or locality
cityKanji	Address.CityKanji	City or locality (in Japanese)
country	Address.Country	Country code, as defined in the Country typelist
county	Address.State	Country-specific administrative area, as defined in the State typelist
department	Address.State	Country-specific administrative area, as defined in the State typelist
district	Address.State	Country-specific administrative area, as defined in the State typelist
do_si	Address.State	Country-specific administrative area, as defined in the State typelist
emirate	Address.State	Country-specific administrative area, as defined in the State typelist
island	Address.State	Country-specific administrative area, as defined in the State typelist
oblast	Address.State	Country-specific administrative area, as defined in the State typelist
parish	Address.State	Country-specific administrative area, as defined in the State typelist
postalCode	Address.PostalCode	Postal or zip code
prefecture	Address.State	Country-specific administrative area, as defined in the State typelist
province	Address.State	Country-specific administrative area, as defined in the State typelist
sortingCode	Address.CEDEXBureau	Sorting code (France only)
state	Address.State	Country-specific administrative area, as defined in the State typelist

Address locale configuration

While the Address schema supports a wide range of properties for locale-specific administrative areas, only one such property can be used in a given address. For example, an address cannot use both `state` and `province` properties. Furthermore, only one administrative area property is valid in an address, and this is determined by the country or territory of the address. Studio provides an address configuration file at `configuration/config/Integration/i18n/addresses.i18n.yaml`:

```
countries:
  . . .
  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
  . . .
```

The `countries` field contains a property for each country or region, the name of which is derived from the relevant ISO 3166-1 alpha-2 country code. The previous code block displays two such properties, `JP` and `US`, for Japan and the United States, respectively. Each country property contains the following fields and values:

- `name`: The name of the region (typically country or territory)
- `addressFields`: The address fields from the Address schema that can be included in an address for the country or region
- `addressRequire`: The minimum subset of address fields that must be included in an address for the country or region

When starting the server, the `addresses.i18n.yaml` file is loaded, and its rules are applied to Address resources. The Address schema contains code that enables this functionality:

```
"Address": {
  "type": "object",
  "x-gw-extensions": {
    "discriminatorProperty": "country"
  },
  "properties": {
    . . .
    }
  }
}
```

In the previous code snippet, the `x-gw-extensions.discriminatorProperty` field is set to `country`. As a result, when setting the country property on an Address resource, the address fields associated with that country will be valid for that resource, and the fields not associated with the country will be unavailable.

part 2

Business flows: PolicyCenter

The *InsuranceSuite Cloud API* is a set of RESTful system APIs that expose functionality in PolicyCenter so that caller applications can request data from or initiate action within PolicyCenter.

The following topics discuss how caller applications can initiate specific PolicyCenter business flows or interact with specific types of PolicyCenter resources. This includes:

- Creating accounts
- Working with submissions
- Working with renewals
- Working with cancellations
- Working with policy changes
- Working with reinstatements
- Working with rewrites

Preemptions

A *preemption* is an event that occurs when two or more jobs are created from the same base policy period. The first job to be bound preempts the second job. If you want to continue with the second job, it must be modified to handle the changes made by the first job.

For example, suppose that there is a bound personal auto policy with a single car. A policy change is started to add a motorcycle to the policy. Then, while the first job is still in progress, a second policy change is started that adds a van to the policy. The motorcycle job has no knowledge of the van, and the van job has no knowledge of the motorcycle. If the motorcycle job is bound first, it will preempt the van job. The policy now has both a car and a motorcycle on it. The van job was only aware of the car. Before the van policy change can be completed, it must be modified to include the motorcycle.

PolicyCenter provides support for working with preemptions. PolicyCenter identifies when a job has been preempted. It provides a list of information on the preempting job that is not present in the preempted job. It also provides "handle preemption" functionality that copies information from the preempting job over to the preempted job. For more information, see the *Application Guide*.

Similarly, Cloud API provides support for preemptions. For a preempted job, you can view information about the preempting job and execute "handle preemption" functionality. Similarly, when a policy change is bound and there is an existing renewal for the policy, you can apply changes from the policy change to the renewal.

Viewing preemption information

Use the following endpoint to view information about a job's preemptions:

- GET `/jobs/{jobId}/preemptions`

The `/preemptions` endpoint returns two main types of information:

- An array of job diffs.
- Information about the preempting job.

The diffs array

The `diffs` array identifies all of the changes between the base job and the preempting job. These are the changes that are not in the preempted job but would need to be for the preempted job to be bindable.

Each member of the array identifies one difference between the base job and the preempting job. This includes the object (`entity`), the label of the diff field (`field`), the base job value (`existingValue`) and the preempting job value (`changedValue`). The following code shows the syntax of each job diff array member.

```
{
  "changedValue": <value>,
  "entity": {
    "displayName": <display name>,
    "id": <id>
  },
  "existingValue": <value>,
  "field": <value>
}
```

The difference can pertain to the value of a single field. For example, the following payload indicates that on a Medical Payments coverage object, the `choiceTerm1` field in the base job was 5000, but the preempting job changed it to 15000.

```
{
  "changedValue": "15,000",
  "entity": {
    "displayName": "Medical Payments",
    "id": "85"
  },
  "existingValue": "5,000",
  "field": "choiceTerm1"
}
```

The difference can also pertain to the addition or removal of an object. When this occurs, a "√" is used to indicate the object is present, and a "" (an empty string) is used to indicate the object is absent. For example, the following payload indicates that a 2000 Honda Civic was not present in the base job but was present in the preempting job. (In other words, the preempting job added a Honda Civic.)

```
{
  "changedValue": "√",
  "entity": {
    "displayName": "2000 Honda Civic in California",
    "id": "32"
  },
  "existingValue": "",
  "field": "2000 Honda Civic in California"
},
```

The preempting job

The payload also includes information about the preempting job, including:

- The job itself

- The job's effective date
- The job number
- The job type

Preemption payload example

For example, suppose that there is a personal auto policy with an Acura Integra. At roughly the same time, two policy changes are started:

- First policy change:
  - Job number: 0004375724
  - Job id of pc:421
  - Effective date: 11/05/2021
  - Material changes: Adds a Honda Civic and associated coverages for driver Alicia Shirley
- Second policy change:
  - Job number: 0004408964
  - Job id of pc:505
  - Effective date: 11/06/2021
  - Material changes: Adds a Toyota Prius and associated coverages for driver Alicia Shirley

The first policy change is bound, causing that job to preempt the second job. As a result, the second job now has a preemption.

The following shows a portion of the response payload for a GET `/preemptions` for job pc:505. Note that the array of diffs in the payload contains the following information.

- The effective date for the preempting job
- The vehicle for the preempting job
- The vehicle's collision coverage for the preempting job
- The vehicle's comprehensive coverage for the preempting job
- The vehicle driver for the preempting job

The payload also contains information about the preempting job.

```
GET /job/v1/jobs/pc:505/preemptions

{
  "count": 1,
  "data": [
    {
      "attributes": {
        "diffs": [
          {
            "changedValue": "11/05/2021",
            "entity": {
              "displayName": "4509454228, 11/04/2021, 05/04/2022, 0004375724",
              "id": "pc:421",
              "type": "Job",
              "uri": "/job/v1/jobs/pc:421"
            },
            "existingValue": "11/04/2021",
            "field": "writtenDate"
          },
          {
            "changedValue": "√",
            "entity": {
              "displayName": "2000 Honda Civic in California",
              "id": "32"
            },
            "existingValue": "",
            "field": "2000 Honda Civic in California"
          },
          {
            "changedValue": "√",
            "entity": {
              "displayName": "Collision",
              "id": "64"
            },
            "existingValue": "",
            "field": "Collision"
```

```
          },
          {
            "changedValue": "v",
            "entity": {
              "displayName": "Comprehensive",
              "id": "63"
            },
            "existingValue": "",
            "field": "Comprehensive"
          },
          {
            "changedValue": "v",
            "entity": {
              "displayName": "Alicia Shirley",
              "id": "15"
            },
            "existingValue": "",
            "field": "Assigned Driver: Alicia Shirley"
          }
        ],
        "job": {
          "displayName": "0004375724",
          "id": "pc:421",
          "type": "Job",
          "uri": "/job/v1/jobs/pc:421"
          },
        "jobEffectiveDate": "2021-11-05T00:01:00.000Z",
        "jobNumber": "0004375724",
        "jobType": {
          "code": "PolicyChange",
          "name": "Policy Change"
        }
      },
...
```


Handling preemptions

Once a job has been preempted, it can no longer be bound in its current state. This is because the base policy period has been modified by the preempting job, and the preempted job is now missing information present in the base policy period.

PolicyCenter provides the ability to handle preemptions. This is an action you can take on a preempted job to copy over the changes made by the preempting job. Handling a preemption brings a preempted job in line with the new version of the base period. Once you have handled preemptions for a job, you can bind it.

To handle preemptions through Cloud API, use the following endpoint:

- POST `/job/v1/jobs/{jobId}/handle-preemptions`

You do not need to provide a request body.

Typically, handling a preemption modifies the contents of a job. Therefore, you need to quote the job prior to binding it, even if you quoted the job prior to the preemption.

Example of handling preemptions

In the previous topic, there was an example of a personal auto policy with an Acura Integra. At roughly the same time, two policy changes are started:

- First policy change:
  - Job number: 0004375724
  - Job id of pc:421
  - Effective date: 11/05/2021
  - Material changes: Adds a Honda Civic and associated coverages for driver Alicia Shirley
- Second policy change:
  - Job number: 0004408964
  - Job id of pc:505
  - Effective date: 11/06/2021
  - Material changes: Adds a Toyota Prius and associated coverages for driver Alicia Shirley

The first policy change is bound, causing that job to preempt the second job. As a result, the second job now has a preemption.

Now, suppose you want to handle preemptions for the second job and then bind it. (For clarity sake, portions of the response payload have been omitted in the following examples.)

First, you must handle preemptions.

```
POST /job/v1/jobs/pc:505/handle-preemptions

{
  "data": {
    "attributes": {
      "job": {
        "id": "pc:505",
        "isPreempted": false,
        "jobNumber": "0004408964",
        "jobStatus": {
            "code": "Draft",
            "name": "Draft"
          },
        "jobType": {
          "code": "PolicyChange",
          "name": "Policy Change"
        }
      }
    }
  }
}
```

Next, you must quote the job.

```
POST /job/v1/jobs/pc:505/quote

{
  "data": {
    "attributes": {
      "id": "pc:505",
      "isPreempted": false,
      "jobNumber": "0004408964",
      "jobStatus": {
        "code": "Quoted",
        "name": "Quoted"
      },
      "jobType": {
        "code": "PolicyChange",
        "name": "Policy Change"
      },
      "taxesAndSurcharges": {
        "amount": "94.00",
        "currency": "usd"
      },
      "totalCost": {
        "amount": "1396.00",
        "currency": "usd"
      },
      "totalPremium": {
        "amount": "1302.00",
        "currency": "usd"
      }
    }
  }
}
```

And finally, you can bind and issue the job.

```
POST /job/v1/jobs/pc:505/bind-and-issue

{
  "data": {
    "attributes": {
      "id": "pc:505",
      "isPreempted": false,
      "jobNumber": "0004408964",
      "jobStatus": {
        "code": "Bound",
        "name": "Bound"
      },
      "jobType": {
        "code": "PolicyChange",
        "name": "Policy Change"
      }
    }
```

```
  }
}
```

Applying changes to a renewal

It is possible for a policy change to preempt a future renewal whose state is either draft or bound. When the policy change is bound, the user interface provides the option of applying the changes in the preempting job to the renewal. This is similar to "handle preemption" functionality, but there are differences:

- The copying of information is initiated from the preempting job, not the preempted job.
- The copying of information is optional. If you choose to not copy the information, you can still bind the renewal.

To apply changes from a policy change to a renewal through Cloud API, use the following endpoint:

- POST `job/v1/jobs/{jobId}/apply-changes-to-renewal`

You do not need to provide a request body.

Once the changes are applied to the renewal, you need to quote (or requote) the renewal before it can be bound.

Identifying when a renewal exists

When you execute a policy change from the PolicyCenter user interface, there is a button that identifies a renewal exists and that gives you the option to copy changes to the renewal.

When you execute a policy change from Cloud API, you can identify whether there is an existing renewal by checking the `links` array of the response payload. If there is an `apply-changes-to-renewal` member, the job's policy has a renewal and you may want to consider applying the policy changes to that renewal.

For example, the following response payload snippet contains an `apply-changes-to-renewal` member, indicating the changed policy has a renewal.

```
"links": {
  ..
  "activity-patterns": {
    "href": "/job/v1/jobs/pc:232/activity-patterns",
    "methods": [
      "get"
    ]
  },
  "apply-changes-to-renewal": {
    "href": "/job/v1/jobs/pc:232/apply-changes-to-renewal",
      "methods": [
        "post"
      ]
  },
  "contacts": {
    "href": "/job/v1/jobs/pc:232/contacts",
      "methods": [
        "get"
      ]
  },
  ...
```

chapter 12

Accounts

In PolicyCenter, an account is associated with an individual or company, and is typically created when quoting a policy for a new prospect. At a minimum, an account must have the name of the account holder, basic location information for the account holder, and information about the producer associated with the account.

Creating an account

To create a new account in PolicyCenter, a caller can use the POST `/account/v1/accounts` endpoint from the Account API. When creating an account, the caller must provide the following data in the request payload:

- An account holder
- A primary location
- A producer code

A caller can provide additional account information in the request payload, as defined by the schema for the `Account` resource.

Name clearance

Name clearance is the process of checking against one or more producer/account databases to ensure that a person or company is not an existing account in PolicyCenter. Because name clearance requirements differ across insurers, it is not implemented in the system APIs directly. Implementors of services that require name clearance can configure that functionality in their own code.

Account holder

The account holder indicates the primary contact that will be associated with the account. An account holder can be an individual or a company, and this information must be specified in the request. Additionally, the request must also include a name and address for the account holder.

On account creation, the account status is set to Pending. This indicates that the account exists but that there are no policies associated with it.

Calls to POST `/account/v1/accounts` are not idempotent. A caller can create multiple new accounts for the same person or company.

Setting an account holder

To set the account holder, the caller can use the `initialAccountHolder` attribute in the request payload. Alternatively, the caller can use the `accountHolder` attribute in tandem with an include containing an `AccountContact` resource. This latter approach involves request inclusion. For more information, see “Request inclusion” on page 80.

When setting an account holder, the caller must provide values for the `contactSubtype`, `primaryAddress.addressLine1`, `primaryAddress.city`, `primaryAddress.state`, and `primaryAddress.postalCode` fields. If the account is for an individual, then the caller must also include the `firstName` and `lastName` fields. If the account is for a company, then the caller must include the `companyName` field.

In the following example, the necessary fields for an individual account holder are attached to the `initialAccountHolder` attribute:

```
{
  "data": {
    "attributes": {
      "initialAccountHolder": {
        "contactSubtype": "Person",
        "firstName": "Bill",
        "lastName": "Preston",
        "primaryAddress": {
          "addressLine1": "2850 S. Delaware St.",
          "city": "San Mateo",
          "postalCode": "94403",
          "state": {
            "code": "CA"
          }
        }
      },
      . . .
    }
  }
}
```

The next example demonstrates how to set the account holder for the same individual, using the `accountHolder` attribute with an `AccountContact` include:

```
{
  "data": {
    "attributes": {
      "accountHolder": {
        "refid": "newperson"
      },
      . . .
    }
  },
  "included": {
    "AccountContact": [
      {
        "attributes": {
          "contactSubtype": "Person",
          "firstName": "Bill",
          "lastName": "Preston",
          "primaryAddress": {
            "addressLine1": "2850 S. Delaware St.",
            "city": "San Mateo",
            "postalCode": "94403",
            "state": {
              "code": "CA"
            }
          }
        },
        "method": "post",
        "refid": "newperson",
        "uri": "/account/v1/accounts/this/contacts"
      }
    ]
  }
}
```

The next example illustrates how to set the account holder for a company, using the `initialAccountHolder` attribute:

```
{
  "data": {
    "attributes": {
      "initialAccountHolder": {
        "contactSubtype": "Company",
        "companyName": "Preston, Inc.",
        "primaryAddress": {
          "addressLine1": "2850 S. Delaware St.",
          "city": "San Mateo",
          "postalCode": "94403",
          "state": {
            "code": "CA"
          }
        }
      },
```

```
      . . .
    }
  }
}
```

Primary location

The primary location indicates the primary address that will be associated with the account.

To set the primary account location, the caller can use the `initialPrimaryLocation` attribute in the request payload. Alternatively, the caller can use the `primaryLocation` attribute in tandem with an include containing an `AccountLocation` resource. This latter approach involves request inclusion. For more information, see "Request inclusion" on page 80.

When creating an account, the account location can be specific or non-specific. A specific account location must include the street address, city, state or province, and postal code. A non-specific account location must include the state or province at minimum, as this is necessary to determine several locale-related constraints for the account.

Setting a specific account location

When setting a specific location, the caller must provide values for the `addressLine1`, `city`, `state`, and `postalCode` fields. In the following example, these fields are attached to the `initialPrimaryLocation` attribute:

```
{
  "data": {
    "attributes": {
      . . .
      "initialPrimaryLocation": {
        "addressLine1": "2850 S. Delaware St.",
        "city": "San Mateo",
        "state": {
          "code": "CA"
        },
        "postalCode": "94403"
      }
    }
  }
}
```

The next example accomplishes the same thing with request inclusion, using the `primaryLocation` attribute with an `AccountLocation` include:

```
{
  "data": {
    "attributes": {
      . . .
      "primaryLocation": {
        "refid": "newlocation"
      }
    }
  },
  "included": {
    "AccountLocation": [
      {
        "attributes": {
          "addressLine1": "2850 S. Delaware St.",
          "city": "San Mateo",
          "postalCode": "94403",
          "state": {
            "code": "CA"
          }
        },
        "method": "post",
        "refid": "newlocation",
        "uri": "/account/v1/accounts/this/locations"
      }
    ]
  }
}
```

Setting a non-specific account location

When setting a non-specific location, the caller must provide values for the `state` and `nonSpecific` fields. In the following example, these fields are attached to the `initialPrimaryLocation` attribute:

```
{
  "data": {
    "attributes": {
      . . .
      "initialPrimaryLocation": {
        "nonSpecific": true,
        "state": {
          "code": "CA"
        }
      }
    }
  }
}
```

The next example accomplishes the same thing with request inclusion, using the `primaryLocation` attribute with an `AccountLocation` include:

```
{
  "data": {
    "attributes": {
      . . .
      "primaryLocation": {
        "refid": "newlocation"
      }
    }
  },
  "included": {
    "AccountLocation": [
      {
        "attributes": {
          "nonSpecific": true,
          "state": {
            "code": "CA"
          }
        },
        "method": "post",
        "refid": "newlocation",
        "uri": "/account/v1/accounts/this/locations"
      }
    ]
  }
}
```

Producer code

At creation time, an account must be associated with one producer code. This is a PolicyCenter requirement, and cannot be circumvented by the system APIs.

The request payload must include a `producerCodes` attribute. This is an array of one object that contains an `id` field for the producer code:

```
{
  "data": {
    "attributes": {
      . . .,
      "producerCodes": [
        {
          "id": "pc:6"
        }
      ]
    }
  }
}
```

Example: Creating an account

The following code sample depicts a complete request payload that can be POSTed to the `/account/v1/accounts` endpoint. The example includes the optional properties `preferredCoverageCurrency` and `preferredSettlementCurrency`. The account holder and primary location properties are applied through request inclusion. For details on request inclusion, see “Request inclusion” on page 80.

Request payload for creating a new account using request inclusion

```
{
  "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": "Preston",
          "primaryAddress": {
            "addressLine1": "2850 S Delaware St #400",
            "city": "San Mateo",
            "postalCode": "94403",
            "state": {
              "code": "CA"
            }
          }
        },
        "method": "post",
        "refid": "newperson",
        "uri": "/account/v1/accounts/this/contacts"
      }
    ],
    "AccountLocation": [
      {
        "attributes": {
          "locationCode": "0001",
          "locationName": "Location 0001",
          "nonSpecific": true,
          "postalCode": "94403",
          "state": {
            "code": "CA"
          }
        },
        "method": "post",
        "refid": "newloc",
        "uri": "/account/v1/accounts/this/locations"
      }
    ]
  }
}
```

Account search

Through the Account API, a caller can use the `/acccount/v1/search/accounts` endpoint to search for accounts. To execute a search, a caller can submit a POST request to the search endpoint. The request payload must contain one of the following items:

- `accountNumber`: Account number
- `companyName`: Complete company name. When searching by company name, do not include personal name data.
- `firstName` and `lastName`: Complete first name and last name of the account holder. When searching by account holder name, both fields must be included in the request. When searching by personal name, do not include company name data.
- `organization.id`: Company identification number

- `phoneNumber`: Telephone number
- `producerCode.id`: Producer code
- `taxId`: Tax identification number

Note:

When searching by personal name or company name, the complete name must be entered. Partial matching is not supported. Also, searching by address is not supported.

The following code block is a request body for a search based on a personal name:

```
{
  "data": {
    "attributes": {
      "firstName": "Ray",
      "lastName": "Newton"
    }
  }
}
```

A successful search request returns an array of account objects. For example, a search using the request body above returns the following:

```
{
    "count": 1,
    "data": [
        {
            "attributes": {
                "accountHolder": {
                    "displayName": "Ray Newton",
                    "id": "test_pp:2"
                },
                "accountNumber": "C000143542",
                "accountStatus": {
                    "code": "Active",
                    "name": "Active"
                },
                "businessOperationsDescription": "business description",
                "createdDate": "2021-04-10T22:12:13.688Z",
                "frozen": false,
                "id": "pc:2",
                "industryCode": {
                    "code": "0740",
                    "description": "Veterinarians / Veterinarian / Veterinary Services",
                    "id": "SIC:0740"
                },
                "numberOfContacts": "8",
                "organizationType": {
                    "code": "commonownership",
                    "name": "Common ownership"
                },
                "preferredCoverageCurrency": {
                    "code": "usd",
                    "name": "USD"
                },
                "preferredSettlementCurrency": {
                    "code": "usd",
                    "name": "USD"
                },
                "primaryLanguage": {
                    "code": "en_US",
                    "name": "English (US)"
                },
                "primaryLocale": {
                    "code": "en_US",
                    "name": "United States (English)"
                },
                "primaryLocation": {
                    "displayName": "4: 1253 Paloma Ave, Arcadia, CA",
                    "id": "pc:749",
                    "type": "AccountLocation",
                    "uri": "/account/v1/accounts/pc:2/locations/pc:749"
                },
                "producerCodes": [
                    {
                        "displayName": "100-002541",
                        "id": "pc:6"
                    }
                ]
            },
            . . .
}
```

The search endpoint also supports certain locale-specific fields:

- `companyNameKanji`: Complete company name, in Japanese
- `firstNameKanji`: Complete first name, in Japanese. Can be used in combination with `firstName` or `lastNameKanji.`
- `lastNameKanji`: Complete last name, in Japanese. Can be used in combination with `lastName` or `firstNameKanji`.
- `particle`: Particle used in account holder's name (matching the `particle` property of an AccountContact resource)

The following code block is a request body for a search based on first name and first name in Japanese:

```
{
  "data": {
    "attributes": {
      "firstNameKanji": "太郎",
      "firstName": "Taro"
    }
  }
}
```

chapter 13

Policy transactions

The system APIs support the following policy transactions:

- Submission
- Issuance
- Renewal
- Cancellation
- Policy change
- Reinstatement
- Rewrite
- Rewrite new account

Transactions not in this list are not supported at this time.

When using the system APIs, policy transactions are conducted through the Policy and Job APIs. The Policy API provides endpoints for initiating policy transactions on existing policies. The Job API provides endpoints for modifying, quoting, and completing policy transactions, as well as for creating new policies (submissions). The LOB-specific aspects of policy transactions are encapsulated in the Job API.

> **Note:** The base configuration of the system APIs does not include LOB-specific APIs. Clients must generate LOB-specific APIs using Advanced Product Designer (APD). Once generated, the LOB-specific API endpoints are available through the Job API. For details, see “Generate LOB-specific APIs” on page 256.

Conceptually, a policy transaction entails four basic steps:

1. Initiate the policy transaction through a business action POST.
   - This step creates a new job element, on which subsequent actions will be applied.
   - The new job is in Draft state, and thus can be modified.
2. Modify the job as needed.
   - Depending on the policy transaction type, a number of calls might be required to apply all necessary data to the job.
3. Quote the job.
   - This step changes the job from Draft to Quoted state.
   - Jobs in Quoted state cannot be modified. To further modify a quoted job, it must be changed to Draft state, in which case the process then reverts to step 2.
4. Complete the policy transaction by finalizing the job.
   - A quoted job can be accepted (bound) or declined.

Initiating the policy transaction

To initiate a policy transaction, a caller submits a business action POST to an appropriate endpoint. The following table lists the policy transaction type with its relevant endpoint:

Policy transaction	Endpoint
Submission	`/job/v1/submissions`
Issuance	`/policy/v1/policies/{policyId}/issue`
Renewal	`/policy/v1/policies/{policyId}/renew`
Cancellation	`/policy/v1/policies/{policyId}/cancel`
Policy change	`/policy/v1/policies/{policyId}/change`
Reinstatement	`/policy/v1/policies/{policyId}/reinstate`
Rewrite	`/policy/v1/policies/{policyId}/rewrite`
Rewrite new account	`/policy/v1/policies/{policyId}/rewrite-account`

All policy transactions other than submissions are initiated from an existing policy element through a Policy API call. The submission policy transaction is initiated using the Job API. In either case, the call returns a response payload that contains a JSON object for a new job element that is accessible through the Job API. Subsequent actions for the policy transaction are executed on this job element.

The following example depicts portions of a response payload returned by a call to initiate a submission policy transaction for a personal auto line:

```
{
    "data": {
        "attributes": {
            "account": {
                "displayName": "0015863842",
                "id": "pc:401",
                "type": "Account",
                "uri": "/account/v1/accounts/pc:401"
            },
            . . .
            "id": "pc:601",
            . . .
            "jobStatus": {
                "code": "Draft",
                "name": "Draft"
            },
            "jobType": {
                "code": "Submission",
                "name": "Submission"
            },
            . . .
            "product": {
                "displayName": "Personal Auto",
                "id": "PersonalAuto"
            },
            . . .
        },
        . . .
        }
    }
}
```

- The `id` property contains the ID of the new job element, in this case `pc:601`.
- The `account.id` property is the ID for the account holder. It is important to not mistake `account.id` for `id`.
- The `jobStatus` property indicates that the job is in Draft state.

- The `jobType` property shows that the job is for a submission policy transaction.

Subsequent actions for this policy transaction are executed on the `/job/v1/jobs/pc:601` element.

Modifying the job

As stated previously, initiating a policy transaction produces a job element that is accessible through the Job API (`/job/v1/jobs/{jobId}`). All subsequent work involved in processing a policy transaction occurs on this job element.

Depending on the policy transaction type, it might be necessary to modify the job by submitting additional data. For example, after initiating a submission policy transaction for a personal auto line, a caller must modify the job by adding vehicles and drivers and applying coverages.

Quoting the job

For any policy transaction, the associated job must be quoted before it can be completed. To generate a quote, a caller submits a business action POST to `/job/v1/jobs/{jobId}/quote`. This action changes the job to Quoted state. If subsequent modifications are deemed necessary, then the job must be reverted to Draft state by submitting a business action POST to `/job/v1/jobs/{jobId}/make-draft`. After modifications are applied, the job must be quoted again.

Completing the policy transaction

To complete the policy transaction, the job must be accepted (bound) or rejected.

For accepted policy transactions, in most cases the transaction can be completed by submitting a business action POST to the `/job/v1/jobs/{jobId}/bind-and-issue` endpoint. This action binds the transaction to the policy.

A policy transaction can be rejected for a number of reasons. If the insurer rejects the transaction, then the job can be rescinded through a business action POST to the `/job/v1/jobs/{jobId}/decline`. If the policy transaction contains faulty data, then it can be discarded through a business action POST to the `/job/v1/jobs/{jobId}/withdrawn` endpoint.

Transaction outcome	Condition	Job status	Endpoint
Accepted	The account holder and insurer agree to the quote. The policy is bound and issued.	Bound	`/job/v1/jobs/{jobId}/bind-and-issue`
Rejected	The insurer declines to offer a policy to the account holder.	Declined	`/job/v1/jobs/{jobId}/decline`
Rejected	The quote contains erroneous data.	Withdrawn	`/job/v1/jobs/{jobId}/withdrawn`

Furthermore, some policy transaction types can be completed by being placed in Pending status.

The specific actions available to complete a policy transaction depend on the transaction type. For details, see subsequent sections for each policy transaction type.

LOB-specific endpoints

Generally speaking, an LOB defines an insurance policy, which provides coverages (risk protection) for coverables (risk exposures, such as tangible assets). Additionally, an LOB could define other policy features such as modifiers, exclusions, or pre-qualification questions for underwriting.

After generating LOB-specific APIs using Advanced Product Designer (APD), the LOB-related endpoints are contained in the Job API. In processing a policy transaction, a caller will add, revise, or remove data related to the LOB.

The following sections provide general guidance on the data structures that can appear in LOB endpoints.

Products

Every job has at least one product. The available products can be accessed through the `/job/v1/jobs/{jobId}/lines` collection. Each product can be found at `/job/v1/jobs/{jobId}/lines/{productId}`. Here, `productId` is a placeholder for a specific product.

For example, on a job for a personal auto line policy, the personal auto line product is accessed through the `/job/v1/jobs/{jobId}/lines/PersonalAutoLine` endpoint.

Coverables

Coverables refer to any risk exposures that can be covered by the LOB or product. Coverables are accessible as collections on LOB endpoints. For example, one of the coverables for the `PersonalAutoLine` LOB is vehicle. Each `PersonalAutoLine` typically has a collection of one or more vehicles. Coverables can also be nested under other coverables. For example, another one of the coverables for `PersonalAutoLine` is driver. For each vehicle on a `PersonalAutoLine`, there can be a collection of drivers for that vehicle.

The basic pattern for accessing coverables is as follows:

- `/job/v1/jobs/{jobId}/lines/{productId}/{coverableType}/{coverableId}`
- `/job/v1/jobs/{jobId}/lines/{productId}/{coverableType}/{coverableId}/{coverableType}/{coverableId}`

Here, `coverableType` is a placeholder for a collection of some type of coverable, and `coverableId` refers to a specific coverable.

For example, in a personal auto line, vehicles and drivers are coverables. In the Jobs API, vehicles can be accessed through the `/job/v1/jobs/{jobId}/lines/PersonalAutoLine/vehicles` collection. Drivers associated with a covered vehicle are can be accessed through the `/job/v1/jobs/{jobId}/lines/PersonalAutoLine/vehicles/{vehicleId}/drivers` collection.

Also, the Job API provides a `locations` coverable type that is associated with the job directly, rather than the LOB. This collection can be used to define coverables in certain LOBs, such as for commercial property.

Coverages

A coverage is protection from a specific risk. Coverages are attached to coverables. There are two basic types of coverages: property and liability. A property coverage is a coverage for a tangible asset belonging to the insured, such as a vehicle. A liability coverage protects the insured, such as covering a driver for damage done to another vehicle.

The available coverages for a product can be accessed through the `/job/v1/jobs/{jobId}/lines/{productId}/coverages` collection. Each coverage can be found at `/job/v1/jobs/{jobId}/lines/{productId}/coverages/{coverageId}`. Depending on the LOB, coverages could be nested. For example, a location (a coverable) could have a collection of buildings (child coverables), and a coverage would be applied to each building. For property coverages, the basic pattern is `/job/v1/jobs/{jobId}/lines/{productId}/{coverableType}/{coverableId}/coverages`. For liability coverages, the policy line itself is a coverable, and represents the named insureds. The pattern is `/job/v1/jobs/{jobId}/lines/{productId}/coverages`.

For example, the personal auto line has both property (vehicle) and liability (driver) coverages. The vehicle coverages can be accessed through the `/job/v1/jobs/{jobId}/lines/PersonalAutoLine/vehicles/{vehicleId}/coverages` endpoint. The driver coverages cover liability, and thus can be found on the line itself: `/job/v1/jobs/{jobId}/line/PersonalAutoLine/coverages`.

Modifiers

Modifiers are factors that are applied as part of a rating algorithm. For example, the personal auto line could have an "Anti-Lock Breaks Discount" modifier for vehicles that provides a discount on the premium if the vehicle has anti-lock breaks. Modifiers can be associated with coverables or with the LOB itself.

- `/job/v1/jobs/{jobId}/lines/{productId}/modifiers`
- `/job/v1/jobs/{jobId}/lines/{productId}/{coverableType}/{coverableId}/modifiers`

For example, the personal auto line could have an "Anti-Lock Breaks Discount" modifier for vehicles that provides a discount on the premium if the vehicle has anti-lock breaks.

Exposures, exclusions, or conditions

An LOB might provide additional exposures, exclusions, or conditions. Exposures can refer to liability coverables, such as a driver in an auto policy. Exclusions define causes of loss that are explicitly not covered by the policy, so that the insurer has no exposure to claims in those areas. Conditions define contractual obligations of the insurance policy that are neither a coverage nor an exclusion.

Exposures, exclusions, or conditions can be accessed at the LOB level.

For example, a workers compensation line could make workplace exclusions accessible through a `/job/v1/jobs/{jobId}/lines/WorkersCompLine/excluded-workplaces` endpoint. Depending on the LOB, exclusions could be defined on a coverable, in which case the pattern would be `/job/v1/jobs/{jobId}/lines/{productId}/{coverableType}/{coverableId}/{excludedType}/{excludedId}`.

Pre-qualification questions for underwriting

In PolicyCenter, prequalifiction is accomplished through a series of questions answered during the submission process that can reveal possible underwriting issues. Answers to pre-qualification questions can be used to raise underwriting issues or block binding of the submission.

Pre-qualification could involve posing queries that are general to all products as well as specific to product, location, or coverable. As such, depending on the LOB, pre-qualification questions could be accessed through the following endpoints:

- `/job/v1/jobs/{jobId}/questions`
- `/job/v1/jobs/{jobId}/locations/{locationId}/questions`
- `/job/v1/jobs/{jobId}/lines/{productId}/questions`
- `/job/v1/jobs/{jobId}/lines/{productId}/locations/{locationId}/questions`
- `/job/v1/jobs/{jobId}/lines/{productId}/{coverableType}/{coverableId}/questions`

To illustrate, in a personal auto line, a GET request to `/job/v1/jobs/{jobId}/questions` could return the following:


```
{
  "data": {
    "attributes": {
      "answers": {
        "q1": {
          "question": {
            "displayName": "Have you been convicted for a moving traffic violation within the past 3 years?",
            "id": "q1"
          },
          "questionType": {
            "code": "Boolean",
            "name": "Boolean"
          }
        },
        "q2": {
          "question": {
            "displayName": "Has any policy or coverage been declined, canceled, or non-renewed during the prior 3
years?",
            "id": "q2"
          },
          "questionType": {
            "code": "Boolean",
            "name": "Boolean"
          }
        },
        "q3": {
          "question": {
            "displayName": "Has your license ever been canceled, suspended or revoked?",
            "id": "q3"
          },
          "questionType": {
            "code": "Choice",
            "name": "Choice"
          }
        },
        "q4": {
          "question": {
            "displayName": "Are you currently insured?",
            "id": "q4"
```

```
          },
          "questionType": {
            "code": "Boolean",
            "name": "Boolean"
          }
        }
      }
    },
    "checksum": "*",
    "links": "*"
  }
}
```

chapter 14

Submission

In PolicyCenter, all quotes and policies are originated through the submission policy transaction. During this transaction, an insurer collects and evaluates account holder data for the purpose of quoting and possibly creating a policy for a specific insurance product. The submission policy transaction is supported by the Job API.

For a general overview of how to conduct policy transactions using the system APIs, see "Policy transactions" on page 131. For details on the business functionality of submissions, see the *Application Guide*.

Basic process

1. Initiate the submission policy transaction by creating a submission job.
2. Modify the submission job.
   - Add coverables
   - Apply coverages and modifiers
   - Apply exposures, exclusions, or conditions
   - Pre-qualify the account holder
3. Generate a quote.
4. Complete the submission policy transaction in one of the following ways:
   - Bind the submission
   - Reject the submission
   - Withdraw the submission

Initiating the submission policy transaction

Initiating a submission policy transaction creates a new submission job that is in Draft state. All submissions, regardless of product or coverage, require the following data:

- **Account ID**: An existing account ID. If the account does not already exist, then you will have to create a new account before proceeding. For details, see "Creating an account" on page 123.
- **Base state code**: The jurisdiction of the primary location of the account, from the Jurisdiction typelist
- **Job effective date**: A date string in the form of `"YYYY-MM-DD"`
- **Producer code**: The code of the producer that is selling the policy
- **Product ID**: The type of policy that will be created through the submission. For example, `PersonalAuto` for a personal auto policy.

To initiate a submission, construct a request payload having the above data, and then submit it through a business action POST to `/job/v1/submissions`.

The following example contains a valid request payload for a personal auto line request:

```
{
  "data": {
    "attributes": {
      "account": {
        "id": "pc:401"
      },
      "baseState": {
        "code": "CA"
      },
      "jobEffectiveDate": "2020-08-01",
      "producerCode": {
        "id": "pc:6"
      },
      "product": {
        "id": "PersonalAuto"
      }
    }
  }
}
```

Initiating the submission creates a job element, which is returned in the response payload. All subsequent work in the transaction is executed on the job element. In the case of this example, that element is `/job/v1/jobs/pc:601`.

```
{
    "data": {
        "attributes": {
            "account": {
                "displayName": "0015863842",
                "id": "pc:401",
                "type": "Account",
                "uri": "/account/v1/accounts/pc:401"
            },
            . . .
            "id": "pc:601",
            . . .
            "jobStatus": {
                "code": "Draft",
                "name": "Draft"
            },
            "jobType": {
                "code": "Submission",
                "name": "Submission"
            },
            . . .
            "product": {
                "displayName": "Personal Auto",
                "id": "PersonalAuto"
            },
            . . .
        },
        . . .
        }
    }
}
```

Modifying the submission job

Modifying a submission job involves adding to the new draft submission all data that is necessary in order to generate a quote and create a policy. The specific types of data to add will depend on your product or line of business (LOB). Applying necessary data can entail one or more calls. The following sections provide general guidance on the types of data that might need to be added.

Add coverables

A coverable refers to any risk exposure that can be covered by a policy. Coverables are accessible as collections on LOB endpoints, and can also be nested.

For example, in a personal auto line, vehicles and drivers are coverables. In the Jobs API, vehicles can be added at `/job/v1/jobs/{jobId}/lines/PersonalAutoLine/vehicles`, with a request payload that resembles the following:

```
{
  "data": {
    "attributes": {
      "annualMileage": 10000,
```

```
      "bodyType": {
        "code": "convertible"
      },
      "color": "Yellow",
      "commutingMiles": 50,
      "costNew": {
        "amount": "25000.00",
        "currency": "usd"
      },
      "leaseOrRent": true,
      "lengthOfLease": {
        "code": "SixMonthsOrMore"
      },
      "licensePlate": "123456",
      "licenseState": {
        "code": "CA"
      },
      "make": "NewMake",
      "model": "NewModel",
      "modelYear": 2015,
      "primaryUse": {
        "code": "pleasure"
      },
      "statedValue": {
        "amount": "20000.00",
        "currency": "usd"
      },
      "vehicleType": {
        "code": "PP"
      },
      "vin": "1234567890"
    }
  }
}
```

Drivers associated with a covered vehicle can be added at `/job/v1/jobs/{jobId}/lines/PersonalAutoLine/vehicles/{vehicleId}/drivers`, with a request payload that resembles the following:

```
{
  "data": {
    "attributes": {
      "percentageDriven": 100,
      "policyDriver": {
        "id": "pc:401"
      }
    }
  }
}
```

The driver ID value must correspond to the ID of a contact element on the `/account/v1/accounts/{accountId}/contacts` collection.

Apply coverages and modifiers

A coverage is protection from a specific risk. Coverages are attached to coverables. There are two types of coverages: property and liability. To illustrate, on an automobile policy, a collision property coverage protects the insured's vehicle and a liability coverage protects the driver for damage done to another vehicle.

Typically, property coverages can be accessed at `/job/v1/jobs/{jobId}/lines/{lineId}/coverages`. Sometimes coverables are nested. For example, a location (a coverable) can have a collection of buildings (child coverables), and a coverage would be applied to each. The basic pattern is `/job/v1/jobs/{jobId}/lines/{lineId}/{coverableType}/{coverableId}/coverages`.

For liability coverages, the policy line itself is the coverable, and represents the named insureds. The pattern is `/job/v1/jobs/{jobId}/lines/{lineId}/coverages`.

A submission with multiple coverables and coverages can have implicit but unresolved dependencies. Also, the LOB product might support policy modifiers. Depending on the LOB, it might be necessary to synchronize the coverages and modifiers when composing the submission. For this purpose, the Job API provides `*/sync-coverages` and `*/sync-modifiers` endpoints on jobs that are in Draft state. These endpoints are accessible on the LOB as well as on each coverable, through the following paths:

- `/job/v1/jobs/{jobId}/lines/{productId}/sync-coverages`
- `/job/v1/jobs/{jobId}/lines/{productId}/{coverableType}/{coverableId}/sync-coverages`
- `/job/v1/jobs/{jobId}/lines/{productId}/sync-modifiers`

- `/job/v1/jobs/{jobId}/lines/{productId}/{coverableType}/{coverableId}/sync-modifiers`

To synchronize coverages and modifiers on a submission, a caller would submit a business action POST to these endpoints.

For example, in a personal auto line submission, a caller would synchronize vehicle coverages by submitting a business action POST to the `/job/v1/jobs/{jobId}/lines/PersonalAutoLine/vehicles/{vehicleId}/sync-coverages` endpoint. To apply coverages for the driver, a caller would submit a business action POST to the `/job/v1/jobs/{jobId}/lines/PersonalAutoLine/sync-coverages` endpoint. Likewise, to synchronize the policy modifiers with the submission, a caller would submit a business action POST to the `/job/v1/jobs/{jobId}/lines/PersonalAutoLine/sync-modifiers` and `/job/v1/jobs/{jobId}/lines/PersonalAutoLine/vehicles/{vehicleId}/sync-modifiers` endpoints.

Add exposures, exclusions, or conditions

Exposures, exclusions, and conditions are optional, and their applicability depends on the LOB.

In policies, exposures indicate exposure to risk, and are often identified by class code and jurisdiction. Exposures are associated with a coverable. For example, a farm insurance policy might support cow exposures, which could be accessed at the `/job/v1/jobs/{jobId}/lines/FarmInsuranceLine/farms/{farmId}/cows/{cowId}/cow-exposures` endpoint.

An exclusion is a specific cause of loss that will not be covered. For example, a workers compensation policy might allow certain workplace locations to be excluded from coverage, and those locations could be accessed at the `/job/v1/jobs/{jobId}/lines/WorkersCompLine/excluded-workplaces` endpoint.

Policy conditions are contractual obligations that neither provide nor exclude coverage. Policy conditions are diverse, and depend on the LOB.

Pre-qualify the account holder

Pre-qualifiction can be accomplished through a series of questions answered during the submission process that can reveal possible underwriting issues. Answers to pre-qualification questions can be used to raise underwriting issues or block binding of the submission. If answers are satisfactory, then the requested coverage can be provisionally approved by the producer. Pre-qualification questions are specific to an LOB product configuration. In the Jobs API, a `*/questions` collection can be associated with a job, an LOB product, or a coverable.

Pre-qualification could involve posing queries that are general to all products as well as specific to LOB, location, or coverable. As such, depending on the LOB, pre-qualification questions could be accessed through the following endpoints:

- `/job/v1/jobs/{jobId}/questions`
- `/job/v1/jobs/{jobId}/locations/{locationId}/questions`
- `/job/v1/jobs/{jobId}/lines/{productId}/questions`
- `/job/v1/jobs/{jobId}/lines/{productId}/locations/{locationId}/questions`
- `/job/v1/jobs/{jobId}/lines/{productId}/{coverableType}/{coverableId}/questions`

To illustrate, in a personal auto line, a GET request to `/job/v1/jobs/{jobId}/questions` could return the following:

```
{
  "data": {
    "attributes": {
      "answers": {
        "q1": {
          "question": {
            "displayName": "Have you been convicted for a moving traffic violation
                            within the past 3 years?",
            "id": "q1"
          },
          "questionType": {
            "code": "Boolean",
            "name": "Boolean"
          }
        },
        "q2": {
          "question": {
            "displayName": "Has any policy or coverage been declined, canceled,
                             or non-renewed during the prior 3 years?",
```

```
            "id": "q2"
          },
          "questionType": {
            "code": "Boolean",
            "name": "Boolean"
          }
        },
        "q3": {
          "question": {
            "displayName": "Has your license ever been canceled, suspended or revoked?",
            "id": "q3"
          },
          "questionType": {
            "code": "Choice",
            "name": "Choice"
          }
        },
        "q4": {
          "question": {
            "displayName": "Are you currently insured?",
            "id": "q4"
          },
          "questionType": {
            "code": "Boolean",
            "name": "Boolean"
          }
        }
      }
    },
    "checksum": "*",
    "links": "*"
  }
}
```


The Job API only provides endpoints that support pre-qualification. It is up to implementors to write the backing code.

Generating a quote

After applying all necessary modifications to the draft submission, a caller can generate a policy quote. The insurer and account holder will review the quote to determine how they wish to proceed with the prospective policy.

To generate a quote, a caller can submit a business action POST to the `/job/v1/jobs/{jobId}/quote` endpoint. This call will return a response payload containing the quote data, and will set the job to Quoted state.

If a quoted submission needs to be revised, then it must be reverted to draft status. This can be achieved by submitting a business action POST to the `/job/v1/jobs/{jobId}/make-draft` endpoint. A job in Draft state can be revised and re-quoted.

Also, a caller has the option to generate multiple quotes of the same job. This is useful for providing side-by-side comparison of policy options. For details, see “Multi-version quoting” on page 181.

Completing the submission

To complete the submission policy transaction, a caller must either bind, reject, or withdraw the quoted job.

Bind the submission

If the account holder and insurer agree to the terms and conditions associated with the quoted job, then the submission can be bound. Binding the submission transforms the quote into a policy, which is a legally binding contract between the two parties.

If at the time of binding the insurer requires no further information from the account holder, then the policy can also be issued. In issuing a policy, the insurer adds the new policy information to their systems and provides any necessary insurance documents to the account holder.

To bind and issue a quoted submission job, a caller can submit a business action POST to the `/job/v1/jobs/{jobId}/bind-and-issue` endpoint. Following the call, the `jobStatus` code will be set to `Bound`.

Alternatively, a submission can be bound only, with issuance postponed to a later time. This scenario can occur when an account holder must provide additional paperwork, or the insurer is not ready to deliver the policy documentation.

To only bind a quoted submission job but not issue it, a caller can submit a business action POST to the `/job/v1/jobs/{jobId}/bind-only` endpoint. Following the call, the `jobStatus` code will be set to `Bound`.

To later issue a bound-only policy, a caller can execute the issuance policy transaction on the policy. For details, see "Issuance" on page 161.

Reject the submission

A submission is rejected when the account holder does not accept the quote or the insurer declines to offer a policy to the account holder.

If the insurer declines to offer a policy, a caller can submit a business action POST to the `/job/v1/jobs/{jobId}/decline` endpoint. Following the call, the `jobStatus` code will be set to `Declined`.

If the account holder does not accept the policy quote, a caller would submit a business action POST to the `/job/v1/jobs/{jobId}/not-take` endpoint. Following the call, the `jobStatus` code will be set to `NotTaken`.

Withdraw the submission

If there is an error in the submission data, then the transaction can be withdrawn. To withdraw a submission policy transaction, a caller can submit a business action POST to the `/job/v1/jobs/{jobId}/withdraw` endpoint. Following the call, the `jobStatus` code will be set to `Withdrawn`.

The following table encapsulates the several ways that a quoted submission job can be completed:

Quote outcome	Condition	Target endpoint	Job status after call
Accepted	The account holder and insurer agree to the quote, and all necessary paperwork has been received by the insurer. The policy is bound and issued.	`/job/v1/jobs/{jobId}/bind-and-issue`	`Bound`
Accepted	The account holder and insurer agree to the quote, but the account holder must provide necessary paperwork. The policy is bound only. The policy can later be issued by executing the issuance policy transaction.	`/job/v1/jobs/{jobId}/bind-only`	`Bound`
Rejected	The insurer declines to offer a policy to the account holder. The policy request is declined.	`/job/v1/jobs/{jobId}/decline`	`Declined`
Rejected	The account holder rejects the quote. The policy is not taken.	`/job/v1/jobs/{jobId}/not-take`	`NotTaken`
Withdrawn	The quote contains erroneous data.	`/job/v1/jobs/{jobId}/withdrawn`	`Withdrawn`

Tutorial: Submission policy transaction

This tutorial presents a complete submission policy transaction for a personal auto line. It includes complete payload samples for each call.

The tutorial will demonstrate the following tasks:

- Create a new account for an individual
- Add an additional individual to the new account
- Initiate a submission policy transaction for a personal auto line
- Modify the job
  - Add a vehicle
  - Add a primary driver to the vehicle
  - Add a secondary driver to the vehicle
  - Synchronize coverages

    - Synchronize modifiers
- Quote the job
- Bind and issue the policy
- Review the policy

Create a new account for an individual

Every submission must be associated with an existing account. If the account does not already exist, then it must be created. With the Account API, a caller can create an account by submitting a POST request to the `/account/v1/accounts endpoint`. For details on creating new accounts, see “Creating an account” on page 123.

The following code block contains the essential data for creating a new personal account. This example takes advantage of request inclusion in applying `AccountContact` and `AccountLocation` resource data. For details on request inclusion, see “Request inclusion” on page 80.

```
{
  "data": {
    "attributes": {
      "accountHolder": {
        "refid": "newperson"
      },
      "organizationType": {
        "code": "individual"
      },
      "primaryLocation": {
        "refid": "newloc"
      },
      "producerCodes": [
        {
          "id": "pc:6"
        }
      ]
    }
  },
  "included": {
    "AccountContact": [
      {
        "attributes": {
          "contactSubtype": "Person",
          "firstName": "Bill",
          "lastName": "Preston",
          "dateOfBirth": "1970-01-01",
          "emailAddress1": "bpreston@example.com",
          "licenseNumber": "D123456789",
          "licenseState": {
            "code": "CA"
          },
          "numberOfAccidents": {
            "code": "0"
          },
          "numberOfViolations": {
            "code": "0"
          },
          "primaryAddress": {
            "addressLine1": "2850 S. Delaware St. #400",
            "city": "San Mateo",
            "postalCode": "94403",
            "state": {
              "code": "CA"
            }
          }
        },
        "method": "post",
        "refid": "newperson",
        "uri": "/account/v1/accounts/this/contacts"
      }
    ],
    "AccountLocation": [
      {
        "attributes": {
          "nonSpecific": true,
          "postalCode": "94403",
          "state": {
            "code": "CA"
          }
        },
        "method": "post",
        "refid": "newloc",
        "uri": "/account/v1/accounts/this/locations"
      }
    ]
```

```
  }
}
```

Submitting the previous payload in a POST request to `/account/v1/accounts` will return a response payload similar to the following:


```
{
    "data": {
        "attributes": {
            "accountHolder": {
                "displayName": "Bill Preston",
                "id": "pc:697"
            },
            "accountNumber": "2768207338",
            "accountStatus": {
                "code": "Pending",
                "name": "Pending"
            },
            "createdDate": "2020-07-23T00:21:48.323Z",
            "frozen": false,
            "id": "pc:8",
            "numberOfContacts": "1",
            "organizationType": {
                "code": "individual",
                "name": "Individual"
            },
            "preferredCoverageCurrency": {
                "code": "usd",
                "name": "USD"
            },
            "preferredSettlementCurrency": {
                "code": "usd",
                "name": "USD"
            },
            "primaryLanguage": {
                "code": "en_US",
                "name": "English (US)"
            },
            "primaryLocale": {
                "code": "en_US",
                "name": "United States (English)"
            },
            "primaryLocation": {
                "displayName": "1: CA",
                "id": "pc:799",
                "type": "AccountLocation",
                "uri": "/account/v1/accounts/pc:8/locations/pc:799"
            },
            "producerCodes": [
                {
                    "displayName": "100-002541",
                    "id": "pc:6"
                }
            ]
        },
        "checksum": "0",
        "links": {
            "activities": {
                "href": "/account/v1/accounts/pc:8/activities",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "activity-assignees": {
                "href": "/account/v1/accounts/pc:8/activity-assignees",
                "methods": [
                    "get"
                ]
            },
            "activity-patterns": {
                "href": "/account/v1/accounts/pc:8/activity-patterns",
                "methods": [
                    "get"
                ]
            },
            "contacts": {
                "href": "/account/v1/accounts/pc:8/contacts",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "documents": {
                "href": "/account/v1/accounts/pc:8/documents",
                "methods": [
                    "get",
                    "post"
```

```
                ]
            },
            "jobs": {
                "href": "/account/v1/accounts/pc:8/jobs",
                "methods": [
                    "get"
                ]
            },
            "locations": {
                "href": "/account/v1/accounts/pc:8/locations",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "notes": {
                "href": "/account/v1/accounts/pc:8/notes",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "policies": {
                "href": "/account/v1/accounts/pc:8/policies",
                "methods": [
                    "get"
                ]
            },
            "self": {
                "href": "/account/v1/accounts/pc:8",
                "methods": [
                    "get",
                    "patch"
                ]
            }
        }
    }
}
```

The new account is in Pending status. The ID for this account is `pc:8`. This value will be used in the later sections.

Add an additional individual to the new account

In this exercise, the new account will include a second individual, who will be the secondary driver. This contact can be added to the account by submitting the following payload in a POST request to `/account/v1/accounts/pc:8/contacts`:

```
{
  "data": {
    "attributes": {
      "contactSubtype": "Person",
      "dateOfBirth": "1971-01-01",
      "emailAddress1": "jill@example.com",
      "firstName": "Jill",
      "lastName": "Preston",
      "licenseNumber": "D234567890",
      "licenseState": {
        "code": "CA"
      },
      "numberOfAccidents": {
        "code": "0"
      },
      "numberOfViolations": {
        "code": "0"
      },
      "primaryAddress": {
          "addressLine1": "2850 S. Delaware St. #400",
          "city": "San Mateo",
          "postalCode": "94403",
          "state": {
              "code": "CA"
          }
      }
    }
  }
}
```

The response payload contains data for the new contact element, which is accessible at `/account/v1/accounts/pc:8/contacts/pc:698`:

```
{
    "data": {
        "attributes": {
            "contactSubtype": "Person",
```

```
            "dateOfBirth": "1971-01-01",
            "displayName": "Jill Preston",
            "emailAddress1": "jill@example.com",
            "externalId": "pcext:698",
            "firstName": "Jill",
            "id": "pc:698",
            "lastName": "Preston",
            "licenseNumber": "D123456789",
            "licenseState": {
                "code": "CA",
                "name": "California"
            },
            "numberOfAccidents": {
                "code": "0",
                "name": "0"
            },
            "numberOfViolations": {
                "code": "0",
                "name": "0"
            },
            "primaryAddress": {
                "addressLine1": "2850 S. Delaware St. #400",
                "city": "San Mateo",
                "country": "US",
                "displayName": "2850 S. Delaware St. #400, San Mateo, CA 94403",
                "id": "pc:800",
                "postalCode": "94403",
                "state": {
                    "code": "CA",
                    "name": "California"
                }
            }
        },
        "checksum": "3df12e427af4fbcc5e92ac25418b18eb",
        "links": {
            "self": {
                "href": "/account/v1/accounts/pc:8/contacts/pc:698",
                "methods": [
                    "get",
                    "patch"
                ]
            }
        }
    }
}
```


Initiate the submission policy transaction

A submission policy transaction can be initiated through a business action POST to the `/job/v1/submissions` endpoint. The request must include a payload:

```
{
  "data": {
    "attributes": {
      "account": {
        "id": "pc:8"
      },
      "baseState": {
        "code": "CA"
      },
      "jobEffectiveDate": "2020-08-01",
      "producerCode": {
        "id": "pc:6"
      },
      "product": {
        "id": "PersonalAuto"
      }
    }
  }
}
```

The payload contains the following properties:

- `account.id`: The account ID, as created in the previous example
- `baseState.code`: The jurisdiction of the account. This value is derived from the Jurisdiction typelist
- `jobEffectiveDate`: The date on which the transaction should take effect
- `producerCode.id`: The producer code ID
- `product.id`: The product name

The call will create a new submission job in Draft status, the data for which will be returned in the response payload:


```
{
    "data": {
        "attributes": {
            "account": {
                "displayName": "2768207338",
                "id": "pc:8",
                "type": "Account",
                "uri": "/account/v1/accounts/pc:8"
            },
            "baseState": {
                "code": "CA",
                "name": "California"
            },
            "createdDate": "2020-07-23T00:25:46.038Z",
            "id": "pc:16",
            "jobEffectiveDate": "2020-08-01T00:01:00.000Z",
            "jobNumber": "0001584961",
            "jobStatus": {
                "code": "Draft",
                "name": "Draft"
            },
            "jobType": {
                "code": "Submission",
                "name": "Submission"
            },
            "organization": {
                "displayName": "Armstrong and Company",
                "id": "pc:1"
            },
            "periodEnd": "2021-02-01T00:01:00.000Z",
            "periodStart": "2020-08-01T00:01:00.000Z",
            "primaryInsured": {
                "displayName": "Bill Preston",
                "id": "pc:697",
                "type": "PolicyContact",
                "uri": "/job/v1/jobs/pc:16/contacts/pc:697"
            },
            "producerCode": {
                "displayName": "100-002541",
                "id": "pc:6"
            },
            "product": {
                "displayName": "Personal Auto",
                "id": "PersonalAuto"
            },
            "quoteType": {
                "code": "Full",
                "name": "Full Application"
            },
            "termType": {
                "code": "HalfYear",
                "name": "6 months"
            }
        },
        "checksum": "1b6514990bb9746dfaa78fbddef25733",
        "links": {
            "activities": {
                "href": "/job/v1/jobs/pc:16/activities",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "activity-assignees": {
                "href": "/job/v1/jobs/pc:16/activity-assignees",
                "methods": [
                    "get"
                ]
            },
            "activity-patterns": {
                "href": "/job/v1/jobs/pc:16/activity-patterns",
                "methods": [
                    "get"
                ]
            },
            "change-version": {
                "href": "/job/v1/jobs/pc:16/change-version",
                "methods": [
                    "post"
                ]
            },
            "contacts": {
                "href": "/job/v1/jobs/pc:16/contacts",
                "methods": [
                    "get",
                    "post"
                ]
```

```
            },
            "costs": {
                "href": "/job/v1/jobs/pc:16/costs",
                "methods": [
                    "get"
                ]
            },
            "decline": {
                "href": "/job/v1/jobs/pc:16/decline",
                "methods": [
                    "post"
                ]
            },
            "documents": {
                "href": "/job/v1/jobs/pc:16/documents",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "lines": {
                "href": "/job/v1/jobs/pc:16/lines",
                "methods": [
                    "get"
                ]
            },
            "locations": {
                "href": "/job/v1/jobs/pc:16/locations",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "not-take": {
                "href": "/job/v1/jobs/pc:16/not-take",
                "methods": [
                    "post"
                ]
            },
            "notes": {
                "href": "/job/v1/jobs/pc:16/notes",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "payment-plans": {
                "href": "/job/v1/jobs/pc:16/payment-plans",
                "methods": [
                    "get"
                ]
            },
            "questions": {
                "href": "/job/v1/jobs/pc:16/questions",
                "methods": [
                    "get",
                    "patch"
                ]
            },
            "quote": {
                "href": "/job/v1/jobs/pc:16/quote",
                "methods": [
                    "post"
                ]
            },
            "self": {
                "href": "/job/v1/jobs/pc:16",
                "methods": [
                    "get",
                    "patch"
                ]
            },
            "user-roles": {
                "href": "/job/v1/jobs/pc:16/user-roles",
                "methods": [
                    "get",
                    "patch"
                ]
            },
            "versions": {
                "href": "/job/v1/jobs/pc:16/versions",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "withdraw": {
                "href": "/job/v1/jobs/pc:16/withdraw",
                "methods": [
                    "post"
```

```
                ]
            }
        }
    }
}
```

The `id` property contains the ID of the new job element that was created by the request. Subsequent actions in the transaction will occur on the job element, in this case `/job/v1/jobs/pc:16`.

Modify the job: Add a vehicle

On a personal auto line, a vehicle is a coverable. For any submission job, each coverable must be added through a POST request. To add a vehicle to this job, submit a POST request to `/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles` with the following payload:

```
{
  "data": {
    "attributes": {
      "annualMileage": 10000,
      "bodyType": {
        "code": "convertible"
      },
      "color": "Yellow",
      "commutingMiles": 50,
      "costNew": {
        "amount": "25000.00",
        "currency": "usd"
      },
      "leaseOrRent": true,
      "lengthOfLease": {
        "code": "SixMonthsOrMore"
      },
      "licensePlate": "7JDX543",
      "licenseState": {
        "code": "CA"
      },
      "make": "NewMake",
      "model": "NewModel",
      "modelYear": 2015,
      "primaryUse": {
        "code": "pleasure"
      },
      "statedValue": {
        "amount": "20000.00",
        "currency": "usd"
      },
      "vehicleType": {
        "code": "PP"
      },
      "vin": "WDDHF8JB3CA549096"
    }
  }
}
```

This call returns the following response payload. The vehicle is now accessible at the `/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6` endpoint:

```
{
    "data": {
        "attributes": {
            "annualMileage": 10000,
            "bodyType": {
                "code": "convertible",
                "name": "Convertible"
            },
            "color": "Yellow",
            "commutingMiles": 50,
            "costNew": {
                "amount": "25000.00",
                "currency": "usd"
            },
            "garageLocation": {
                "displayName": "1: CA",
                "id": "46"
            },
            "id": "6",
            "leaseOrRent": true,
            "lengthOfLease": {
                "code": "SixMonthsOrMore",
                "name": "6 months or greater"
            },
            "licensePlate": "7JDX543",
            "licenseState": {
```

```
                "code": "CA",
                "name": "California"
            },
            "make": "NewMake",
            "model": "NewModel",
            "modelYear": 2015,
            "primaryUse": {
                "code": "pleasure",
                "name": "Pleasure"
            },
            "statedValue": {
                "amount": "20000.00",
                "currency": "usd"
            },
            "vehicleNumber": 1,
            "vehicleType": {
                "code": "PP",
                "name": "Passenger Vehicles"
            },
            "vin": "WDDHF8JB3CA549096"
        },
        "checksum": "d69df836a5af3ef1cd06d0025694c7bc",
        "links": {
            "coverages": {
                "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/coverages",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "drivers": {
                "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/drivers",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "modifiers": {
                "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/modifiers",
                "methods": [
                    "get"
                ]
            },
            "self": {
                "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6",
                "methods": [
                    "delete",
                    "get",
                    "patch"
                ]
            },
            "sync-coverages": {
                "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/sync-coverages",
                "methods": [
                    "post"
                ]
            },
            "sync-modifiers": {
                "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/sync-modifiers",
                "methods": [
                    "post"
                ]
            },
            "vhcle-addl-interests": {
                "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/vhcle-addl-interests",
                "methods": [
                    "get",
                    "post"
                ]
            }
        }
    }
}
```


Modify the job: Add a primary driver to the vehicle

In this example, Bill Preston will be the primary driver, and he will drive the vehicle 75% of the time. The account contact ID for Bill is `pc:697`. To add Bill as a driver of this vehicle, submit a POST request to `/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/drivers`, with the following payload:


```
{
  "data": {
    "attributes": {
      "percentageDriven": 75,
      "policyDriver": {
        "id": "pc:697"
```

```
      }
    }
  }
}
```

This call returns the following response:

```
{
    "data": {
        "attributes": {
            "id": "5",
            "percentageDriven": 75,
            "policyDriver": {
                "displayName": "Bill Preston",
                "id": "pc:697",
                "type": "PolicyContact",
                "uri": "/job/v1/jobs/pc:16/contacts/pc:697"
            }
        },
        "checksum": "109d5f8706f79c9481f6a61738e2b554",
        "links": {
            "self": {
                "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/drivers/5",
                "methods": [
                    "delete",
                    "get",
                    "patch"
                ]
            }
        }
    }
}
```

Modify the job: Add a secondary driver to the vehicle

In this example, Jill Preston will be the secondary driver, and she will drive the vehicle 25% of the time. The account contact ID for Jill is

```
pc:698
```

. To add Jill as a driver of this vehicle, submit a POST request to `/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/drivers`, with the following payload:

```
{
  "data": {
    "attributes": {
      "percentageDriven": 25,
      "policyDriver": {
        "id": "pc:698"
      }
    }
  }
}
```

This call returns the following response:

```
{
    "data": {
        "attributes": {
            "id": "7",
            "percentageDriven": 25,
            "policyDriver": {
                "displayName": "Jill Preston",
                "id": "pc:698",
                "type": "PolicyContact",
                "uri": "/job/v1/jobs/pc:16/contacts/pc:698"
            }
        },
        "checksum": "e10a37e9890009522fa8fa0c5574c2a0",
        "links": {
            "self": {
                "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/drivers/7",
                "methods": [
                    "delete",
                    "get",
                    "patch"
                ]
            }
        }
    }
}
```

Modify the job: Synchronize coverages

To synchronize coverages, submit a business action POST to `/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/sync-coverages`.

To review coverages, submit a GET request to `/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/coverages`. This call returns the following payload:


```
{
    "count": 2,
    "data": [
        {
            "attributes": {
                "clauseType": "coverage",
                "id": "PACollisionCov",
                "pattern": {
                    "displayName": "Collision",
                    "id": "PACollisionCov"
                },
                "selected": true,
                "terms": {
                    "PACollDeductible": {
                        "choiceValue": {
                            "code": "500",
                            "name": "500"
                        },
                        "covTermType": "choice",
                        "displayValue": "500",
                        "pattern": {
                            "displayName": "Collision Deductible",
                            "id": "PACollDeductible"
                        }
                    }
                }
            },
            "checksum": "006b1f7b0a4a803105eeb6a6e86ee270",
            "links": {
                "self": {
                    "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/coverages/PACollisionCov",
                    "methods": [
                        "get"
                    ]
                }
            }
        },
        {
            "attributes": {
                "clauseType": "coverage",
                "id": "PAComprehensiveCov",
                "pattern": {
                    "displayName": "Comprehensive",
                    "id": "PAComprehensiveCov"
                },
                "selected": true,
                "terms": {
                    "PACompDeductible": {
                        "choiceValue": {
                            "code": "500",
                            "name": "500"
                        },
                        "covTermType": "choice",
                        "displayValue": "500",
                        "pattern": {
                            "displayName": "Comprehensive Deductible",
                            "id": "PACompDeductible"
                        }
                    }
                }
            },
            "checksum": "77fae36ca9d0dcdc16a1edec63e94877",
            "links": {
                "self": {
                    "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/coverages/PAComprehensiveCov",
                    "methods": [
                        "get"
                    ]
                }
            }
        }
    ],
    "links": {
        "first": {
            "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/coverages",
            "methods": [
                "get"
            ]
        },
```

```
        "self": {
            "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/coverages",
            "methods": [
                "get"
            ]
        }
    }
}
```

Modify the job: Synchronize modifiers

To synchronize modifiers, submit a business action POST to `/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/sync-modifiers`.

To review modifiers, submit a GET request to `/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/modifiers`. This call returns the following payload:

```
{
    "count": 3,
    "data": [
        {
            "attributes": {
                "booleanModifier": false,
                "eligible": true,
                "id": "PAAntiLockBrakes",
                "modifierType": {
                    "code": "boolean",
                    "name": "boolean"
                },
                "pattern": {
                    "displayName": "Anti-Lock Brakes Discount",
                    "id": "PAAntiLockBrakes"
                },
                "referenceDate": "2020-08-01T00:00:00.000Z",
                "state": {
                    "code": "CA",
                    "name": "California"
                },
                "valueFinal": true
            },
            "checksum": "332014107be09a1a45b46398c2a730ea",
            "links": {
                "self": {
                    "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/modifiers/PAAntiLockBrakes",
                    "methods": [
                        "get"
                    ]
                }
            }
        },
        {
            "attributes": {
                "eligible": true,
                "id": "PAAntiTheft",
                "modifierType": {
                    "code": "typekey",
                    "name": "typekey"
                },
                "pattern": {
                    "displayName": "Anti Theft Discount",
                    "id": "PAAntiTheft"
                },
                "referenceDate": "2020-08-01T00:00:00.000Z",
                "state": {
                    "code": "CA",
                    "name": "California"
                },
                "valueFinal": true
            },
            "checksum": "63a2e842883e8993ce10073e4f5a33e7",
            "links": {
                "self": {
                    "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/modifiers/PAAntiTheft",
                    "methods": [
                        "get"
                    ]
                }
            }
        },
        {
            "attributes": {
                "eligible": true,
                "id": "PAPassiveRestraint",
                "modifierType": {
                    "code": "typekey",
                    "name": "typekey"
```

```
                },
                "pattern": {
                    "displayName": "Passive Restraint System",
                    "id": "PAPassiveRestraint"
                },
                "referenceDate": "2020-08-01T00:00:00.000Z",
                "state": {
                    "code": "CA",
                    "name": "California"
                },
                "valueFinal": true
            },
            "checksum": "cc898e3294ba4d68e81518d96e37c985",
            "links": {
                "self": {
                    "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/modifiers/PAPassiveRestraint",
                    "methods": [
                        "get"
                    ]
                }
            }
        }
    ],
    "links": {
        "first": {
            "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/modifiers",
            "methods": [
                "get"
            ]
        },
        "self": {
            "href": "/job/v1/jobs/pc:16/lines/PersonalAutoLine/vehicles/6/modifiers",
            "methods": [
                "get"
            ]
        }
    }
}
```


Quote the job

To quote the job, submit a business action POST to `/job/v1/jobs/pc:16/quote`. This call returns the following payload:


```
{
    "data": {
        "attributes": {
            "account": {
                "displayName": "2768207338",
                "id": "pc:8",
                "type": "Account",
                "uri": "/account/v1/accounts/pc:8"
            },
            "baseState": {
                "code": "CA",
                "name": "California"
            },
            "createdDate": "2020-7-23T00:25:46.038Z",
            "id": "pc:16",
            "jobEffectiveDate": "2020-08-01T00:01:00.000Z",
            "jobNumber": "0001584961",
            "jobStatus": {
                "code": "Quoted",
                "name": "Quoted"
            },
            "jobType": {
                "code": "Submission",
                "name": "Submission"
            },
            "organization": {
                "displayName": "Armstrong and Company",
                "id": "pc:1"
            },
            "periodEnd": "2021-02-01T00:01:00.000Z",
            "periodStart": "2020-08-01T00:01:00.000Z",
            "primaryInsured": {
                "displayName": "Bill Preston",
                "id": "pc:697",
                "type": "PolicyContact",
                "uri": "/job/v1/jobs/pc:16/contacts/pc:697"
            },
            "producerCode": {
                "displayName": "100-002541",
                "id": "pc:6"
            },
            "product": {
                "displayName": "Personal Auto",
```

```
            "id": "PersonalAuto"
        },
        "quoteType": {
            "code": "Full",
            "name": "Full Application"
        },
        "taxesAndSurcharges": {
            "amount": "68.00",
            "currency": "usd"
        },
        "termType": {
            "code": "HalfYear",
            "name": "6 months"
        },
        "totalCost": {
            "amount": "1012.00",
            "currency": "usd"
        },
        "totalPremium": {
            "amount": "944.00",
            "currency": "usd"
        }
    },
    "checksum": "a90905400e592b9a525ca94b48ae7a1b",
    "links": {
        "activities": {
            "href": "/job/v1/jobs/pc:16/activities",
            "methods": [
                "get",
                "post"
            ]
        },
        "activity-assignees": {
            "href": "/job/v1/jobs/pc:16/activity-assignees",
            "methods": [
                "get"
            ]
        },
        "activity-patterns": {
            "href": "/job/v1/jobs/pc:16/activity-patterns",
            "methods": [
                "get"
            ]
        },
        "bind-and-issue": {
            "href": "/job/v1/jobs/pc:16/bind-and-issue",
            "methods": [
                "post"
            ]
        },
        "bind-only": {
            "href": "/job/v1/jobs/pc:16/bind-only",
            "methods": [
                "post"
            ]
        },
        "change-version": {
            "href": "/job/v1/jobs/pc:16/change-version",
            "methods": [
                "post"
            ]
        },
        "contacts": {
            "href": "/job/v1/jobs/pc:16/contacts",
            "methods": [
                "get"
            ]
        },
        "costs": {
            "href": "/job/v1/jobs/pc:16/costs",
            "methods": [
                "get"
            ]
        },
        "decline": {
            "href": "/job/v1/jobs/pc:16/decline",
            "methods": [
                "post"
            ]
        },
        "documents": {
            "href": "/job/v1/jobs/pc:16/documents",
            "methods": [
                "get",
                "post"
            ]
        },
        "lines": {
            "href": "/job/v1/jobs/pc:16/lines",
            "methods": [
```

```
                    "get"
                ]
            },
            "locations": {
                "href": "/job/v1/jobs/pc:16/locations",
                "methods": [
                    "get"
                ]
            },
            "make-draft": {
                "href": "/job/v1/jobs/pc:16/make-draft",
                "methods": [
                    "post"
                ]
            },
            "not-take": {
                "href": "/job/v1/jobs/pc:16/not-take",
                "methods": [
                    "post"
                ]
            },
            "notes": {
                "href": "/job/v1/jobs/pc:16/notes",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "payment-info": {
                "href": "/job/v1/jobs/pc:16/payment-info",
                "methods": [
                    "get",
                    "patch"
                ]
            },
            "payment-plans": {
                "href": "/job/v1/jobs/pc:16/payment-plans",
                "methods": [
                    "get"
                ]
            },
            "questions": {
                "href": "/job/v1/jobs/pc:16/questions",
                "methods": [
                    "get"
                ]
            },
            "self": {
                "href": "/job/v1/jobs/pc:16",
                "methods": [
                    "get"
                ]
            },
            "user-roles": {
                "href": "/job/v1/jobs/pc:16/user-roles",
                "methods": [
                    "get"
                ]
            },
            "versions": {
                "href": "/job/v1/jobs/pc:16/versions",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "withdraw": {
                "href": "/job/v1/jobs/pc:16/withdraw",
                "methods": [
                    "post"
                ]
            }
        }
    }
}
```


Bind and issue the policy

To bind and issue the policy, submit a business action POST to `/job/v1/jobs/pc:16/bind-and-issue`. This call returns the bound job. The `policy` and `policyNumber` properties contain values that can be used to look up the new policy. The response payload appears as follows:


```
{
    "data": {
        "attributes": {
            "account": {
```

```
            "displayName": "2768207338",
            "id": "pc:8",
            "type": "Account",
            "uri": "/account/v1/accounts/pc:8"
        },
        "baseState": {
            "code": "CA",
            "name": "California"
        },
        "closeDate": "2020-07-23T01:02:02.119Z",
        "createdDate": "2020-07-23T00:25:46.038Z",
        "id": "pc:16",
        "jobEffectiveDate": "2020-08-01T00:01:00.000Z",
        "jobNumber": "0001584961",
        "jobStatus": {
            "code": "Bound",
            "name": "Bound"
        },
        "jobType": {
            "code": "Submission",
            "name": "Submission"
        },
        "organization": {
            "displayName": "Armstrong and Company",
            "id": "pc:1"
        },
        "periodEnd": "2021-02-01T00:01:00.000Z",
        "periodStart": "2020-08-01T00:01:00.000Z",
        "policy": {
            "displayName": "6036550150",
            "id": "pc:7"
        },
        "policyNumber": "6036550150",
        "primaryInsured": {
            "displayName": "Bill Preston",
            "id": "pc:697",
            "type": "PolicyContact",
            "uri": "/job/v1/jobs/pc:16/contacts/pc:697"
        },
        "producerCode": {
            "displayName": "100-002541",
            "id": "pc:6"
        },
        "product": {
            "displayName": "Personal Auto",
            "id": "PersonalAuto"
        },
        "quoteType": {
            "code": "Full",
            "name": "Full Application"
        },
        "taxesAndSurcharges": {
            "amount": "68.00",
            "currency": "usd"
        },
        "termType": {
            "code": "HalfYear",
            "name": "6 months"
        },
        "totalCost": {
            "amount": "1012.00",
            "currency": "usd"
        },
        "totalPremium": {
            "amount": "944.00",
            "currency": "usd"
        }
    },
    "checksum": "426c4ebf6898be74546b06612523d224",
    "links": {
        "activities": {
            "href": "/job/v1/jobs/pc:16/activities",
            "methods": [
                "get",
                "post"
            ]
        },
        "activity-assignees": {
            "href": "/job/v1/jobs/pc:16/activity-assignees",
            "methods": [
                "get"
            ]
        },
        "activity-patterns": {
            "href": "/job/v1/jobs/pc:16/activity-patterns",
            "methods": [
                "get"
            ]
        },
        "contacts": {
```

```
                "href": "/job/v1/jobs/pc:16/contacts",
                "methods": [
                    "get"
                ]
            },
            "costs": {
                "href": "/job/v1/jobs/pc:16/costs",
                "methods": [
                    "get"
                ]
            },
            "documents": {
                "href": "/job/v1/jobs/pc:16/documents",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "lines": {
                "href": "/job/v1/jobs/pc:16/lines",
                "methods": [
                    "get"
                ]
            },
            "locations": {
                "href": "/job/v1/jobs/pc:16/locations",
                "methods": [
                    "get"
                ]
            },
            "notes": {
                "href": "/job/v1/jobs/pc:16/notes",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "payment-info": {
                "href": "/job/v1/jobs/pc:16/payment-info",
                "methods": [
                    "get"
                ]
            },
            "payment-plans": {
                "href": "/job/v1/jobs/pc:16/payment-plans",
                "methods": [
                    "get"
                ]
            },
            "questions": {
                "href": "/job/v1/jobs/pc:16/questions",
                "methods": [
                    "get"
                ]
            },
            "self": {
                "href": "/job/v1/jobs/pc:16",
                "methods": [
                    "get"
                ]
            },
            "user-roles": {
                "href": "/job/v1/jobs/pc:16/user-roles",
                "methods": [
                    "get"
                ]
            },
            "versions": {
                "href": "/job/v1/jobs/pc:16/versions",
                "methods": [
                    "get"
                ]
            }
        }
    }
}
```


Review the policy

The new policy is accessible at `/policy/v1/policies/pc:7`. Submitting a GET request to that endpoint returns the following:


```
{
    "data": {
        "attributes": {
            "account": {
                "displayName": "2768207338",
```

```
            "id": "pc:8",
            "type": "Account",
            "uri": "/account/v1/accounts/pc:8"
        },
        "baseState": {
            "code": "CA",
            "name": "California"
        },
        "createdDate": "2020-07-23T00:25:46.038Z",
        "id": "pc:7",
        "organization": {
            "displayName": "Armstrong and Company",
            "id": "pc:1"
        },
        "periodEnd": "2021-02-01T00:01:00.000Z",
        "periodStart": "2020-08-01T00:01:00.000Z",
        "policyNumber": "6036550150",
        "primaryInsured": {
            "displayName": "Bill Preston",
            "id": "pc:697",
            "type": "PolicyContact",
            "uri": "/policy/v1/policies/pc:7/contacts/pc:697"
        },
        "producerCode": {
            "displayName": "100-002541",
            "id": "pc:6"
        },
        "product": {
            "displayName": "Personal Auto",
            "id": "PersonalAuto"
        },
        "taxesAndSurcharges": {
            "amount": "68.00",
            "currency": "usd"
        },
        "termType": {
            "code": "HalfYear",
            "name": "6 months"
        },
        "totalCost": {
            "amount": "1012.00",
            "currency": "usd"
        },
        "totalPremium": {
            "amount": "944.00",
            "currency": "usd"
        }
    },
    "checksum": "1",
    "links": {
        "activities": {
            "href": "/policy/v1/policies/pc:7/activities",
            "methods": [
                "get",
                "post"
            ]
        },
        "activity-assignees": {
            "href": "/policy/v1/policies/pc:7/activity-assignees",
            "methods": [
                "get"
            ]
        },
        "activity-patterns": {
            "href": "/policy/v1/policies/pc:7/activity-patterns",
            "methods": [
                "get"
            ]
        },
        "cancel": {
            "href": "/policy/v1/policies/pc:7/cancel",
            "methods": [
                "post"
            ]
        },
        "change": {
            "href": "/policy/v1/policies/pc:7/change",
            "methods": [
                "post"
            ]
        },
        "contacts": {
            "href": "/policy/v1/policies/pc:7/contacts",
            "methods": [
                "get"
            ]
        },
        "costs": {
            "href": "/policy/v1/policies/pc:7/costs",
            "methods": [
```

```
                    "get"
                ]
            },
            "documents": {
                "href": "/policy/v1/policies/pc:7/documents",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "lines": {
                "href": "/policy/v1/policies/pc:7/lines",
                "methods": [
                    "get"
                ]
            },
            "locations": {
                "href": "/policy/v1/policies/pc:7/locations",
                "methods": [
                    "get"
                ]
            },
            "notes": {
                "href": "/policy/v1/policies/pc:7/notes",
                "methods": [
                    "get",
                    "post"
                ]
            },
            "payment-info": {
                "href": "/policy/v1/policies/pc:7/payment-info",
                "methods": [
                    "get"
                ]
            },
            "questions": {
                "href": "/policy/v1/policies/pc:7/questions",
                "methods": [
                    "get"
                ]
            },
            "renew": {
                "href": "/policy/v1/policies/pc:7/renew",
                "methods": [
                    "post"
                ]
            },
            "rewrite-account": {
                "href": "/policy/v1/policies/pc:7/rewrite-account",
                "methods": [
                    "post"
                ]
            },
            "self": {
                "href": "/policy/v1/policies/pc:7",
                "methods": [
                    "get"
                ]
            }
        }
    }
}
```

chapter 15

Issuance

Binding a policy refers to making the policy a legally binding contract between the insurer and the insurer. Issuing a policy refers to printing the paperwork related to the policy. Policies are often bound and issued at the same time. But in some situations, an insurer might want to bind a policy without issuing it. This can happen when the insurer wants to put the policy in force now, but the insurer is still gathering supplemental information and therefore does not yet want to print the policy documents.

A submission can be bound but not issued through the system APIs using the `/bind-only` endpoint. When the insurer is ready to issue the policy, they can do so using the `/issue` endpoint. For more information on the business functionality of issuing a bound policy, see *Application Guide*.

Issue the policy

1. Initiate the issuance policy transaction.

   Submit a business action POST to the `/policy/v1/policies/{policyId}/issue` endpoint, containing an empty request payload.

   ```
   {
       "data": {
           "attributes": {}
       }
   }
   ```

   The response payload contains the associated job. Use the job ID in subsequent calls.
2. Modify the job *(optional)*.

   The job is in Draft state, and can be modified. For example, the policy could require a new coverable. Details of modifying a job depend on the LOB and product.
3. Generate a policy quote.

   Submit a business action POST to the `/job/v1/jobs/{jobId}/quote` endpoint.
4. Complete the issuance policy transaction.

   To complete the issuance policy transaction, the quoted job can be bound and issued. If the transaction is to be discarded, such as for erroneous data, then it can be withdrawn.

   The following table illustrates the two ways that a quoted issuance job can be completed:

Transaction outcome	Condition	Job status	Endpoint
Accepted	The account holder and insurer agree to the quote, and all necessary paperwork has been received by the insurer. The policy is bound and issued.	Bound	`/job/v1/jobs/{jobId}/bind-and-issue`
Withdrawn	The quote contains erroneous data.	Withdrawn	`/job/v1/jobs/{jobId}/withdrawn`

chapter 16

Renewal

The renewal process extends the policy for another term beyond the current expiration date. It creates a new policy period for an existing policy. The renewal policy transaction can be applied to a bound policy. For a general overview of how to conduct policy transactions using the system APIs, see "Policy transactions". For details on the business functionality of renewals and other policy transactions in PolicyCenter, see *Application Guide*.

Renew the policy

1. Initiate a renewal policy transaction.

   Submit a POST request to the `/policy/v1/policies/{policyId}/renew` endpoint, containing an empty request payload.

   ```
   {
       "data": {
           "attributes": {}
       }
   }
   ```

   The response payload contains the associated job. Use the job ID in subsequent calls.
2. Modify the job *(optional)*.

   The job is in Draft state, and can be modified. For example, the renewed policy could include a new coverable. Details of modifying a job depend on the LOB and product.
3. Generate a quote.

   Submit a business action POST to the `/job/v1/jobs/{jobId}/quote` endpoint.
4. Complete the policy transaction.

   The following table illustrates the various ways that a quoted renewal job can be completed:

Transaction outcome	Condition	Endpoint
Accepted renewal	Issues the renewal immediately	`/job/v1/jobs/{jobId}/bind-and-issue`
Pending renewal	Puts policy period in `renewing` status and starts the `PendingRenwalWF` workflow	`/job/v1/jobs/{jobId}/pending-renew`
Pending non-renewal	Puts policy period in `nonrenewing` status and starts the `PendingNonRenwalWF` workflow	`/job/v1/jobs/{jobId}/pending-non-renew`
Pending not taken	Puts policy period in `nottaking` status and starts the `PendingNotTakenWF` workflow	`/job/v1/jobs/{jobId}/pending-not-take`

Transaction outcome	Condition	Endpoint
Withdrawn	Withdraws the job	`/job/v1/jobs/{jobId}/withdrawn`

Submit a business action POST to the relevant endpoint.

chapter 17

Cancellation

The cancellation policy transaction can be applied to a bound policy that is in force. A cancellation request must include the requesting source (insurer or insured), a reason for the cancellation, and an effective date. The effective date of the cancellation will depend on several factors, related to the requesting source and product configuration. For a general overview of how to conduct policy transactions using the system APIs, see "Policy transactions" on page 131. For details on cancellations and other policy transactions in PolicyCenter, see *Application Guide*.

Cancel the policy

1. Initiate the cancellation policy transaction.

   Submit a business action POST to the `/policy/v1/policies/{policyId}/cancel` endpoint.

   The request payload must contain at minimum the following fields and value types:

   - `cancellationReasonCode.code`: A typecode from the `ReasonCode` typelist
   - `cancellationSource.code`: A typecode from the `CancellationSource` typelist
   - `jobEffectiveDate`: A date string

```
{
    "data": {
        "attributes": {
            "cancellationReasonCode": {
                "code": "cancel"
            },
            "cancellationSource": {
                "code": "carrier"
            },
            "jobEffectiveDate": "2020-08-15"
        }
    }
}
```

   The response payload contains the associated job, which is in Quoted state. Use the job ID in the subsequent call.

2. Complete the cancellation.

   The following table illustrates the various ways that a quoted cancellation job can be completed:

Transaction outcome	Action	Endpoint
Accepted	Cancel the policy immediately	`/job/v1/jobs/{jobId}/bind-and-issue`
Pending	Schedule a cancellation for a future date	`/job/v1/jobs/{jobId}/pending-cancel`
Rescheduled	Change the process date for the scheduled cancellation	`/job/v1/jobs/{jobId}/reschedule`

Transaction outcome	Action	Endpoint
Rescinded	Rescind the scheduled cancellation	`/job/v1/jobs/{jobId}/rescind`
Withdrawn	Withdraw the cancellation job	`/job/v1/jobs/{jobId}/withdrawn`

Submit a business action POST to the relevant endpoint.

chapter 18

Policy change

A policy change modifies a policy in between the effective and expiration dates. This transaction type can be used to apply changes to a policy, except for those that involve revising its effective date or the producer of record. For a general overview of how to conduct policy transactions using the system APIs, see “Policy transactions” on page 131. For details on the business functionality of policy changes and other policy transactions in PolicyCenter, see *Application Guide*.

Change the policy

1. Initiate a policy change transaction.

   Submit a business action POST to the `/policy/v1/policies/{policyId}/change` endpoint. The request payload must include the job effective date.

   ```
   {
       "data": {
           "attributes": {
               "jobEffectiveDate": "2020-08-18"
           }
       }
   }
   ```

   The response payload contains the associated job, which is in Draft state. Use the job ID in subsequent calls.
2. Modify the job as necessary.

   Modifying actions will depend on the LOB and product. For example, in the case of a personal auto line, a caller could add a driver or revise driving percentages of a vehicle.
3. Sync coverages and modifiers.

   To apply coverages, submit a business action POST to the `/job/v1/jobs/{jobId}/lines/{lineId}/sync-coverages` endpoint. To apply modifiers, submit a business action POST to the `/job/v1/jobs/{jobId}/lines/{lineId}/sync-modifiers` endpoint.
4. Generate a quote.

   Submit a business action POST to the `/job/v1/jobs/{jobId}/quote` endpoint.
5. Complete the policy change transaction.

   The following table illustrates the two ways that a quoted policy change job can be completed:

Transaction outcome	Condition	Job status	Endpoint
Accepted	The account holder and insurer agree to the quote, and all necessary paperwork has been received by the insurer. The policy is bound and issued.	Bound	`/job/v1/jobs/{jobId}/bind-and-issue`
Withdrawn	The quote contains erroneous data.	Withdrawn	`/job/v1/jobs/{jobId}/withdrawn`

Submit a business action POST to the relevant endpoint.

chapter 19

Reinstatement

The reinstatement policy transaction can only be applied to a canceled policy. A reinstatement is executed when the issue causing the cancellaton is rectified. For example, a reinstatement could occur when an insurer receives past-due payment on a policy but had been canceled for lack of payment. For a general overview of how to conduct policy transactions using the system APIs, see "Policy transactions" on page 131. For details on reinstatements and other policy transactions in PolicyCenter, see *Application Guide*.

Reinstate the policy

1. Initiate a reinstatement policy transaction.

   Submit a business action POST to the `/policy/v1/policies/{policyId}/reinstate` endpoint. The request payload must include the `reinstateCode.code` property, which takes a typecode from the `ReinstateCode` typelist.

   ```
   {
       "data": {
           "attributes": {
               "reinstateCode": {
                   "code": "payment"
               }
           }
       }
   }
   ```

   The response payload contains the associated job, which is in Draft state. Use the job ID in the subsequent call.
2. Generate a quote.

   Submit a business action POST to the `/job/v1/jobs/{jobId}/quote` endpoint. Reinstatement (job effective) date is automatically set to the cancellation date. Everything else in the original policy remains unchanged.
3. Complete the reinstatement policy transaction.

   The following table illustrates the two ways that a quoted reinstatement job can be completed:

Transaction outcome	Condition	Job status	Endpoint
Accepted	The account holder and insurer agree to the quote, and all necessary paperwork has been received by the insurer. The policy is bound and issued.	Bound	`/job/v1/jobs/{jobId}/bind-and-issue`
Withdrawn	The quote contains erroneous data.	Withdrawn	`/job/v1/jobs/{jobId}/withdrawn`

Submit a business action POST to the relevant endpoint.

chapter 20

Rewrite and Rewrite New Account

The rewrite and rewrite new account policy transactions can be applied by an underwriter to a bound policy that is in force. The rewrite policy transaction can be used to change the effective date of a policy or the producer of record. The rewrite new account policy transaction can be used to clone an existing policy and assign it to a new account. These actions cannot be executed in a policy change transaction. For a general overview of how to conduct policy transactions using the system APIs, see "Policy transactions" on page 131. For details on rewrites and other policy transactions in PolicyCenter, see *Application Guide*.

There are three types of policy transaction rewrites:

- *Full term rewrite*: Overwrites the entire term of an existing policy
- *New term rewrite*: Creates a new term for the policy
- *Mid-term rewrite*: Rewrites the remainder of an existing policy term

The full term and new term policy rewrites are called *flat rewrites*.

With the system APIs, a policy rewrite transaction is preceded by a cancellation policy transaction. When canceling a policy in preparation for a rewrite, the value provided for the `cancellationReasonCode` property must be either `flatrewrite` or `midtermrewrite`, as described above. When rewriting the policy, the value provided for the `rewriteType` property must be either `rewriteFullTerm`, `rewriteNewTerm`, or `rewriteRemainderOfTerm`, as described above.

Rewrite period	`cancellationReasonCode` property value	`rewriteType` property value
Full term	`flatrewrite`	`rewriteFullTerm`
New term	`flatrewrite`	`rewriteNewTerm`
Mid-term	`midtermrewrite`	`rewriteRemainderOfTerm`

Rewrite transaction

The rewrite policy transaction can be executed on a canceled policy.

1. Initiate the rewrite policy transaction.
   - Submit a POST request to the `/policy/v1/policies/{policyId}/rewrite` endpoint
   - The request payload must contain a value for the `rewriteType` property. Acceptable values are `rewriteFullTerm`, `rewriteNewTerm`, and `rewriteRemainderOfTerm`.

```
{
  "data": {
    "attributes": {
      "rewriteType": {
```

```
        "code": "RewriteFullTerm"
      }
    }
  }
}
```

- The response payload contains the associated job, which is in Draft state. Use the job ID in subsequent calls.

2. Revise the job as needed, to reflect changes to the policy.
3. Generate a quote.

   Submit a business action POST to the `/job/v1/jobs/{jobId}/quote` endpoint.
4. Complete the policy transaction.

   Submit a business action POST to the `/job/v1/jobs/{jobId}/bind-and-issue` endpoint.

Rewrite new account transaction

1. Initiate the rewrite new account policy transaction.

   Submit a POST request to the `/policy/v1/policies/{policyId}/rewrite-account` endpoint. The request payload must contain a valid account ID value for the `account.id` property.

```
{
  "data": {
    "attributes": {
      "account": {
        "id": "pc:102"
      }
    }
  }
}
```

   The response payload contains the associated job, which is in Draft state. Use the job ID in subsequent calls.
2. Revise the job as needed, to reflect changes to the policy.
3. Generate a quote.

   Submit a business action POST to the `/job/v1/jobs/{jobId}/quote` endpoint.
4. Complete the policy transaction.

   Submit a business action POST to the `/job/v1/jobs/{jobId}/bind-and-issue` endpoint.

chapter 21

Out-of-sequence conflicts

Out-of-sequence policy transactions are policy transactions with an effective date that is before the effective date of a previous policy transaction on the same policy. Insurers sometimes call these situations out-of-sequence endorsements. PolicyCenter uses the term *out-of-sequence*.

An out-of-sequence conflict occurs when a policy change has a transaction date later than another policy transaction, but an effective date earlier than that other policy transaction.

The Job API provides endpoints that can be used to identify and resolve out-of-sequence conflicts on policy transactions.

Identifying out-of-sequence conflicts

Policy changes are applied to jobs that are in Draft state, which is when a potential out-of-sequence conflict could be introduced.

While the job is in Draft state, no out-of-sequence warnings are raised, even when a conflicting change is added. When a caller attempts to quote a job that contains one or more out-of-sequence conflicts, a 400 response will be returned with a message informing the caller of the presence of conflicts.

To identify out-of-sequence conflicts on a job, a caller can submit a GET request to the `/job/v1/jobs/{jobId}/oos-conflicts` endpoint. If are no conflicts, then the request will return an empty response.

For a job that has out-of-sequence conflicts, the response body will contain a `conflicts` property, which is an array holding one or more conflict objects.

The following example shows an out-of-sequence conflict for date of birth values set for Bill Preston:


```
{
    "data": {
        "attributes": {
            "conflicts": [
                {
                    "conflictValues": [
                        {
                            "displayValue": "01/01/1975",
                            "effectiveDate": "2019-03-01T00:01:00.000Z"
                        },
                        {
                            "displayValue": "01/01/1977",
                            "effectiveDate": "2019-04-01T00:01:00.000Z"
                        }
                    ],
                    "entity": {
                        "displayName": "Bill Preston",
                        "id": "pc:703",
                        "type": "PolicyContact",
                        "uri": "/job/v1/jobs/pc:204/contacts/pc:703"
```

```
                    },
                    "field": "dateOfBirthInternal",
                    "id": "b0aca4bc",
                    "originalValue": "01/01/1970",
                    "yourValue": "01/01/1980"
                }
            ]
        },
        . . .
    }
}
```

The response body contains the following fields:

- `conflictValues`: An array of objects for each conflict, containing two or more objects
- `displayValue`: A value that is in conflict
- `effectiveDate`: The effective date of the associated value
- `entity`: The target Guidewire entity
- `field`: The field that has the conflict
- `id`: The unique identifier for the conflict
- `originalValue`: The original field value prior to the conflict
- `yourValue`: The latest field value applied by the caller that gave rise to the conflict

Resolving out-of-sequence conflicts

To resolve out-of-sequence conflicts on a job, a caller must submit a POST request to the `/job/v1/jobs/{jobId}/oos-conflicts/resolve` endpoint. The request body must contain an `overrides` field, which is an array of override selections for resolving the conflicts. All conflicts must be resolved in the POST.

To retain the original field values for each conflict, thus discarding all subsequent changes, a caller can submit a request body that contains an empty array for the `overrides` field:

```
{
  "data": {
    "attributes": {
      "overrides": [ ]
    }
  }
}
```

If the caller wishes to keep one or more changes to any conflicted fields, then the request body must explicitly declare value selections for all fields that are in conflict. In this case, the

```
overrides
```

array must contain an object for each conflict, each having the following properties:

- `id`: The conflict ID, as a string
- `resolution`: A string value of either `acceptYours` or `discardYours`

As previously mentioned, each conflict object contains an `originalValue` and `yourValue` field. When resolving conflicts, a caller must choose which of those values to retain. To keep the original value, the caller would apply the `discardYours` value to the `resolution` property. Otherwise, the caller would apply the `acceptYours` value to the property.

In the following example request body, the conflicting birthdates for Bill Preston are resolved by choosing the value indicated by the `yourValue` field in the initial conflict object:

```
{
  "data": {
    "attributes": {
      "overrides": [
        "id": "b0aca4bc",
        "resolution": "acceptYours"
      ]
    }
  }
}
```

Example: Identifying and resolving out-of-sequence conflicts

It is possible to create multiple out-of-sequence conflicts on a job. This example shows how to resolve three conflicts on the same job.

Consider the following sequence:

1. In the initial job, an account was created for Bill Preston, his date of birth was set to January 1, 1970, and his address was set to `null`. Additionally, the number of employees at his company was set at 0.
2. In the first change to the job, Bill's date of birth was revised to January 1, 1975, and he was given an address, the first line of which is 24 Appletree Rd.
3. In the second change, Bill's date of birth was revised to January 1, 1977, and the number of employees at his company was raised to 2.
4. In the third change, Bill's date of birth was revised yet again, to January 1, 1980, and the first line of his address was changed to 88 Maple Lane. Also, the number of employees at his company was changed to `null`.

The above information, in table form:

Sequence	Date of birth	Address	Employee count
1	1970-01-01	null	0
2	1975-01-01	24 Appletree Rd.	
3	1977-01-01		2
4	1980-01-01	88 Maple Lane	null

After these changes have been submitted, a call to `/job/v1/jobs/{jobId}/oss-conflicts` returns the following response:


```
{
  "data": {
    "attributes": {
      "conflicts": [
        {
          "conflictValues": [
            {
              "displayValue": "01/01/1975",
              "effectiveDate": "2019-03-01T00:01:00.000Z"
            },
            {
              "displayValue": "01/01/1977",
              "effectiveDate": "2019-04-01T00:01:00.000Z"
            }
          ],
          "entity": {
            "displayName": "Bill Preston",
            "id": "pc:703",
            "type": "PolicyContact",
            "uri": "/job/v1/jobs/pc:204/contacts/pc:703"
          },
          "field": "dateOfBirthInternal",
          "id": "b0aca4b",
          "originalValue": "01/01/1970",
          "yourValue": "01/01/1980"
        },
        {
          "conflictValues": [
            {
              "displayValue": "Address Line 1; Future Change",
              "effectiveDate": "2019-03-01T00:01:00.000Z"
            },
            {
              "displayValue": "Address Line 1; Future Change",
              "effectiveDate": "2019-04-01T00:01:00.000Z"
            }
          ],
          "entity": {
            "displayName": "1: Address Line 1; OOS Change, CA",
            "id": "201",
            "type": "PolicyLocation",
            "uri": "/job/v1/jobs/pc:204/locations/201"
          },
          "field": "addressLine1Internal",
          "id": "df78662",
          "originalValue": "",
```

```
          "yourValue": "Address Line 1; OOS Change"
        },
        {
          "conflictValues": [
            {
              "displayValue": "2",
              "effectiveDate": "2019-04-01T00:01:00.000Z"
            }
          ],
          "entity": {
            "displayName": "1: Address Line 1; OOS Change, CA",
            "id": "201",
            "type": "PolicyLocation",
            "uri": "/job/v1/jobs/pc:204/locations/201"
          },
          "field": "employeeCountInternal",
          "id": "47761d4",
          "originalValue": "0",
          "yourValue": ""
        }
      ]
    },
    . . .
}
```

The same response can be constrained using field selection, by appending `?fields=conflicts.id,conflicts.field,conflicts.originalValue,conflicts.yourValue` to the request URL:

```
{
    "data": {
        "attributes": {
            "conflicts": [
                {
                    "field": "dateOfBirthInternal",
                    "id": "b0aca4b",
                    "originalValue": "01/01/1970",
                    "yourValue": "01/01/1980"
                },
                {
                    "field": "addressLine1Internal",
                    "id": "df78662",
                    "originalValue": "",
                    "yourValue": "88 Maple Lane"
                },
                {
                    "field": "employeeCountInternal",
                    "id": "47761d4",
                    "originalValue": "0",
                    "yourValue": ""
                }
            ]
        }
    }
}
```

Note that in the response object the JSON `null` type is represented by an empty string.

To resolve these conflicts, choices must be made between the original values and the most recent values, and this information must be passed in the request body to POST `/job/v1/jobs/{jobId}/oss-conflicts/resolve`:

```
{
  "data": {
    "attributes": {
      "overrides": [
        {
          "id": "b0aca4b",
          "resolution": "acceptYours"
        },
        {
          "id": "df78662",
          "resolution": "acceptYours"
        },
        {
          "id": "47761d4",
          "resolution": "discardYours"
        }

      ]
    }
  }
}
```

Following this post, Bill's birthdate will be January 1, 1980, his address will be 88 Maple Lane, and the number of employees at his company will be set to "0".

chapter 22

Policy and job search

The Job and Policy APIs provide endpoints that can be used to search for jobs or policies, respectively:

- `/job/v1/search/jobs`
- `/policy/v1/search/policies`

To execute a search, a caller can submit a POST request to a search endpoint. The request payload must contain at least one of the following properties:

- `companyName`: Complete company name of the insured
- `firstName` and `lastName`: Complete first and last name of the insured
- `jobNumber` *(job search only)*: The job number
- `officialId`: Official identification code
- `phone`: A phone number
- `policyNumber`: The policy number
- `producerCode.id`: A valid producer code

When searching for jobs, the caller must also include the `jobType.code` property, which denotes the type of job (such as submission or renewal).

The response payload of a search request can be further filtered by adding any of the following properties to the request:

- `city`: A city location associated with a job or policy
- `inForceOn` *(policy search only)*: A date string *(YYYY-MM-DD)* specifying a date on which the policy is in force
- `jurisdiction.code`: A jurisdiction code
- `postalCode`: A postal code associated with a job or policy
- `product.id`: A product ID for the LOB
- `state.code`: A state associated with a job or policy
- `street`: A street address associated with a job or policy

To get a count of the total number of matches for the search, add the query parameter `&includeTotal=true` to the request. For more information on controlling pagination, see “Controlling pagination” on page 54.

chapter 23

Multi-version quoting

Multi-version quoting is a PolicyCenter feature that is available on submission, policy change, renewal, and rewrite policy transactions. Within a single policy transaction, multiple quotes can be generated and compared. The Job API supports this functionality. For details on multi-version quoting and other policy transactions in PolicyCenter, see *Application Guide*.

Multi-version quoting is supported through the following Job API endpoints:

- `/job/v1/jobs/{jobId}/versions`
- `/job/v1/jobs/{jobId}/versions/{versionId}`
- `/job/v1/jobs/{jobId}/change-version`
- `/job/v1/jobs/{selectedJobId}?jobVersion={unselectedJobId}`

Job version properties

By default, a job has one version from the moment it is created, and that version is in selected state. To access the job versions, callers can submit a GET request to the `/job/v1/jobs/{jobId}/versions` collection.

The following code block shows the response payload of a GET request to the `/versions` collection of a newly created job, which contains one version:

```
{
    "count": 1,
    "data": [
        {
            "attributes": {
                "id": "pc:401",
                "name": "Version #1",
                "number": 1,
                "selected": true,
                "status": {
                    "code": "Draft",
                    "name": "Draft"
                }
            },
            . . .
        }
    ],
    . . .
}
```

The `attributes` field contains the following properties:

- `id` : The ID value of the version. The first version of a job will have the same ID value as the job itself.
- `name` : A human-readable string that can be used to identify the job version.

- `number` : The 1-based index number of the version in the versions collection. The property is syntactic sugar, providing a handle that applications can use when iterating over a versions array.
- `selected` : A Boolean value indicating the selected state of the version.
- `status` : A typekey value indicating the job status. Jobs are in Draft status from time of creation, or in Quoted status once a quote has been generated.

Other than `name`, these properties are read-only values.

Creating a new job version

A caller can create a new version of a job by submitting a POST request to the `/job/v1/jobs/{jobId}/versions` endpoint. At minimum, the request payload must contain data and attributes properties, and the latter can be empty:

```
{
  "data": {
    "attributes": { }
  }
}
```

Optionally, the caller can provide a name for the version:

```
{
  "data": {
    "attributes": {
        "name": "My new version"
     }
  }
}
```

On creation, the new job version is in Draft status, it is not selected, and the version number is incremented by one. The ID value is specific to the version.

```
{
    "data": {
        "attributes": {
            "id": "pc:402",
            "name": "My new version",
            "number": 2,
            "selected": false,
            "status": {
                "code": "Draft",
                "name": "Draft"
            }
        },
        . . .
    }
}
```

Selecting a job version

By default, the initial job and first version are the same, having the same ID value, and this version is selected. Subsequent versions are unselected by default.

A caller can select a job version by submitting a POST request to the `/job/v1/jobs/{jobId}/change-version` endpoint. The request payload must specify the version to be selected:

```
{
  "data": {
    "attributes": {
      "selectedVersion": {
        "id": "pc:402"
      }
    }
  }
}
```

In the payload above, the `selectedVersion.id` value is set to the ID value of the job version that was created in the previous section. This job version will now be selected, and the all other job versions will be in an unselected state.

Working with job versions

A job version that is selected and in Draft status can be modified or quoted through the `/job/v1/jobs/{selectedJobId}` endpoints.

Alternatively, a caller can modify or quote an unselected job version that is in Draft status by applying the `jobVersion` query parameter to the request. These calls have the following patterns:

- `/job/v1/jobs/{selectedJobId}?jobVersion={unselectedJobId}`
- `/job/v1/jobs/{selectedJobId}/{businessAction}?jobVersion={unselectedJobId}`

Typically, each version is modified until considered done and then quoted. The quoted versions can then be offered for side-by-side comparison. To further modify a quoted version, a caller can apply the `/make-draft` business action to that version, make the modification, and then re-quote.

chapter 24

Working with contingencies

In some situations, an insurer may need to identify that a job that is in progress has an issue to resolve, but it might be appropriate to resolve the issue after the job is bound. Similarly, an insurer may need to identify that a policy has an issue to resolve, even though there are no active jobs on the policy. For example, the rating for a personal auto submission may include a good driver discount that requires a report on the driver's driving history. The insurer could choose to bind the policy before the driving history has been received.

When situations like this occur, these outstanding issues can be managed through contingencies. A *contingency* is an object that identifies an issue affecting a job or policy that must be resolved by a particular date. Depending on the outcome, the insurer may opt to change or cancel the policy. Contingencies can have associated activities, documents, and notes.

For more information on the business functionality of contingencies, refer to the *Application Guide*.

Cloud API provides a set of endpoints for managing contingencies. These endpoints exist in both the Job API and the Policy API.

Querying for contingencies

Querying for contingencies

Use the following endpoints to retrieve information about existing contingencies.

Endpoint	Description
GET `/jobs/{jobId}/contingencies`	Retrieve all contingencies for the given job.
GET `/jobs/{jobId}/contingencies/{contingencyId}`	Retrieve the given contingency associated with the given job.
GET `/policies/{policyId}/contingencies`	Retrieve all contingencies for the given policy.
GET `/policies/{policyId}/contingencies/{contingencyId}`	Retrieve the given contingency associated with the given policy.

Contingencies use the same schema, whether they are associated with jobs or with policies.

For example, the following is the attributes portion of the payload for contingency pc:70 associated with policy pc:62:

```
GET /policy/v1/policies/pc:62/contingencies/pc:70

{
  ...
        "attributes": {
            "action": {
                "code": "ChangeRetroactively",
```

```
                "name": "Change policy retroactively"
            },
            "actionStartDate": "2021-11-04T00:00:00.000Z",
            "actionStarted": false,
            "createDate": "2021-11-04T23:26:33.069Z",
            "createUser": {
                "displayName": "Alice Applegate",
                "id": "pc:S8PjyiRN14A1u20zgJ76k",
                "type": "User",
                "uri": "/admin/v1/users/pc:S8PjyiRN14A1u20zgJ76k"
            },
            "description": "Driving record required for
                            good driver discount",
            "dueDate": "2021-11-11T00:00:00.000Z",
            "id": "pc:111",
            "status": {
                "code": "Pending",
                "name": "Pending"
            },
            "title": "Driving record required"
        },
...
```


Querying for related information

Use the following endpoints to retrieve information about activities, documents, and notes related to existing contingencies.

Contingencies associated with jobs

Endpoint	Description
GET `/jobs/{jobId}/contingencies/{contingencyId}/activities`	Retrieve the given activity associated with the given contingency on the given job.
GET `/jobs/{jobId}/contingencies/{contingencyId}/documents`	Retrieve the given document associated with the given contingency on the given job.
GET `/jobs/{jobId}/contingencies/{contingencyId}/notes`	Retrieve the given note associated with the given contingency on the given job.

Contingencies associated with policies

Endpoint	Description
GET `/policies/{policyId}/contingencies/{contingencyId}/activities`	Retrieve the given activity associated with the given contingency on the given policy.
GET `/policies/{policyId}/contingencies/{contingencyId}/documents`	Retrieve the given document associated with the given contingency on the given policy.
GET `/policies/{policyId}/contingencies/{contingencyId}/notes`	Retrieve the given note associated with the given contingency on the given policy.

Activities, documents, and notes use the same schema, whether they are primarily associated with account, jobs, policies, or contingencies.

For more information on working with activities, documents, or notes, see the following:

- “Working with activities” on page 201
- “Working with documents” on page 209
- “Working with notes” on page 215

Creating contingencies

Use the following endpoints to create a contingency associated with either a job or a policy:

- POST `/jobs/{jobId}/contingencies`
- POST `/policies/{policyId}/contingencies`

Minimum creation criteria

At a minimum, a contingency must specify:

- A `title`
- A `description`
- A `dueDate`
- An `action`, which must be set to a typecode from the `ContingencyAction` typelist

For example, the following request creates a new contingency for policy pc:62 with the given title and description. The due date is November 11, 2021. If the contingency is not resolved by then, PolicyCenter will take the action of retroactively changing the policy (to remove the good driver discount).

```
POST /policy/v1/policies/pc:62/contingencies

{
  "data": {
    "attributes": {
      "action": {
        "code": "ChangeRetroactively"
      },
      "description": "Driving record required for good driver discount",
      "dueDate": "2021-11-11T00:00:00.000Z",
      "title": "Driving record required"
    }
  }
}
```

Creating related objects

Use the following endpoint to create activities, documents, and notes for a contingency.

Contingencies associated with jobs

Endpoint	Description
POST /jobs/{jobId}/contingencies/{contingencyId}/activities	Create an activity associated with the given contingency on the given job.
POST `/jobs/{jobId}/contingencies/{contingencyId}/documents`	Create a document associated with the given contingency on the given job.
POST `/jobs/{jobId}/contingencies/{contingencyId}/notes`	Create a note associated with the given contingency on the given job.

Contingencies associated with policies

Endpoint	Description
POST `/policies/{policyId}/contingencies/{contingencyId}/activities`	Create an activity associated with the given contingency on the given policy.
POST `/policies/{policyId}/contingencies/{contingencyId}/documents`	Create a document associated with the given contingency on the given policy.
POST `/policies/{policyId}/contingencies/{contingencyId}/notes`	Create a note associated with the given contingency on the given policy.

When an activity is created directly for an account, job, or policy, you must specify an activity pattern. However, when an activity is created for a contingency, you must omit the activity pattern. All activities created through contingencies use the "New Contingency" activity pattern (`uw_review_contingency`). If the request payload contains an activity pattern, the request fails.

For more information on working with activities, documents, or notes, see the following:

- “Working with activities” on page 201
- “Working with documents” on page 209
- “Working with notes” on page 215

Closing contingencies

Initially, the status of each contingency is set to "Pending". While the status is pending, the contingency is considered to be open.

A contingency can be closed either by resolving it or by waiving it. You can both resolve and waive contingencies through Cloud API. Except for the endpoint paths and the final status of the contingency, the behaviors of the two operations are the same.

Resolving contingencies

To resolve a contingency, use one of the following endpoints as appropriate:

- POST `/jobs/{jobId}/contingencies/{contingencyId}/resolve`
- POST `/policies/{policyId}/contingencies/{contingencyId}/resolve`

The `/resolve` endpoints do not require a request body.

When you resolve a contingency:

- `status` is set to "Resolved".
- `closedDate` is set to the current time.
- `closedUser` is set to the API call's session user.

For more information on how the session user is determined, see the *Cloud API Authentication Guide*.

Waiving contingencies

To waive a contingency, use one of the following endpoints as appropriate:

- POST `/jobs/{jobId}/contingencies/{contingencyId}/waive`
- POST `/policies/{policyId}/contingencies/{contingencyId}/waive`

The `/waive` endpoints do not require a request body.

When you waive a contingency:

- `status` is set to "Waived".
- `closedDate` is set to the current time.
- `closedUser` is set to the API call's session user.

For more information on how the session user is determined, see the *Cloud API Authentication Guide*.

chapter 25

Working with product definitions

he Product Definition API supports a set of endpoints for querying products, lines, and reference code data.

Products and lines

The product and line endpoints provide read-only access to product data models.

The `/productdefinition/v1/products/*` endpoints can be used to query products and their associated lines and questions.

The `/productdefinition/v1/lines/*` endpoints can be used to query lines and their associated coverables, coverages, exposures, and questions.

Reference code data

The `/productdefinition/v1/reference-data/*` endpoints can be used to query collections of reference code data, such as cost codes or class codes.

The following types of reference code data are supported:

Endpoint	Description	Resource	Guidewire entity
`/productdefinition/v1/reference-data/bop-class-codes`	BOP class codes	BOPClassCode	BOPClassCode
`/productdefinition/v1/reference-data/cost-codes`	Cost codes	CostCode	CostCode
`/productdefinition/v1/reference-data/industry-codes`	Industry codes	IndustryCode	IndustryCode
`/productdefinition/v1/reference-data/risk-classes`	Risk classes	RiskClass	RiskClass
`/productdefinition/v1/reference-data/tax-locations`	Tax locations	TaxLocation	TaxLocation
`/productdefinition/v1/reference-data/uw-issue-types`	Underwriting issue codes	UWIssueType	UWIssueType
`/productdefinition/v1/reference-data/wc-class-codes`	Workers' Comp class codes	WCClassCode	WCClassCode
`/productdefinition/v1/reference-data/wc-fed-liab-class-codes`	Workers' Comp federal liability class codes	WCFedLiabClassCode	WCFedLiabClassCode

chapter 26

Working with underwriting issues

The Job API provides endpoints that can be used to get, create, approve, reject, or reopen underwriting issues associated with a job. The Policy API provides endpoints that can be used to get underwriting issues associated with a policy.

Underwriting-related endpoints are typically available to callers whose roles pertain to an insurance underwriter or the producer associated with the target policy or job. The insurance underwriter roles includes such functions as underwriter, underwriter supervisor, and underwriter assistant.

For general guidance on underwriting processes and configuration in PolicyCenter, see *Configuration Guide*.

Underwriting issue resources

In the system APIs, a PolicyCenter underwriting issue is formatted as a UWIssue resource. This resource maps to the UWIssue entity.

An underwriting issue resource has the following basic properties:

- `active`: Is the underwriting issue active? (Boolean)
- `hasApprovalOrRejection`: Has the issue been approved or rejected? (Boolean)
- `id`: The resource ID, as a string
- `issueKey`: A synonym for resource ID, as a string
- `issueType.code`: The code for the underwriting issue type, from a `UWIssue.IssueType.Code` property
- `issueType.displayName`: The name of the underwriting issue type as it appears in PolicyCenter, from a `UWIssueType.IssueType.Name` property

  Note:

  In PolicyCenter, users with administrator or superuser privileges can access underwriting issue types and their associated data. From **Administration** > **Export Data** > **Export Administrative Data** select **Underwriter Issue Types**, and then click **Export**. This action returns an XML file that contains an array of `<UWissueType>` elements, each of which has a child `<Code>` element that maps to an underwriting issue resource `issueType.code` property.

Issue value types

In PolicyCenter, an underwriting issue is often associated with an issue value type and an acceptable range, as defined by the backing UWIssue entity. For example, an underwriting issue could be triggered on a personal auto line when coverage is requested for a driver who is under the age of 25. In this case, the underwriting issue value type is an age (an integer-based value), and the triggering range is a value below 25.

An underwriting issue resource supports properties associated with issue value types. The value type property used with a particular resource is determined by the required value type of the associated UWIssue entity.

Possible issue value types are integer, decimal, monetary amount, or states set, which can be declared using the following resource properties:

- `integerValue`: An integer value, passed as a string
- `decimalValue`: A decimal value, passed as a string
- `moneyValue`: A MonetaryAmount resource, containing `currency` and `amount` properties that specify the appropriate currency and amount
- `setStateValue`: An ExclusiveStateSet resource, containing the `inclusionType` and `states` properties

If the underlying UWIssue entity does not require an issue value type, then the resource will not have one either. In such cases, the issue value type is considered to be unformatted or none.

Underwriting issue resources can also contain the following properties:

- `valueType`: A read-only property indicating the issue value type
- `comparator`: A read-only property indicating the comparator type, if applicable, corresponding to a ValueComparator typecode. This property is present on issues that require a value to be within some range.

Blocking points

An underwriting issue can have blocking points, meaning the associated job can proceed beyond the specified blocking point only after the underwriting issue is approved.

Blocking points are set by UWIssueBlockingPoint typecode values, and these are applied to the following underwriting issue resource properties:

- `blockingPoint`: The default blocking point set for the underwriting issue
- `approvalBlockingPoint`: When approving an issue, the blocking point can be reset, such as in the case when additional approvals are required. When no more approvals are required, this value is set to `NonBlocking`.
- `currentBlockingPoint`: The current blocking point for the issue

Resolution properties

Underwriting issues can be resolved through either approval or rejection. Underwriting issue resources that have been resolved will display properties appropriate for the type of resolution.

For all resolved issues, the `hasApprovalOrRejection` property will be set to `true`.

Approved issues have the following properties:

- `approvalDurationType`: A UWApprovalDurationType typecode value specifying the approval duration
- `approvingUser`: A SimpleReference for the user who approved the issue
- `autoApprovable`: Is the issue auto-approvable? (Boolean)
- `canEditApprovalBeforeBind`: Can the approval be edited before binding? (Boolean)
- `currentBlockingPoint`: On approval, the current blocking point is set to `NonBlocking`

When an approval is submitted with a corresponding issue value type, that value can be added to the resource through one of the following properties:

- `integerApprovalValue`: The approved integer value
- `decimalApprovalValue`: The approved decimal value
- `moneyApprovalValue`: The approved monetary amount
- `stateSetApprovalValue`: The approved states set

In such cases, the resource will subsequently contain a `displayApprovalValue` property, showing the approved value in string format.

Rejected issues have the following properties:

- `rejected`: A Boolean set to `true`
- `rejectingUser`: A SimpleReference for the user who rejected the issue
- `currentBlockingPoint`: On rejection, the current blocking point is set to `Rejected`

Getting underwriting issues

The Job and Policy APIs contain endpoints that can be used to retrieve underwriting issues. Authorized callers are those having an underwriter role with the insurer organization or a producer role associated with the specific job.

The Job API provides the following endpoints:

- GET `/job/v1/jobs/{jobId}/uw-issues`: Returns a collection of underwriting issues associated with the job
- GET `/job/v1/jobs/{jobId}/uw-issues/{uwIssueId}`: Returns the specified underwriting issue
- GET `/job/v1/jobs/{jobId}/uw-issues/{uwIssueId}/history`: Returns an array of resources that trace the state history of the specified underwriting issue

The Policy API provides the following endpoints:

- GET `/policy/v1/policies/{policyId}/uw-issues`: Returns a collection of underwriting issues associated with the policy
- GET `/policy/v1/policies/{policyId}/uw-issues/{uwIssueId}`: Returns the specified underwriting issue
- GET `/policy/v1/policies/{policyId}/uw-issues/{uwIssueId}/history`: Returns an array of resources that trace the state history of the specified underwriting issue

For details on underwriting resource content, see “Underwriting issue resources” on page 191.

Getting underwriting history

The Job and Policy APIs contain endpoints that can be used to retrieve the history of an underwriting issue associated with a job or policy, respectively. Authorized callers are those having an underwriter role with the insurer organization or a producer role associated with the specific job or policy.

Underwriting history resources can be retrieved through the following endpoints:

- GET `job/v1/jobs/{jobId}/uw-issues/{uwIssueId}/history`
- GET `policy/v1/policies/{policyId}/uw-issues/{uwIssueId}/history`

Calls to these endpoints return an array of UWIssueHistory resources that trace the state history of the specified underwriting issue. A UWIssueHistory resource contains many of the same properties as a UWIssue resource, along with the following additional properties:

- `createDate`: A date-time string for the creation date
- `effectiveDate`: A date-time string for the effective date
- `status`: A UWIssueHistoryStatus typecode value for issue history status

Creating underwriting issues

To create an underwriting issue on a job, a caller can submit a POST request to the `/job/v1/jobs/{jobId}/uw-issues` endpoint. Authorized callers are those having an underwriter role with the insurer organization.

At minimum, the POST request must include the underwriting issue type in the `issueType.code` field. The request body can also include a value for the associated underwriting issue value type. For this latter value, the following properties are supported:

- `integerValue`: An integer value, passed as a string
- `decimalValue`: A decimal value, passed as a string
- `moneyValue`: A MonetaryAmount resource, containing `currency` and `amount` properties that specify the appropriate currency and amount

- `setStateValue`: An ExclusiveStateSet resource, containing the `inclusionType` and `states` properties

The code block below shows a complete request body for creating an underwriting issue type of `TSTManualStateSet` that explicitly includes the state of California:

```
{
  "data": {
    "attributes": {
      "issueType": {
        "code": "TSTManualStateSet"
      },
      "stateSetValue": {
        "inclusionType": "inclusive",
        "states": [
          "CA"
        ]
      }
    }
  }
}
```

The response body contains the newly created UWIssue resource:

```
{
  "data": {
    "attributes": {
      "active": true,
      "autoApprovable": true,
      "blockingPoint": {
        "code": "BlocksQuote",
        "name": "Blocks Quote"
      },
      "comparator": {
        "code": "State_Set",
        "name": "In set"
      },
      "currentBlockingPoint": {
        "code": "BlocksQuote",
        "name": "Blocks Quote"
      },
      "displayValue": "CA",
      "hasApprovalOrRejection": false,
      "id": "45",
      "issueKey": "45",
      "issueType": {
        "code": "TSTManualStateSet",
        "displayName": "TST issue with state set format"
      },
      "shortDescription": "Special Issue",
      "stateSetValue": {
        "inclusionType": "inclusive",
        "states": [
          "CA"
        ]
      },
      "valueType": "stateSet",
      "valueVariesAcrossSlices": false
    },
    . . .
  }
}
```

This resource represents an open (`active`) underwriting issue that requires the state value (`valueType`) to be set to California (`stateSetValue`). This issue blocks on quote (`blockingPoint`), and at present the job associated with this issue cannot be quoted until the issue is resolved (`currentBlockingPoint`). The issue has not yet been approved or rejected (`hasApprovalOrRejection`).

For details on underwriting resource content, see “Underwriting issue resources” on page 191.

Approving underwriting issues

To approve an underwriting issue, authorized callers can submit a business action POST to the `/job/v1/jobs/{jobId}/uw-issues/{uwIssueId}/approve` endpoint.

Alternatively, authorized callers can apply special approval permission to an issue by submitting a business action POST to the `/job/v1/jobs/{jobId}/uw-issues/{uwIssueId}/special-approve` endpoint.

By default, only the superuser role has authorization to execute the above actions.

Approval types

An approval can be classified into the following four types:

- *Full approval*: The underwriting issue has been reviewed and approved, and if necessary it contains a value that is within the expected value range.
- *Partial approval*: The underwriting issue has been reviewed and approved, but its value is outside of the expected value range. Typically, a partially approved issue will be forwarded to a higher authority for final approval.
- *Automatic approval*: The underwriting issue is configured to support automatic approval when specified criteria are met. Such an issue has its `autoApprovable` resource property set to `true`.
- *Special approval*: Permission is provided for approving an issue regardless of value, as an approval override. Guidewire recommends that you never use this permission unless there is a time-critical policy that cannot be advanced by any underwriter.

Submitting approval requests

When calling an approval endpoints, at minimum the request body must contain the following content:

```
{
  "data": {
    "attributes": {
    }
  }
}
```

If the underwriting issue expects a formatted value, then the request can include an issue value type of either integer, decimal, monetary amount, or states set through one of the following properties:

- `integerApprovalValue`: An integer value, passed as a string
- `decimalApprovalValue`: A decimal value, passed as a string
- `moneyApprovalValue`: A MonetaryAmount resource, containing `currency` and `amount` properties that specify the appropriate currency and amount
- `stateSetApprovalValue`: An ExclusiveStateSet resource, containing the `inclusionType` and `states` properties

Optionally, the caller can also set values for the following properties:

- `approvalBlockingPoint`: A UWIssueBlockingPoint typecode value. This can be applied to set a subsequent blocking point
- `approvalDurationType`: A UWApprovalDurationType typecode value. This can be applied if the caller wishes to change the default duration type.
- `canEditApprovalBeforebind`: Can the approval be edited before binding? (Boolean)

After a successful call to an approval endpoint, the resource's `hasApprovalOrRejection` property will be set to `true`.

The following code block depicts an request for a `PAHighValueAuto` underwriting issue that approves a monetary amount of USD$8.50, sets the next blocking point on issuance, rescinds the approval duration, and enables editing of the underwriting issue prior to binding:

```
{
  "data": {
    "attributes": {
      "approvalBlockingPoint": {
        "code": "BlocksIssuance"
      },
      "approvalDurationType": {
        "code": "Rescinded"
      },
      "canEditApprovalBeforeBind": true,
      "moneyApprovalValue": {
        "amount": "8.5",
        "currency": "usd"
      }
    }
  }
}
```

However, the approved monetary amount (USD$8.50) is less than the default threshold for this issue type (USD $10.00), therefore the issue will be partially approved.

The response body appears as follows:


```
{
  "data": {
    "attributes": {
      "active": true,
      "approvalBlockingPoint": {
        "code": "BlocksIssuance",
        "name": "Blocks Issuance"
      },
      "approvalDurationType": {
        "code": "Rescinded",
        "name": "Rescinded"
      },
      "approvingUser": {
        "displayName": "Super User",
        "id": "default_data:1",
        "type": "User",
        "uri": "/admin/v1/users/default_data:1"
      },
      "autoApprovable": true,
      "blockingPoint": {
        "code": "BlocksQuote",
        "name": "Blocks Quote"
      },
      "canEditApprovalBeforeBind": true,
      "comparator": {
        "code": "Monetary_LE",
        "name": "At most (monetary)"
      },
      "currentBlockingPoint": {
        "code": "BlocksQuote",
        "name": "Blocks Quote"
      },
      "displayApprovalValue": "$8.50",
      "displayValue": "$10.00",
      "hasApprovalOrRejection": true,
      "id": "407",
      "issueKey": "407",
      "issueType": {
        "code": "PAHighValueAuto",
        "displayName": "PA: High value vehicle"
      },
      "longDescription": "Test long description",
      "moneyApprovalValue": {
        "amount": "8.50",
        "currency": "usd"
      },
      "moneyValue": {
        "amount": "10.00",
        "currency": "usd"
      },
      "shortDescription": "Test short description",
      "valueType": "money",
      "valueVariesAcrossSlices": false
    },
    . . .
  }
}
```


A subsequent approval request is necessary in order to fully approve this issue. A caller with sufficiently elevated authority can approve this issue at the current value by submitting a call with the minimum request body described above. Alternatively, a caller can submit an approval request that contains a value within the expected range. In either case, the fully approved issue will return a response body showing that the blocking point has been removed.

Rejecting underwriting issues

To reject an underwriting issue, authorized callers can submit a business action POST to the `/job/v1/jobs/{jobId}/uw-issues/{uwIssueId}/reject` endpoint. Calls to this endpoint do not take a request body. By default, only superuser and underwriter roles are authorized to execute this request.

The reject action cannot be applied to underwriting issues that have been approved, special approved, or rejected. In these cases, the underwriting issue resource property `hasApprovalOrRejection` reads `true`. To enable rejection of an approved issue, it must first be reopened.

Policy transaction jobs that have one or more rejected underwriting issues cannot be completed. A rejected underwriting issue permanently blocks the job.

A rejected underwriting issue resource contains the following properties:

```
{
  "data": {
    "attributes": {
      . . .
      "currentBlockingPoint": {
        "code": "Rejected",
        "name": "Rejected"
      },
      "hasApprovalOrRejection": true,
      . . .
      "rejected": true,
      "rejectingUser": {
        "displayName": "Super User",
        "id": "default_data:1",
        "type": "User",
        "uri": "/admin/v1/users/default_data:1"
      },
      . . .
    },
    . . .
  }
}
```

Reopening underwriting issues

To reopen an underwriting issue, a caller can submit a business action POST to the `/job/v1/jobs/{jobId}/uw-issues/{uwIssueId}/reopen` endpoint. Calls to this endpoint do not take a request body. By default, only superuser and underwriter roles are authorized to execute this request.

The reopen action can be applied to underwriting issues that have been approved, special approved, or rejected. In these cases, the underwriting issue resource property `hasApprovalOrRejection` reads `true`. If the reopen call succeeds, then the `currentBlockingPoint` property will be reset to its default value and the `hasApprovalOrRejection` property will be set to `false`.

Locking jobs for underwriting review

A job can be locked to prevent changes to it while any associated underwriting issues are being reviewed. A locked job cannot be edited by users without the underwriting role. Following review, the job can be unlocked to reenable editing.

To apply an editing lock, an authorized caller can submit a business action POST to the `/job/v1/jobs/{jobId}/edit-lock` endpoint. Calls to this endpoint do not take a request body.

The resource of a locked job has the `editLocked` property set to `true`.

To release an editing lock on a job, an authorized caller can submit a business action POST to the `/job/v1/jobs/{jobId}/release-edit-lock` endpoint. Calls to this endpoint do not take a request body.

By default, only superuser and underwriter roles are authorized to use these endpoints.

For details on edit locking, see the *Configuration Guide*.

part 3

Business flows: Framework APIs

The *InsuranceSuite Cloud API* is a set of RESTful system APIs that expose functionality in PolicyCenter so that caller applications can request data from or initiate action within PolicyCenter.

The following topics discuss how caller applications can initiate business flows related to the Common API and the Admin API. This includes:

- Working with activities
- Working with documents
- Working with notes
- Working with users
- Working with organizations
- Working with producer codes

chapter 27

Working with activities

An *activity* is an action related to the processing of an account, job, or policy that a user must attend to or be aware of. Activities are ultimately assigned to a group and a user in that group. This user has the primary responsibility for closing the activity.

Activities are typically created by users or by automatic PolicyCenter processes, and they are typically closed by users. But, activities can be both created and closed by system API calls.

For a complete description of the functionality of activities in PolicyCenter, see the *Application Guide*.

> **Note:** Activities exist in all InsuranceSuite applications. To ensure that activities behave in a common way across all applications, some activity endpoints, such as the endpoints for querying for or assigning activities, are declared in the Common API. Activities can also belong to accounts, jobs, and policies. These objects do not exist in all InsuranceSuite applications. This means that other activity endpoints, such as the endpoint for creating an activity for a job, are declared in the Account API, Job API, or Policy API. This topic always identifies the API in which each endpoint is declared.

Querying for activities

Activities cannot exist on their own. They must be attached to a parent object. In PolicyCenter, activities can be attached to accounts, jobs, or policies.

You can use the following endpoints to GET activities.

Endpoint	Returns
`/common/v1/activities`	All activities
`/common/v1/activities/{activityId}`	The activity with the given ID
`/account/v1/accounts/{accountId}/activities`	All activities associated with the given account
`/job/v1/jobs/{jobId}/activities`	All activities associated with the given job
`/policy/v1/policies/{policyId}/activities`	All activities associated with the given policy

Creating activities

Activities must be created from an existing account, job, or policy using one of the following endpoints:

- POST `account/v1/accounts/{accountId}/activities`
- POST `job/v1/jobs/{jobId}/activities`

- POST `policy/v1/policies/{policyId}/activities`

Activity patterns

Activities are created from activity patterns. An *activity pattern* is a set of default values for fields in the activity (such as description, subject, and priority). Every activity pattern has a code, such as `contact_insured` or `legal_review`.

When creating an activity through the system APIs, the only required field is `activityPattern`, which must specify the activity pattern's code. Because the activity pattern typically contains all the necessary default values, the activity pattern is often the only field the caller application needs to specify.

You can retrieve a list of activity patterns using the following:

- GET `/account/v1/accounts/{accountId}/activity-patterns`
- GET `/job/v1/jobs/{jobId}/activity-patterns`
- GET `/policy/v1/policies/{policyId}/activity-patterns`

You can optionally specify values for the following fields, each of which overrides the value coming from the activity pattern:

Field	Datatype	Description	Default
`description`	String	The activity description	Typically comes from the activity pattern
`dueDate`	datetime	The date by which the activity is expected to be completed	Typically calculated based on values in the activity pattern
`escalationDate`	datetime	The date on which the activity will be escalated if it has not yet been completed	Typically calculated based on values in the activity pattern
`externallyOwned`	Boolean	Whether the activity is to be assigned to an external group or external user	Typically comes from the activity pattern
`importance`	Typekey (`ImportanceLevel`)	The activity importance (as reflected on the user's calendar)	Typically comes from the activity pattern
`mandatory`	Boolean	Whether the activity must be completed (true) or can be skipped (false)	Typically comes from the activity pattern
`priority`	Typekey (`Priority`)	The activity's priority	Typically comes from the activity pattern
`recurring`	Boolean	Whether the activity repeats. If true, completing the activity creates a new one.	Typically comes from the activity pattern
`subject`	String	The activity subject	Typically comes from the activity pattern

Examples of creating activities

The following is an example of creating a `contact_insured` activity for account pc:10. The activity defaults to all values in the `contact_insured` activity pattern.

```
POST /account/v1/accounts/pc:10/activities

{
  "data": {
    "attributes": {
      "activityPattern": "contact_insured"
    }
  }
}
```

The following is an example of creating a `legal_review` activity for account pc:10. In this case, two activity pattern values are overridden: the activity is `mandatory` (it cannot be skipped) and the `priority` is urgent.

```
POST /account/v1/accounts/pc:10/activities

{
  "data": {
```

```
    "attributes": {
      "activityPattern": "legal_review",
      "mandatory": true,
      "priority": {
        "code": "urgent"
      }
    }
  }
}
```

Assigning activities

Ultimately, every activity is assigned to a group and a user in that group. This user has the primary responsible for closing the activity.

Activities can be temporarily assigned to queues. A *queue* is a repository belonging to a group which contains activities assigned to the group but not yet to any user in that group. Users in the group can then take ownership of activities manually as desired.

When you create an activity through the system APIs, PolicyCenter automatically executes the activity assignment rules to initially assign the activity to a group and user. You can use the `/{activityId}/assign` endpoint to reassign the activity as needed.

Assignment options

An activity can be assigned through the system APIs in the following ways:

- To a specific group and user in that group
- To a specific group only (and then PolicyCenter uses assignment rules to select a user in that group)
- To a specific group and queue
- To the user who holds a given role on the parent account, job, or policy
- By re-running the activity assignment rules
  - This can be appropriate if you have modified the activity since the last time assignment rules were run and the modification might affect who the activity would be assigned to.

The root resource for the `/{activityId}/assign` endpoint is `Assignee`. This resource specifies assignment criteria. The `Assignee` schema has the following fields:

Field	Type	Description
`autoAssign`	Boolean	Whether to assign the activity using assignment rules
`groupId`	string	The ID of the group to assign the activity to
`queueId`	string	The ID of the queue to assign the activity to
`role`	TypeKeyReference (`UserRole`)	The role on the parent object that identifies the user to assign the activity to
`userId`	string	The ID of the user to assign the activity to

The `Assignee` resource must specify an assignment option. It cannot be empty.

Assignment examples

When assigning activities to users, the user must be active and must have the "own activity" system permission.

Assigning to a specific group (and user)

The following payload assigns activity xc:1 to group demo_sample:31 and user demo_sample:1.

```
POST /common/v1/activities/xc:1/assign

{
  "data": {
    "attributes" : {
      "groupId" : "demo_sample:31",
      "userId" : "demo_sample:1"
```

```
    }
  }
}
```

The following payload assigns activity xc:1 to group demo-sample:31. Because no user has been specified, PolicyCenter will execute assignment rules to assign the activity to a user in group demo-sample:31.

```
POST /common/v1/activities/xc:1/assign

{
  "data": {
    "attributes" : {
      "groupId": "demo_sample:31"
    }
  }
}
```

Note that there is currently no endpoint that returns groups or group IDs. To assign activities to a specific group, the caller application must determine the group ID using some method other than a groups system API.

Assigning to a specific queue

The following payload assigns activity xc:1 to queue cc:32. Every queue is associated with a single group, so the activity will also be assigned to that group. Users in that group who have access to this queue can then manually take ownership of the activity.

```
POST /common/v1/activities/xc:1/assign

{
  "data": {
    "attributes" : {
      "queueId": "cc:32"
    }
  }
}
```

Note that there is currently no endpoint that returns queues or queue IDs. To assign activities to a queue, the caller application must determine the queue ID using some method other than a queues system API.

Assigning to a user with a given role

When an activity is assigned by role, PolicyCenter looks at the activity's parent (the account, job, or policy) and identifies the user with the given role. The activity is then assigned to that user. The role must be a code from the `UserRole` typelist.

For example, if an account activity is to be assigned to "`Underwriter`" and the underwriter for the account is Bruce Baker, the activity is assigned to Bruce Baker.

If you attempt to assign an activity by role and there is no user on the parent object with the given role, PolicyCenter:

- Identifies the user that created the object (This is the user with the "`Creator`" role.)
- Adds the given role to this user
- Assigns the activity to this user

For example, suppose there is an account created by Christine Craft. The account has no underwriter. If a system API attempts to assign an activity for this account to "`Underwriter`", PolicyCenter will add the underwriter role to Christine Craft and then assign the activity to her.

The following payload assigns activity xc:1 to the user on the activity parent that has the underwriter role.

```
POST /common/v1/activities/xc:1/assign

{
  "data": {
    "attributes" : {
       "role" : {
      "code" : "Underwriter"
       }
    }
  }
}
```

Using automated assignment

The following payload assigns activity xc:1 using automated assignment rules.

```
POST /common/v1/activities/xc:1/assign

{
  "data": {
    "attributes": {
        "autoAssign" : true
    }
  }
}
```

For more information on assignment rules, see the *Gosu Rules*.

Retrieving recommended assignees

When PolicyCenter users are assigning activities manually, the user interface includes a drop-down list of "recommended assignees". Typically, this list includes:

- The roles held by users on the parent object
- Users in the group the activity is currently assigned to.
- Any queues belonging to the group the activity is currently assigned to.

The contents of this drop-down list are generated by an application-specific `SuggestedAssigneeBuilder` class. You can access the same contents by executing a GET with one of the following `/assignee` endpoints:

Endpoint	Returns
`/common/v1/activity/{activityId}/assignee`	The list of suggested assignees for this activity
`/account/v1/accounts/{accountId}/activity-assignees`	The list of suggested assignees for activities on this account
`/job/v1/jobs/{jobId}/activity-assignees`	The list of suggested assignees for activities on this job
`/policy/v1/policies/{policyId}/activity-assignees`	The list of suggested assignees for activities on this policy

The following is a portion of an example response from the Common API's `/assignee` endpoint.

```
GET /common/v1/activities/pc:301/assignees

{
    "count": 5,
    "data": [
        {
            "attributes": {
                "name": "Creator",
                "role": {
                    "code": "Creator",
                    "name": "Creator"
                }
            }
        },
        {
            "attributes": {
                "groupId": "systemTables:1",
                "name": "Edward Lee (Enigma Fire & Casualty)",
                "userId": "pc:23"
            }
        },
        {
            "attributes": {
                "groupId": "systemTables:1",
                "name": "Alice Applegate (Enigma Fire & Casualty)",
                "userId": "pc:8"
            }
        },
        ...
    ],
    ...
```

Closing activities

An activity is closed either by completing it or skipping it. In order to be closed, the activity must be opened and assigned to a user.

When closing an activity, there are two options for the request payload:

- An empty payload
- A payload with an `included` note. (This option is used when you want to create a note while you close the activity. The payload has no data section, but it does have an `included` section.)

All endpoints for closing activities are in the Common API.

Completing an activity

Completing an activity indicates that the corresponding action has been taken or the assignee is aware of the corresponding issue.

The following payload completes activity xc:1.

```
POST /common/v1/activities/xc:1/complete

<no request payload>
```

The following payload completes activity xc:1 and creates a note.

```
POST /common/v1/activities/xc:1/complete

{
  "included": {
    "Note": [
      {
        "attributes": {
          "body": "This activity  was completed through a system API call."
        },
        "method": "post",
        "uri": "/common/v1/activities/xc:1/notes"
      }
    ]
  }
}
```

Skipping an activity

Skipping an activity indicates that there is no longer a need to take the corresponding action. Activities have a `mandatory` Boolean field. If this is set to true, the activity cannot be skipped.

The following payload skips activity xc:1.

```
POST /common/v1/activities/xc:1/skip

<no request payload>
```

The following payload skips activity xc:1 and creates a note.

```
POST /common/v1/activities/xc:1/skip

{
  "included": {
    "Note": [
      {
        "attributes": {
          "body": "This activity was skipped by a system API call."
        },
        "method": "post",
        "uri": "/common/v1/activities/xc:1/notes"
      }
    ]
  }
}
```

Additional activity functionality

The Common API contains these additional activity endpoints.

PATCHing activities

- PATCH `/activities/{activityId}`

Working with activity notes

- GET `/activities/{activityId}/notes`
- POST `/activities/{activityId}/notes`

For more information on notes, see “Working with notes” on page 215.

chapter 28

Working with documents

In PolicyCenter, a document is an electronic file (such as a PDF, Word document, or digital photograph) which contains information relevant to an account, job, or policy. Examples of documents could include photographs of covered jewelry, assessments from property inspectors, or correspondences with the insured. (Note that documents do not contain contractual parts of the policy. The policy contract is specified in PolicyCenter forms, not PolicyCenter documents.)

For a complete description of the functionality of documents in PolicyCenter, see the *Application Guide*.

> **Note:** Documents exist in all InsuranceSuite applications. To ensure that documents behave in a common way across all applications, some document endpoints, such as the endpoints for querying for document metadata or contents, are declared in the Common API. Documents can also belong to accounts, jobs, and policies. These objects do not exist in all InsuranceSuite applications. This means that other document endpoints, such as the endpoint for creating a document for a job, are declared in the Account API, Job API, or Policy API. This topic always identifies the API in which each endpoint is declared.

Overview of documents

Document owners

Documents cannot exist on their own. They must be attached to a parent object. From a system API perspective, PolicyCenter documents are always attached to accounts, jobs, or policies.

Document metadata and content

The PolicyCenter data model includes a `Document` entity. Instances of `Document` contain only *document metadata*, such as the author, MIME type, and status (draft, final, and so on).

Document contents are stored in and managed by a Document Management System. PolicyCenter is almost always integrated with a Document Management System so that users can upload documents and view and edit document contents.

> **Note:** The base configuration includes code to mimic Document Management System integration. This code is suitable for demonstration purposes, but it lacks the full range of features found in a Document Management System, such as versioning.

Documents can exist in PolicyCenter with metadata but no contents. For example, this may be appropriate when a document is a physical piece of paper retained by the insurer. The insurer may want to track the existence of the document and metadata about the document, even though the contents are not in the Document Management System.

Documents cannot exist in PolicyCenter with contents but no metadata.

Querying for document information

Querying for document metadata

You can use the following endpoints to GET document metadata.

Endpoint	Returns
`/common/v1/documents/{documentId}`	Metadata for the given document
`/account/v1/accounts/{accountId}/documents`	Metadata for documents on the given account
`/job/v1/jobs/{jobId}/documents`	Metadata for documents on the given job
`/policy/v1/policies/{policyId}/documents`	Metadata for documents on the given policy

The following is a portion of an example response from the Common API's `/documents/{documentId}` endpoint.

```
GET /common/v1/documents/xc:101

{
    "data": {
        "attributes": {
            "author": "Sue Smith",
            "dateModified": "2020-12-07T23:40:23.534Z",
            "docUID": "2020/11/7/235-53-425892/Legal Ownership of Property",
            "id": "xc:101",
            "inbound": false,
            "mimeType": "text/plain",
            "name": "Legal Ownership of Property",
            "obsolete": false,
            "section": {
                "code": "legal",
                "name": "Legal"
            },
            "status": {
                "code": "final",
                "name": "Final"
            },
            "type": {
                "code": "other",
                "name": "Other"
            }
        },
      ...
```

Querying for document content

You can use the following to GET document contents. However, these contents are base64 encoded and therefore are not human-readable until they are decoded.

- `/common/v1/documents/{documentId}/content`

The following is a portion of an example response from the Common API's `/documents/{documentId}/content` endpoint.

```
GET /common/v1/documents/xc:101/content

{
    "data": {
        "attributes": {
            "contents": "REVWIEJVSUxEDQoNCklmIHRoZXJlIGlzIG9ubHkgYSB2aXN1YWxp
emVkIHByb2R1Y3QNCiAgQU5EIHRoZSAiRW5hYmxlZCBmb3IgUmVzdCBBUEkiIGZpZWxkIGlzIGlua
XRpYWxseSBzZXQgdG8gRW5hYmxlZA0KICBBTkQgeW91IGNoYW5nZSB0aGUgZmllbGQgdG8gRGlzYW
JsZWQNCiAgVEhFTiB0aGUgcHJvZHVjdCBpcyBpbW1lZGlhdGVseSB1bmF2YWlsYWJsZSB0byB0aGU
gc3lzdGVtIEFQSXMNCg0KSWYgdGhlcmUgaXMgYSB2aXN1YWxpemVkIHByb2R1Y3QgYW5kIGEgZmlu
YWxpemVkIHByb2R1Y3QNCiAgQU5EIHRoZSAiRW5hYmxlZCBmb3IgUmVzdCBBUEkiIGZpZWxkIGlzI
GluaXRpYWxseSBzZXQgdG8gRW5hYmxlZA0KICBBTkQgeW91IGNoYW5nZSB0aGUgZmllbGQgdG8gRG
lzYWJsZWQNCiAgVEhFTiB0aGUgZmluYWxpemVkIHByb2R1Y3QgaXMgaW1tZWRpYXRlbHkgYXZhaWx
hYmxlIHRvIHRoZSBzeXN0ZW0gQVBJcw0KDQpDVVNUT01FUiBCVUlMRA0KDQpJZiB0aGVyZSBpcyBv
bmx5IGEgdmlzdWFsaXplZCBwcm9kdWN0DQogIEFORCB0aGUgIkVuYWJsZWQgZm9yIFJlc3QgQVBJI
iBmaWVsZCBpcyBpbml0aWFsbHkgc2V0IHRvIEVuYWJsZWQNCiAgQU5EIHlvdSBjaGFuZ2UgdGhlIG
ZpZWxkIHRvIERpc2FibGVkDQogIFRIRU4gdGhlIHByb2R1Y3QgaXMgaW1tZWRpYXRlbHkgW5hdmF
pbGFibGUgdG8gdGhlIHN5c3RlbSBBUElzDQooSSBhc3N1bWUgdGhpcyBpcyB0aGUgc2FtZSBhcyAN
Cg0KSWYgdGhlcmUgaXMgYSB2aXN1YWxpemVkIHByb2R1Y3QgYW5kIGEgZmluYWxpemVkIHByb2R1Y
3QNCiAgQU5EIHRoZSAiRW5hYmxlZCBmb3IgUmVzdCBBUEkiIGZpZWxkIGlzIGluaXRpYWxseSBzZX
```

```
QgdG8gRW5hYmxlZA0KICBBTkQgeW91IGNoYW5nZSB0aGUgZmllbGQgdG8gRGlzYWJsZWQNCiAgVEh
FTiB0aGUgZmluYWxpemVkIHByb2R1Y3QgaXMgaW1tZWRpYXRlbHkgYXZhaWxhYmxlIHRvIHRoZSBz
eXN0ZW0gQVBJcw==",
            "responseMimeType": "text/plain"
        },
    ...
```

POSTing documents

You can use the following to POST documents.

- `/account/v1/accounts/{accountId}/documents`
- `/job/v1/jobs/{jobId}/documents`
- `/policy/v1/policies/{policyId}/documents`

FormData objects

For most Cloud API resource, the request object is constructed as a body with a single string of JSON text. However, this format is not sufficiently robust for documents. When working with documents, the caller application must send two sets of data: the document metadata and the document contents. This is accomplished using `FormData` objects.

FormData is an industry-standard interface that construct an object as a set of key/value pairs. When a caller application is constructing a POST `/documents` call, the request object must be a FormData object with the following keys:

- `metadata`, whose value is a JSON string identifying the document metadata
- `content`, whose value is the contents of the document (and whose format varies based on the document type)

Two approaches to POSTing documents

There are two ways that a caller application can create a document:

1. POST both the document metadata and content to PolicyCenter using a `/documents` endpoint. In this approach:
   - PolicyCenter adds the document to the Document Management System through its own integration point.
   - The integration point is responsible for storing values created by the Document Management System in the document metadata (such as the document's DocUID).
2. Add the document to the Document Management System directly, and then POST the document metadata to PolicyCenter. In this approach:
   - The POST `/documents` call must provide any required information that comes from the Document Management System in the metadata (such as the document's DocUID).

Minimum creation criteria

When POSTing a document:

- The `metadata` JSON must include the following fields:
  - `name`
  - `status` (a typecode from the `DocumentStatusType` typelist)
  - `type` (a typecode from the `DocumentType` typelist)
- The `content` value is not required. For example, it may be appropriate to omit content when the document is a physical piece of paper that does not exist in the Document Management System, or when the caller application has added the document to the Document Management System directly.

Examples of POSTing documents

The following is an example of POSTing a "Property Assessment Report.pdf" file for account pc:10 through PolicyCenter.

```
POST /account/v1/accounts/pc:10/documents

Metadata:
{
```

```
    "data": {
        "attributes": {
            "name": "Property Assessment Report",
            "status": {
                "code": "draft"
            },
            "type": {
                "code": "letter_received"
            }
        }
    }
}

Contents:
<contents of "Property Assessment Report.pdf" file>
```

POSTing documents using Postman

About this task

From Postman, you can POST documents using FormData objects. When doing so, both the metadata and content must be stored in separate files referenced by the Postman call.

Note: Every POST `/documents` endpoint supports the ability to receive the metadata as either a string or a file. However, there is a known issue with Postman which prevents the sending of metadata as a string. When using Postman, the metadata can be sent only as file. This is described in the following procedure. (Client applications other than Postman may support both string and file.)

Procedure

1. Identify the files needed for the FormData object. This includes:
   - A text file that contains the metadata JSON.
   - The document file that has the content.
2. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
3. Under the **Untitled Request** label, select POST.
4. In the **Enter request URL** field, enter the URL for the server and the endpoint.
   - For example, to POST a document to an account on an instance of PolicyCenter on your machine, enter: `http://localhost:8180/pc/rest/account/v1/accounts/{accountId}/documents`
5. On the **Authorization** tab, specify authorization information as appropriate.
6. 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: **metadata**
   d) Click outside of the **metadata** 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 the file containing the JSON-formatted metadata.
   f) On the second line, for KEY, enter: **content**
   g) 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*.
   h) For VALUE, click the **Select Files** button and navigate to the file containing the document content.
7. Click **Send**. The response payload appears below the request payload.

PATCHing documents

You can use the following to PATCH documents.

- `/common/v1/documents/{documentId}`

Logically speaking, a PATCH call can modify only the metadata of a document, only the content of a document, or both.

PATCHing document metadata

Every PATCH `/{documentId}` call must include a `metadata` key/value pair, even if you want to modify only the content.

- If you want to modify the metadata, specify the fields to modify in the JSON referenced by the `metadata` key.
- If you do not want to modify the metadata, have the `metadata` key reference a payload with no attribute values, as shown below:

```
{
    "data": {
        "attributes": {
        }
    }
}
```

PATCHing document content

A PATCH `/{documentId}` call is not required to include a `content` key/value pair. If no `content` is specified, the PATCH will update the document metadata only.

PATCHes to content are destructive, not additive. If you specify `content`, the new content replaces the previous content entirely.

Examples of PATCHing documents

The following is an example of PATCHing only the metadata for document xc:101. In this example, the document status is changed to final.

```
PATCH /common/v1/documents/xc:101

Metadata:
{
    "data": {
        "attributes": {
            "status": {
                "code": "final"
            }
        }
    }
}

(No contents included with the API call)
```

The following is an example of PATCHing only the contents for document xc:101. Presumably, the contents of "Property Assessment Report.pdf" are different than the current contents of document xc:101.

```
PATCH /common/v1/documents/xc:101

Metadata:
{
    "data": {
        "attributes": {
            }
    }
}

Contents:
<contents of "Property Assessment Report.pdf" file>
```

The following is an example of PATCHing both the metadata for document xc:101 and the content.

```
PATCH /common/v1/documents/xc:101

Metadata:
{
    "data": {
        "attributes": {
            "status": {
                "code": "final"
            }
        }
    }
}
```

```
Contents:
<contents of "Property Assessment Report.pdf" file>
```

DELETEing documents

You can use the following to DELETE documents.

- `/common/v1/documents/{documentId}`

chapter 29

Working with notes

A *note* is a free-form record of the actions or thinking of a user or process. Notes are typically used to capture information that cannot be easily captured in some other way on some other business object. Notes are typically created by users, but they can be created by batch processes or other system behavior within PolicyCenter. They can also be created by caller applications using system APIs.

Through Cloud API, a note can be attached to an account, a job, or a policy. Notes can also be attached to activities.

For a complete description of the functionality of notes in PolicyCenter, see the *Application Guide*.

> **Note:** Notes exist in all InsuranceSuite applications. To ensure that notes behave in a common way across all applications, some note endpoints, such as the endpoints for querying for a note with a given ID, are declared in the Common API. Notes can also belong to accounts, jobs, and policies. These objects do not exist in all InsuranceSuite applications. This means that other note endpoints, such as the endpoint for querying for notes related to a given job, are declared in the Account API, Job API, or Policy API. This topic always identifies the API in which each endpoint is declared.

Querying for notes

You can use the following endpoints to GET notes.

Endpoint	Returns
`/common/v1/notes/{noteId}`	The note with the given ID (see below)
`/account/v1/accounts/{accountId}/notes`	All notes associated with the given account
`/job/v1/jobs/{jobId}/notes`	All notes associated with the given job
`/policy/v1/policies/{policyId}/notes`	All notes associated with the given policy
`/common/v1/activities/{activityId}/notes`	All notes associated with the given activity

The `/common/v1/notes/{noteId}` endpoint can be used to retrieve any note in PolicyCenter. This includes notes that are attached to a parent object other than an account, job, or policy.

Creating account, job, and policy notes

Notes must be created from an existing account, job, policy, or activity using one of the following endpoints:

- POST `account/v1/accounts/{accountId}/notes`
- POST `job/v1/jobs/{jobId}/notes`

- POST `policy/v1/policies/{policyId}/notes`
- POST `common/v1/activities/{activityId}/notes`

Be aware that PolicyCenter supports notes attached to other objects as well, such as PolicyPeriods. However, as of the current release, Cloud API does not yet support the ability to create notes for parent objects beyond accounts, jobs, policies, and activities.

The only field required for a note is `body`, which stores the note's text. You can optionally specify these fields:

Field	Datatype	Description	Default
`confidential`	Boolean	Whether the note is confidential	false
`securityType`	Typekey (`NoteSecurityType`)	The note's security type	NULL
`subject`	string	The note's subject	NULL
`topic`	Typekey (`NoteTopicType`)	The note's topic type	`general`

Minimal notes

The following is an example of creating a note for account pc:10.

```
POST /account/v1/accounts/pc:10/notes

{
  "data": {
    "attributes": {
        "body": "The insured's last name, Cahill, is pronounced 'KAH-hill', not 'KAY-hill'."
    }
  }
}
```

Notes with additional details

The following is an example of creating a detailed note for account pc:10.

```
POST /account/v1/accounts/pc:10/notes

{
  "data": {
    "attributes": {
        "body": "The insured's last name, Cahill, is pronounced 'KAH-hill', not 'KAY-hill'." ,
        "confidential": false,
        "securityType": {
          "code": "unrestricted"
        },
        "subject": "Pronunciation note",
        "topic": {
          "code": "general"
        }
    }
  }
}
```

Notes for an activity

The following is an example of creating a note for activity xc:22.

```
POST /common/v1/activities/xc:22/notes

{
  "data": {
    "attributes": {
        "body": "This activity was completed during a telephone call with the insured on 11/17."
    }
  }
}
```

Additional notes functionality

The Common API contains these additional notes endpoints.

PATCHing notes

- PATCH `common/v1/notes/{noteId}`

DELETEing notes

- DELETE `common/v1/notes/{noteId}`

chapter 30

Working with users

Cloud API provides endpoints in the Admin API to support several of the tasks associated with user management.

Note that the Admin API provides functionality for working with internal users. An *internal user* is a user who is known to PolicyCenter and who is listed in the PolicyCenter database (such as policy underwriters, claims adjusters, and billing clerks). This is not to be confused with *external users*, who are users known to PolicyCenter but who are not listed in the PolicyCenter database (such as account holders, policy holders, and service vendors). For information on working with external users, see the *Cloud API Authentication Guide*.

Querying for user information

Querying for users

To retrieve information about a user, you can use the following endpoints:

- GET /admin/v1/users
- GET /admin/v1/users/{userId}

For example, the following is the snippet of the response payload when retrieving the information for user pc:S-sAtIMQDbK0z3b2E7Mvw (Alice Applegate).

```
GET /admin/v1/users/pc:S-sAtIMQDbK0z3b2E7Mvw

{
    "data": {
        "attributes": {
            "active": true,
            "displayName": "Alice Applegate",
            "externalUser": false,
            "firstName": "Alice",
            "groups": [
                {
                    "displayName": "Los Angeles Branch UW",
                    "id": "pc:SSXQ8EaLxQq4LZ33Ia76r"
                },
                {
                    "displayName": "Eastern Region Underwriting",
                    "id": "pc:SVZTqTgdrHJKYQV9aGCbN"
                }
            ],
            "id": "pc:S-sAtIMQDbK0z3b2E7Mvw",
            "lastName": "Applegate",
            "organization": {
                "displayName": "Enigma Fire & Casualty",
                "id": "systemTables:1",
                "type": "Organization",
                "uri": "/admin/v1/organizations/systemTables:1"
            },
```

```
            "roles": [
                {
                    "displayName": "Reinsurance Manager",
                    "id": "reinsurance_manager",
                    "type": "Role"
                },
                {
                    "displayName": "Underwriter",
                    "id": "underwriter",
                    "type": "Role"
                }
            ],
            "useOrgAddress": true,
            "useProducerCodeSecurity": false,
            "userType": {
                "code": "underwriter",
                "name": "Underwriter"
            },
            "username": "aapplegate",
            "uwAuthorityProfiles": [
                {
                    "displayName": "Underwriter 1",
                    "id": "pc:underwriter1",
                    "type": "UWAuthorityProfile",
                    "uri": "/admin/v1/uw-authority-profiles/pc:underwriter1"
                }
            ],
            "vacationStatus": {
                "code": "atwork",
                "name": "At work"
            },
            "workPhone": {
                "displayName": "213-555-8164",
                "number": "2135558164"
            }
        }
    }
}
```

Querying for user roles

A *user role* is a named group of system permissions. User roles simplify the work of granting access by allowing a set of related system permissions to be assigned to a user.

You cannot create or modify user roles through Cloud API. However, you can retrieve information about user roles. This is useful when you want to create users and assign roles to them through Cloud API.

To retrieve information about user roles, use the following endpoints:

- GET /admin/v1/roles

For example, the following is the snippet of the response payload when retrieving the first five roles in the base configuration.

```
GET /admin/v1/roles?pageSize=5

{
    "count": 5,
    "data": [
        {
            "attributes": {
                "carrierInternal": true,
                "description": "Role with full permissions, deprecated"
                "displayName": "All Permissions (Deprecated)",
                "id": "superuser",
                "name": "All Permissions (Deprecated)",
                "roleType": {
                    "code": "user",
                    "name": "User Role"
                }
            }
        },
        {
            "attributes": {
                "carrierInternal": true,
                "description": "Permissions for audit examiner",
                "displayName": "Audit Examiner",
                "id": "audit_examiner",
                "name": "Audit Examiner",
                "roleType": {
                    "code": "user",
                    "name": "User Role"
                }
```

```
            }
        },
        {
            "attributes": {
                "carrierInternal": true,
                "description": "Permissions for audit supervisor",
                "displayName": "Audit Supervisor",
                "id": "audit_supervisor",
                "name": "Audit Supervisor",
                "roleType": {
                    "code": "user",
                    "name": "User Role"
                }
            }
        },
        {
            "attributes": {
                "carrierInternal": false,
                "description": "Base permissions for a claims adjuster",
                "displayName": "Claims Adjuster",
                "id": "claims_adjuster",
                "name": "Claims Adjuster",
                "roleType": {
                    "code": "userproducercode",
                    "name": "User Producer Code Role"
                }
            }
        },
        {
            "attributes": {
                "carrierInternal": true,
                "description": "Permissions for community admin",
                "displayName": "Community Admin",
                "id": "community_admin",
                "name": "Community Admin",
                "roleType": {
                    "code": "user",
                    "name": "User Role"
                }
            }
        }
    ]
}
```

Creating and updating users

Creating users

To create a user, use the following endpoint:

- POST `/admin/v1/users`

  Note: When a user is created through Cloud API, the user's password is set to a value that cannot be used for authentication. (The password is set to a value that is not a valid Base64 string, but the authentication framework can authenticate passwords only when they are valid Base64 strings.) In order for the new user to authenticate, the password must first be changed to a valid Base64 string through some other method, such as through the user interface.

Create a minimal user

The minimum creation criteria for a user is the `username`. For example, the following request creates a user with the user name "amartin".

```
{
  "data": {
    "attributes": {
      "username": "amartin"
    }
  }
}
```

The following is the response payload.

```
POST /admin/v1/users

{
```

```
    "data": {
        "attributes": {
            "active": true,
            "displayName": "",
            "externalUser": false,
            "id": "pc:SatEdbNuwVSfc2BvbG4g4",
            "organization": {
                "displayName": "Enigma Fire & Casualty",
                "id": "systemTables:1",
                "type": "Organization",
                "uri": "/admin/v1/organizations/systemTables:1"
            },
            "useOrgAddress": true,
            "useProducerCodeSecurity": false,
            "userType": {
                "code": "other",
                "name": "Other"
            },
            "username": "amartin",
            "vacationStatus": {
                "code": "atwork",
                "name": "At work"
            }
        },
        "checksum": "8b01f84c8076ba3f8c235cb2483cdbfb",
        "links": {
            "self": {
                "href": "/admin/v1/users/pc:SatEdbNuwVSfc2BvbG4g4",
                "methods": [
                    "get",
                    "patch"
                ]
            }
        }
    }
}
```


Create a typical user

You can specify additional information about a user as specified in the User schema. For example, the following payload creates a user with the following attributes:

- First name: Adriana
- Last name: Diaz
- User name: adiaz
- Employee number: ACME-02027
- Roles: audit examiner (`audit_examiner`) and audit supervisor (`audit_supervisor`)

```
POST /admin/v1/users

{
  "data": {
    "attributes": {
        "firstName": "Adriana",
        "lastName": "Diaz",
        "username": "adiaz",
        "employeeNumber": "ACME-02027",
        "roles" : [
            {
                "id": "audit_examiner"
            },
            {
                "id": "audit_supervisor"
            }

        ]
    }
  }
}
```

Updating users

Use the following endpoint to modify an existing user:

- PATCH `/admin/v1/users/{userId}`

Modifying user role assignment

You can use the PATCH `/admin/v1/users/{userId}` endpoint to assign or unassign roles to an existing user by modifying the `roles` array.

Note that, within Cloud API, PATCHing an array does not add the PATCH members to the members already existing in the array. Instead, the PATCH replaces the existing members with the PATCH members. If you want a PATCH to be additive to an array, you must first determine the existing members of the array, and then specify an array in the PATCH with the existing members as well as the ones you wish to add.

For example, suppose you have an existing user named Adriana Diaz with an ID of pc:111 and the following roles:

- audit examiner (`audit_examiner`)
- audit supervisor (`audit_supervisor`)

You want to add the Premium Auditor role (`premium_auditor`) to this user. To do so, you must use the following payload. (Note that the payload specifies the existing roles and the new role.)

```
PATCH /admin/v1/users/pc:111

{
  "data": {
    "attributes": {
      "roles": [
        {
          "id": "audit_examiner"
        },
        {
          "id": "audit_supervisor"
        },
        {
          "id": "premium_auditor"
        }
      ]
    }
  }
}
```

Underwriting authority profiles

An *underwriting issue* is an object attached to a policy transaction that indicates some aspect of the transaction requires review and approval by an underwriter. Underwriting issues are created automatically by underwriting rules.

An *underwriting authority profile* is a collection of authority grants that can be associated with one or more users. Each authority grant lists a type of underwriting issue, with an optional condition, that the associated users can approve. For example, a given underwriting authority profile could grant the ability to approve the following types of underwriting issues:

- Producer code changed (issue type with no condition)
- Claim total incurred is greater than $10,000 (issue type with a "> 10000" condition)

Through Cloud API, you can retrieve information about underwriting authority profiles and assign them to users. There is also limited functionality to create, update, and delete them.

Retrieving information about underwriting authority profiles

To retrieve information about underwriting authority profiles, use the following endpoints:

- GET `/admin/v1/uw-authority-profiles`
- GET `/admin/v1/uw-authority-profiles/{uwAuthorityProfileId}`

For example, the following payload retrieves information about the base configuration underwriting authority profiles. (Checksum and link information has been omitted.)

```
GET /admin/v1/uw-authority-profiles/

{
    "count": 9,
    "data": [
```

```
        {
            "attributes": {
                "description": "Agent 1",
                "displayName": "Agent 1",
                "id": "pc:agent1",
                "name": "Agent 1"
            }
        },
        {
            "attributes": {
                "description": "Agent 2",
                "displayName": "Agent 2",
                "id": "pc:SFWdb3rRQxDA9AyksL1p_",
                "name": "Agent 2"
            }
        },
        {
            "attributes": {
                "description": "External User Profile",
                "displayName": "External User Profile",
                "id": "authorityprofile:extuser",
                "name": "External User Profile"
            },
        },
        {
            "attributes": {
                "description": "Service User Profile",
                "displayName": "Service User Profile",
                "id": "authorityprofile:serviceuser",
                "name": "Service User Profile"
            }
        },
        {
            "attributes": {
                "description": "Unauthenticated User Profile",
                "displayName": "Unauthenticated User Profile",
                "id": "authorityprofile:uauser",
                "name": "Unauthenticated User Profile"
            }
        },
        {
            "attributes": {
                "description": "Underwriter 1",
                "displayName": "Underwriter 1",
                "id": "pc:underwriter1",
                "name": "Underwriter 1"
            }
        },
        {
            "attributes": {
                "description": "Underwriter 2",
                "displayName": "Underwriter 2",
                "id": "pc:underwriter2",
                "name": "Underwriter 2"
            }
        },
        {
            "attributes": {
                "description": "Underwriter Manager",
                "displayName": "Underwriter Manager",
                "id": "pc:SeOY1PmQwKcwQnuWvmCOy",
                "name": "Underwriter Manager"
            }
        },
        {
            "attributes": {
                "description": "internet portal",
                "displayName": "internet portal",
                "id": "pc:SZ6snOkKD_qhPHsqGgGwy",
                "name": "internet portal"
            }
        }
    ]
}
```


Assigning underwriting authority profiles to users

When you create or edit a user, you can specify the user's underwriting authority profiles. For example, the following payload creates a user with the following attributes:

- First name: Devon
- Last name: Byrne
- User name: dbyrne

- Roles: underwriter (`underwriter`)
- Underwriting authority profiles: Underwriter 1 (`pc:underwriter1`)

```
POST /admin/v1/users

{
  "data": {
    "attributes": {
        "firstName": "Devon",
        "lastName": "Byrne",
        "username": "dbyrne",
        "roles": [
            {
                "id": "underwriter"
            }
        ],
        "uwAuthorityProfiles": [
            {
                "id": "pc:underwriter1"
            }
        ]
    }
  }
}
```

Modifying underwriting authority profile assignment

Similar to modifying a user's roles, you can use the PATCH `/admin/v1/users/{userId}` endpoint to assign or unassign underwriting authority profiles to an existing user by modifying the `uwAuthorityProfiles` array.

Note that, within Cloud API, PATCHing an array does not add the PATCH members to the members already existing in the array. Instead, the PATCH replaces the existing members with the PATCH members. If you want a PATCH to be additive to an array, you must first determine the existing members of the array, and then specify an array in the PATCH with the existing members as well as the ones you wish to add.

For example, suppose you have an existing user named Devon Byrne with an ID of pc:235 and the Underwriter 1 underwriting authority profile (`pc:underwriter1`). You want to add the Underwriter 2 underwriting authority profile (`pc:underwriter2`) to this user. To do so, you must use the following payload. (Note that the payload specifies the existing profile and the new profile.)

```
PATCH /admin/v1/users/pc:235

{
  "data": {
    "attributes": {
      "uwAuthorityProfiles": [
        {
          "id": "pc:underwriter1"
        },
        {
          "id": "pc:underwriter2"
        }
      ]
    }
  }
}
```

Creating, updating, and deleting underwriting authority profiles

You can use the following endpoints to manage underwriting authority profiles:

- POST `/admin/v1/uw-authority-profiles`
- PATCH `/admin/uw-authority-profiles/{uwAuthorityProfileId}`
- DELETE `/admin/v1/uw-authority-profiles/{uwAuthorityProfileId}`

Note that these endpoints use the `UWAuthorityProfile` schema, which is designed primarily for assigning existing underwriting authority profiles to users. This schema has only a small number of fields:

- `description`
- `displayName` (read-only)
- `id` (read-only)

- `name`

Thus, you can create an underwriting authority profile. But there are currently no endpoints that let you populate the profile with authority grants. This must be done through the PolicyCenter user interface.

chapter 31

Working with organizations

In the context of the system APIs, an *organization* represents a third-party entity that supports the insurer's business, such as a producer.

With the Admin API, callers can create, update, and retrieve organization data through the `/admin/v1/organizations` endpoints. Typically, internal services associated with administrator, auditor, or underwriter roles are authorized to access the `/organizations` endpoints.

Querying for organizations

Authorized internal users can query for organizations. Callers can use the following endpoints to GET organization resources:

- GET `/admin/v1/organizations`: Returns all organizations that are visible to the caller
- GET `/admin/v1/organizations/{organizationId}`: Returns the organization with the given ID

Creating organizations

Authorized internal users can create organizations. To create an organization, callers can submit a POST request to the `/admin/v1/organizations` endpoint.

At minimum, the request body must contain the following fields:

- `name`: Organization name
- `type.code`: Organization type, from the BusinessType typelist

Optionally, the `groupDescription` field can be included (why?).

If the organization represents a producer, then the `producerStatus.code` field must also be included in the request. This field takes a value from the ProducerStatus typelist. Optionally, the `tier.code` field can also be included for producer organizations, and this field takes a value from the Tier typelist.

```
{
  "data": {
    "attributes": {
      "name": "#(args.name)",
      "producerStatus": {
        "code": "Active"
      },
      "type": {
        "code": "agency"
      }
    }
```

```
  }
}
```

Updating organizations

Authorized internal users can update organizations. To update an organization, callers can submit a PATCH request to the `/admin/v1/organizations/{organizationId}` endpoint.

chapter 32

Working with producer codes

In PolicyCenter, a *producer* refers to any third party who brings business to the insurer, such as an agent or broker. A *producer code* is an identifier used in Guidewire systems to reference a specific producer.

With the Admin API, callers can create, update, and retrieve producer code data through the `/admin/v1/producer-codes` endpoints. Typically, internal services associated with administrator or underwriter roles are authorized to access the `producer-codes` endpoints.

Querying for producer codes

Authorized internal users can query for producer codes. Callers can use the following endpoints to GET producer code resources:

- GET `/admin/v1/producer-codes`: Returns all producer codes
- GET `/admin/v1/producer-codes/{producerCodeId}`: Returns a resource for the given producer code
- GET `/admin/v1/organizations/{organizationId}/producer-codes`: Returns all producer codes associated with the given organization
- GET `/admin/v1/organizations/{organizationId}/producer-codes/{producerCodeId}`: Returns a resource for the given producer code associated with the given organization

Alternatively, callers can use the Account API to retrieve producer codes through the `/account/v1/producer-codes` endpoint. Calls to this endpoint return a collection of producer codes that are available to the caller. This endpoint is useful when creating policy submissions.

Creating producer codes

Authorized internal users can create producer codes. To create a producer code, callers can submit a POST request to the `/admin/v1/producer-codes` endpoint. At minimum, the request body must contain the following fields:

- `code`: The public code for the producer
- `organization.id`: The ID for the producer organization
- `roles`: An array of one or more roles associated with the producer. Each role is an object containing an `id` field associated with a valid value.

```
{
  "data": {
    "attributes": {
      "code": "301-008578",
      "organization": {
        "id": "pc:4"
      },
```

```
      "roles": [
        {
          "id": "producer"
        }
      ]
    }
  }
}
```

The request body can also contain the following optional fields:

- `address`: An Address resource
- `appointmentDate`: A UTC timestamp indicating appointment date
- `branch.id`: The ID for the branch office
- `description`: A string
- `parent.id`: The ID for the parent organization
- `preferredUnderwriter.id`: The ID for the preferred underwriter
- `producerStatus`: The status of the producer
- `terminationDate`: A UTC timestamp indicating termination date

Updating producer codes

Authorized internal users can update producer codes. To update a producer code, callers can submit a PATCH request to the `/admin/v1/producer-codes/{producerCodeId}` endpoint.

Exposing producer codes to external users

By default, producer codes are not exposed to external users. An *external user* is a user that is not in the PolicyCenter database, such as an anonymous user or an account holder.

In the PolicyCenter `config.xml` configuration file, the `ExternallyVisibleProducerCodes` parameter can be used to expose one or more producer codes to external users. The parameter accepts a comma-separated list of producer codes, in string format:

```
<param name="ExternallyVisibleProducerCodes" value="ProdCode1,prodCode2"/>
```

For details on PolicyCenter configuration, see the *Configuration Guide*.

part 4

Configuring the Cloud API

The system APIs in *InsuranceSuite Cloud API* have a set of default behaviors in the base configuration. However, through configuration, their behaviors can be modified.

The following topics discuss how insurers can configure system API behavior. This includes:

- Extending system API resources
- Obfuscating personally identifiable information (PII)
- Generating LOB-specific APIs

chapter 33

Extending system API resources

System API resources are defined by a set of schemas. In the base configuration, system API resources expose a subset of PolicyCenter entities and associated fields through a set of properties. To expose additional entity fields, including your own customizations, you can extend the schemas for the resource.

A resource is structured by three types of schemas. A *schema definition* defines the data structure of a resource, comprising property names and value types. A *mapper* maps the schema definition to a PolicyCenter entity and its fields. An *updater* enables resource data to be written to PolicyCenter, and is only needed for resources that must support write operations.

The basic workflow for extending a system API resource entails the following:

- Extend the schema definition
- Extend the mapper
- Extend the updater *(for POST or PATCH operations only)*

For an end-to-end walk-through, see "Tutorial: Create a resource extension" on page 241.

Schema organization

The system APIs are defined by a large collection of Swagger and JSON Schema files that are located in the `Project/configuration/config/Integration` directory of Guidewire Studio. That directory includes the following relevant subdirectories:

- `apis` contains Swagger files that define the system APIs and their associated endpoints.
- `schemas` contains schema definitions of the resource types associated with system API endpoints.
- `mappings` contains mappers that map schema definitions to PolicyCenter entities.
- `updaters` contains updaters that make resource data writeable to PolicyCenter.

To give a concrete example:

- Within `apis`, the Swagger file for the Common API includes a definition for the `/common/v1/activities/{activityId}` endpoint, the root resource of which is Activity. This endpoint supports GET and PATCH operations.
- Within `schemas`, the Activity schema definition delineates the properties and associated data types for the Activity resource. The schema definition is required to define the resource.
- Within `mappings`, the Activity mapper maps the Activity schema definition with the Activity entity in PolicyCenter. The mapper is required to enable GET operations for the resource.
- Within `updaters`, the Activity updater declares Activity resource properties that can be written to Activity entity fields in PolicyCenter. The updater is required to enable POST or PATCH operations for the resource.

Extension directories

The `apis`, `schemas`, `mappings`, and `updaters` directories each contain two subdirectories, `gw` and `ext`. The `gw` subdirectory holds the base configuration files, and users must not alter these. The `ext` subdirectory contains the extensions facilities. When extending an API, you will work with files in the `ext` subdirectories.

Note: While you must not alter the files in `gw` subdirectories, you might find it helpful to review those files to gain a deeper understanding of how the API schemas are structured.

Swagger specification syntax

All of the schema files conform to the Swagger 2 specification. For syntax details, refer to the spec at https://swagger.io/specification/v2/.

The Swagger specification files in the `schemas`, `mappings`, and `updaters` subdirectories are in JSON format, using JSON Schema syntax. For details on JSON Schema syntax, refer to the spec at http://json-schema.org/.

Additionally, you can find guidance on how Guidewire uses JSON Schema syntax in the *Plugins, Prebuilt Integrations, and SOAP APIs*. This documentation includes information on fully qualified names, which are used for file naming and include references.

Extending schema definitions

By extending a schema definition, you can add properties to a resource that are not otherwise present in the base schema definition. This process involves adding a schema definition extension to a schema extension file.

Schema extension files

In Studio, schema extension files are located in `Integration > schemas > ext > <API name>` directories, with the file naming pattern of `<API name>_ext-<VERSION>.schema.json`.

For example, the schema extension file for the Common API is located at `Integration > schemas > ext > common.v1 > common_ext-1.0.schema.json`. The base file has the following content:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "x-gw-combine": [
      "gw.content.cc.common.v1.common_content-1.0",
      "ext.framework.v1.framework_ext-1.0"
  ],
  "definitions": {}
}
```

- `$schema`: References the JSON Schema namespace declaration
- `x-gw-combine`: References an array of schema files that can be extended. These files are referenced as fully qualified names that are relative paths within the `schemas` directory.
- `definitions`: Contains the schema definition extensions. In this case, the value is an empty object, because no extensions have been created yet.

Schema definition extension syntax

A schema definition extension adheres to the following syntactic conventions:

- The extension for the target resource is defined by a JSON object contained in the `definitions` field of the schema extension file
- The name of the extension must match that of the schema definition for the target resource
- The extension must have a `properties` attribute to contain the extended properties
- Each extended property has the `_Ext` suffix appended to its name
- Each extended property contains a value type declaration

The following example shows a schema definition extension for the Activity resource in the Common API. The extension adds the `shortSubject_Ext` property to the resource, and defines the property value type as a string:

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        "shortSubject_Ext": {
          "type": "string"
        }
      }
    }
  }
}
```

- `Activity`: The name of the schema definition for the resource that is being extended
- `shortSubject_Ext`: The name of the property extension
- `type`: A declaration of the property value type

Property names

Guidewire recommends that you render property names in mixed case, with the first letter being lowercase, and that you append `_Ext` to the property name. This namespacing is important to avoid upgrade conflicts if a property with the same name is added to the core product in the future. The `_Ext` suffix is required even when adding properties from a base entity.

For example, if a PolicyCenter entity field name is `ShortSubject`, then the name of the extended resource property would be `shortSubject_Ext`.

Property value types

Each property extended by the schema definition must include a declaration of the property value type, and that type must align with the field type of the associated PolicyCenter entity. These types fall into the following general categories:

- Scalars
- Objects
- Arrays

Furthermore, property value types can be defined with one of the following attributes:

- `type`: Takes a string that indicates the property value type. Use this attribute for scalars and array value types.
- `$ref`: Takes a URI reference to a definition elsewhere in the schema. Use this attribute for object value types.

Scalars

You can declare a property value type for a scalar by adding a `type` attribute to the resource property and assigning it a JSON Schema primitive type of either `boolean`, `integer`, `number`, or `string`. If the scalar is a date or datetime type, then you would also add a `format` attribute and give it a value of either `date` or `date-time`, respectively.

Table 1: PolicyCenter scalar types with associated JSON primitive types

PolicyCenter scalar type	JSON primitive type
`bit`	`boolean`
`dateonly`	`string` in `date` format
`datetime`	`string` in `date-time` format
`decimal`	`number`
`integer`	`integer`
`longint`	`integer`
`longtext`	`string`

PolicyCenter scalar type	JSON primitive type
mediumtext	string
money	number
percentage	number
shorttext	string
text	string
varchar	string

The following example depicts base resource properties that support PolicyCenter datetime, bit, and varchar value types:

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        "closeDate": {
          "type": "string",
          "format": "date-time"
        },
        "mandatory": {
          "type": "boolean"
        },
        "shortSubject": {
          "type": "string"
        }
      }
    }
  }
}
```

Objects

You can declare a property value type for an object by assigning it a URI reference to an inline resource schema. Typically, objects are formatted through the `SimpleReference` schema. For further details on inline resources, see the *Cloud API Business Flows and Configuration Guide*.

In the following example, `assignedByUser_Ext` is a property extension whose value type is object. That value type is declared through a URI reference to the `SimpleReference` schema:

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        "assignedByUser_Ext": {
          "$ref": "#/definitions/SimpleReference"
        }
      }
    }
  }
}
```

In the schemas generally, property value types that align with PolicyCenter typekeys are formatted as objects. To declare a property value type for a typekey, assign it a URI reference to the `TypeKeyReference` schema. It is also necessary to explicitly associate the typekey with its PolicyCenter typelist through the `x-gw-extensions.typelist` attribute.

The example below shows an `assignmentStatus_Ext` property extension. Its value type is a typekey that is associated with the `AssignmentStatus` typelist:

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

```
        }
      }
    }
  }
}
```

Arrays

You can declare a property value type for an array by adding a `type` attribute to the resource property and assigning it the value of `array`. It is also necessary to indicate the value type of the array members, which can be done by adding an `items.type` attribute for scalars or a URI reference for objects.

The following property declaration is for an array of strings:

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        "exceptionSubtypes_Ext": {
          "type": "array",
          "items": {
            "type": string"
          }
        }
      }
    }
  }
}
```

The following property declaration, for the `data` property of the `RelatedCollections` resource, uses a URI reference to set the array member type as an object that conforms to the `SimpleReference` schema:

```
{
  . . .
  "definitions": {
    "RelatedCollection": {
      "properties": {
        . . .
        "data": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/SimpleReference"
          }
        }
      }
    }
  }
}
```

Virtual properties

Many PolicyCenter entity fields are virtual properties that derive their value through a Gosu method. The value returned from the method is typically dynamic, such as a concatenation of other fields, or a pointer to a specific member of an array (such as the member added most recently). In the PolicyCenter Data Dictionary, the field entry for virtual property lists the return type of the method that underlies the field. That type will be either a scalar, object, typekey, or array, as described previously. After identifying the return type, you can then follow the specific formatting guidance as described above.

Foreign keys

PolicyCenter entity fields are frequently based on foreign keys to other entities or typelists. To declare a property value type for a foreign key, you must first identify the terminating value type of the foreign key reference in the originating source. That type will be either a scalar, object, typekey, array, or virtual property, as described previously. You can then follow the specific formatting guidance for the type.

For example, the `AccountContact` entity has a foreign key to the `Contact` entity, which has a `DisplayName` property that is a string type. The chained name of this property is `AccountContact.Contact.DisplayName`. The schema definition for this property is as follows:

```
"AccountContact": {
  "type": "object",
```

```
  "x-gw-extensions": {
    "discriminatorProperty": "contactSubtype"
  },
  "properties": {
    . . .
    "displayName": {
      "type": "string",
      "readOnly": true
    },
    . . .
  }
}
```

Extending mappers

By extending a mapper, you can associate the schema definition extension of the target resource with the backing data source, a PolicyCenter entity. This step is necessary in order to expose your resource extensions to PolicyCenter through the API. To extend a mapper, you must configure a mapper extension in a mapper extension file.

Mapper extension files

In Studio, mapper extension files are located in `Integration > mappings > ext > <API name>` directories, with the file naming pattern of `<API name>_ext-<VERSION>.mapping.json` .

For example, the mapper extension file for the Common API is located at `Integration > mappings > ext > common.v1 > common_ext-1.0.mapping.json` . That file has the following content:

```
{
  "schemaName": "ext.common.v1.common_ext-1.0",`
  "combine": [
      "gw.content.cc.common.v1.common_content-1.0",
      "ext.framework.v1.framework_ext-1.0"

  ],
  "mappers": {}
}
```

- `schemaName`: References the base name of the schema extension file
- `combine`: References an array of schema files that are being extended. These files are referenced as fully qualified names that are relative paths within the `mappings` directory.
- `mappers`: Contains the mapper extensions

Mapper extension syntax

A mapper extension adheres to the following syntactic conventions:

- The extension is defined by a JSON object contained in the `mappers` field of the mapper extension file
- The name of the mapper extension matches that of the schema definition extension for the resource that is being extended
- The extension must have a `schemaDefinition` attribute that associates the mapper extension with the schema definition extension
- The extension must have a `root` attribute that associates the schema definition extension with a PolicyCenter entity
- The extension must have a `properties` attribute to contain the extended properties
- The name of each extended property must match that found in the associated schema definition extension
- Each extended property must have a `path` attribute pointing to a PolicyCenter entity field
- If the extended property value type is an object, then it must also have a `mapper` attribute that holds a relevant URI reference

The following listing shows a mapper extension for the Activity schema in the Common API. The extension maps an extended `shortSubject_Ext` resource property to the PolicyCenter `Activity.ShortSubject` entity field:

```
{
  . . .
  "mappers": {
    "Activity": {
      "schemaDefinition": "Activity",
```

```
      "root": "entity.Activity",
      "properties": {
        "shortSubject_Ext": {
          "path": "Activity.ShortSubject"
        }
      }
    }
  }
}
```

- `Activity`: The name of the mapper
- `schemaDefinition`: A mapping to the `Activity` schema definition
- `root`: A mapping of the `Activity` schema definition to the `Activity` entity in PolicyCenter
- `shortSubject_Ext`: A property name, as defined in the schema definition
- `path`: A path that associates the extended property with the `Activity.ShortSubject` entity field. Values can be chained. For example, the path for the display name of an account contact is `AccountContact.Contact.DisplayName`.

If a property value type is defined by a URI reference in the schema definition extension, then the extended property must also include a `mapper` attribute. The syntax for this value is `#/mappers/` followed by the schema name. For example, if the property value type in the schema definition extension is `"$ref": "#/definitions/SimpleReference"`, then the `mapper` attribute value would be `"mapper": "#/mappers/SimpleReference"`.

The following listing shows an extended `activityClass_Ext` resource property that maps to the PolicyCenter `Activity.ActivityClass` entity field, which is backed by a typekey. The schema definition extension declares the property value type as `"$ref": "#/definitions/TypeKeyReference"`. Therefore, it is necessary to include the `mapper` attribute:

```
{
  . . .
  "mappers": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
         "activityClass_Ext": {
           "path": "Activity.ActivityClass",
           "mapper": "#/mappers/TypeKeyReference"
        }
      }
    }
  }
}
```

Extending updaters

When extending a resource that will support POST or PATCH operations, you must also configure a matching updater extension in an updater extension file. This is necessary in order to make the extended properties writeable to PolicyCenter.

Updater extension files

In Studio, updater extension files are located in `Integration > updaters > ext > <API name>` directories, with the file naming pattern of `<API name>_ext-<VERSION>.updater.json`.

For example, the updater extension file for the Common API is located at `Integration > updaters > ext > common.v1 > common_ext-1.0.updater.json`. That file has the following content:

```
{
  "schemaName": "ext.common.v1.common_ext-1.0",
  "combine": [
    "gw.content.cc.common.v1.common_content-1.0",
    "ext.framework.v1.framework_ext-1.0"
  ],
  "updaters": { }
}
```

- `schemaName`: References the base name of the extension file

- `combine`: References an array of schema files that are being extended. These files are referenced as fully qualified names that are relative paths within the `updaters` directory.
- `updaters`: Contains the updater extensions

Updater extension syntax

An updater extension adheres to the following syntactic conventions:

- The extension is defined by a JSON object contained in the `updaters` field of the updater extension file
- The name of the updater extension matches that of the schema definition extension for the resource that is being extended
- The extension must have a `schemaDefinition` attribute that associates the updater extension with the schema definition extension
- The extension must have a `root` attribute that associates the schema definition extension with a PolicyCenter entity
- The extension must have a `properties` attribute to contain the extended properties
- The name of each extended property must match that found in the associated schema definition extension
- Each extended property must have a `path` attribute pointing to a PolicyCenter entity field
- If the extended property value type supports a typekey, then it must also have a `valueResolver.typeName` attribute that holds a `TypeKeyValueResolver` URI reference

The following listing shows an updater extension for the Activity schema in the Common API. The extension associates an extended `shortSubject_Ext` resource property with the PolicyCenter `Activity.ShortSubject` entity field:

```
{
  . . .
  "updaters": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        "shortSubject_Ext": {
          "path": "Activity.ShortSubject"
        }
      }
    }
  }
}
```

- `Activity`: The name of the updater
- `schemaDefinition`: A mapping to the `Activity` schema definition
- `root`: A mapping of the `Activity` schema definition to the `Activity` entity in PolicyCenter
- `shortSubject_Ext`: A property name, as defined in the schema definition
- `path`: A path that associates the extended property with the `Activity.ShortSubject` entity field. Values can be chained. For example, the path for the display name of an account contact is `AccountContact.Contact.DisplayName`.

If the property value type of the extended property in the schema definition extension is `TypeKeyReference`, then in the updater extension that property must include a `valueResolver` attribute that sets `typeName` to `TypeKeyValueResolver`:

```
{
  . . .
  "updaters": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        "activityClass_Ext": {
          "path": "Activity.ActivityClass",
          "valueResolver": {
            "typeName": "TypeKeyValueResolver"
          }
        }
      }
    }
```

```
  }
}
```

Tutorial: Create a resource extension

In this tutorial, you can walk through the entire process for creating a system API resource extension. Through creating the resource extension, you will create resource extensions supporting a variety of property types while executing the following tasks:

- Extend a schema definition
- Extend a mapper
- Extend an updater
- Verify the extended resource

Tools

Over the course of this exercise, you will be using Studio, Swagger UI, and Postman. This tutorial assumes that you are already familiar with working in Studio. For further information on Swagger UI, or on setting up Postman, see the *Cloud API Business Flows and Configuration Guide*.

Scenario

You have been asked to create a resource extension for the Activity resource in the Common API. That resource is based on the PolicyCenter Activity entity. You are going to extend resource properties to support the following entity fields:

Entity field name	Resource property name	Value type	Support POST and PATCH?
`ActivityClass`	`activityClass_Ext`	typekey to `ActivityClass` typelist	
`CreateTime`	`createTime_Ext`	datetime	
`CreateUser`	`createUser_Ext`	foreign key to `User` entity	
`ShortSubject`	`shortSubject_Ext`	varchar (10)	yes
`IsAutogenerated_Ext` (user-created field extension)	`isAutogenerated_Ext`	bit	yes

Notice that the last entry is a custom entity field extension. You will begin by creating that field in Studio. You will then configure the resource, mapping, and updater extensions to make these properties available through the system API. Lastly, you will verify the resource extension in Swagger UI and Postman.

Create an entity field extension

In order to make a custom entity extension accessible through a system API, that extension first. You can create a custom entity extension by doing the following:

1. In Studio, open the Activity entity for editing. You can find that file at `Project > configuration > config > Extensions > Entity > Activity.etx`.
2. Click +, and then select **column**.
3. In the new column, enter the following name and value pairs:
4. In the **name** field, enter *IsAutogenerated_Ext*
5. In the **type** field, enter *bit*
6. In the **nullok** field, enter *true*
7. Save the file.
8. To integrate your changes, start the development server in debug mode by selecing **Run > Debug 'Server'**.
9. To verify your work, regenerate the Data Dictionary for PolicyCenter, and then confirm the presence of this extension in the dictionary.

Extend a schema definition

1. In Studio, open the schema extension file associated with the Common API.

   This file is located at `Integration > schemas > ext > common > v1 > common_ext-1.0.schema.json`.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "x-gw-combine": [
    "gw.content.pc.common.v1.common_content-1.0"
  ],
  "definitions": {}
}
```

2. In the definitions field, create a schema definition extension for `Activity`, and add to this a `properties` attribute.

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {}
    }
  }
}
```

3. Within the `properties` attribute, create fields for each of the resource properties to be extended, as outlined in the "Scenario" section previously.

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        "activityClass_Ext": {},
        "createTime_Ext": {},
        "createUser_Ext": {},
        "shortSubject_Ext": {},
        "isAutogenerated_Ext": {}
      }
    }
  }
}
```

   For guidance on property naming conventions, see the "Property names" section of "Schema definition extension syntax" on page 234.

4. Within the `activityClass_Ext` property field, add a property value type for typekey.

   The typekey type is defined in the schema by a URI reference. To set the property value type, enter a `$ref` attribute and assign it a value of `#/definitions/TypeKeyReference`.

   Additionally, the typekey must be associated with a typelist. To set the typelist, add a `x-gw-extensions` attribute to the property, and then assign the appropriate typelist to the typelist field. In this example, the typelist is `ActivityClass`.

   The following code block depicts the completed property field:

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        "activityClass_Ext": {
          "$ref": "#/definitions/TypeKeyReference",
          "x-gw-extensions": {
            "typelist": "ActivityClass"
          }
        },
        . . .      }
    }
  }
}
```

5. In the `createTime_Ext` property field, add a property value type for datetime.

   To set the property value type, enter a type attribute and assign it a value of string. Next, add a format attribute and assign it a value of `date-time`.

The following code block depicts the completed property field:

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        . . .
        "createTime_Ext": {
          "type": "string",
          "format": "date-time"
        },
        . . .      }
    }
  }
}
```

6. In the `createUser_Ext` property field, add a property value type for object.

   You can declare a property value type for an object by adding a `$ref` attribute to the resource property and assigning it a URI reference for an inlined resource (for details, refer to "The attributes section" in this guide). In this instance, enter a `$ref` attribute and assign it a value of `#/definitions/SimpleReference`.

   The following code block depicts the completed property field:

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        . . .
        "createUser_Ext": {
          "$ref": "#/definitions/SimpleReference"
        },
        . . .
      }
    }
  }
}
```

7. In the `shortSubject_Ext` property field, add a property value type for string.

   To set the property value type, enter a type attribute and assign it a value of `string`.

   The following code block depicts the completed property field:

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        . . .
        "shortSubject_Ext": {
          "type": "string"
        },
        . . .
      }
    }
  }
}
```

8. In the `isAutogenerated_Ext` property field, add a property value type for bit.

   To set the property value type, enter a type attribute and assign it a value of `boolean`.

   The following code block depicts the completed property field:

```
{
  . . .
  "definitions": {
    "Activity": {
      "properties": {
        . . .
        "isAutogenerated_Ext": {
          "type": "boolean"
        }
      }
    }
  }
}
```

9. Save your changes.

Extend a mapper

1. In Studio, open the mapper extension file associated with the Common API.

   This file is located at `Integration > mappings > ext > common > v1 > common_ext-1.0.mapping.json`.

   The base file appears as follows:

```
{
      "schemaName": "ext.common.v1.common_ext-1.0",
      "combine": [
         "gw.content.pc.common.v1.common_content-1.0"
      ],
      "mappers": {}
   }
```

2. In the `mappers` field, create a mapper extension for `Activity`.

```
{
  . . .
  "mappers": {
    "Activity": {}
  }
}
```

3. In the `Activity` mapper extension, add a `schemaDefinition` property and give it the value `Activity`. This associates the mapper with the `Activity` resource.

```
{
  . . .
  "mappers": {
    "Activity": {
      "schemaDefinition": "Activity"
    }
  }
}
```

4. Add a root property, and give it the value of `entity.Activity`. This associates the `Activity` resource with the PolicyCenter `Activity` entity.

```
{
  . . .
  "mappers": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity"
    }
  }
}
```

5. Add a `properties` property, and within that create fields for each of the properties that you added previously to the schema definition extension.

```
{
  . . .
  "mappers": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        "activityClass_Ext": {},
        "createTime_Ext": {},
        "createUser_Ext": {},
        "shortSubject_Ext": {},
        "isAutogenerated_Ext": {}
      }
    }
  }
}
```

6. Configure the `activityClass_Ext` property.

   a. Add a path attribute and assign it the value `Activity.ActivityClass`.

      This maps the resource property to the `ActivityClass` entity field.

   b. Add a `mapper` attribute and assign it the value `#/mappers/TypeKeyReference`.

      Any property whose type is declared by a URI reference must have a mapping set to a URI reference for the related mappers schema.

The following code block depicts the completed property field:

```
{
  . . .
  "mappers": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        "activityClass_Ext": {
          "path": "Activity.ActivityClass",
          "mapper": "#/mappers/TypeKeyReference"
        },
        . . .
      }
    }
  }
}
```

7. In the `createTime_Ext` property, add a path attribute and assign it the value `Activity.CreateTime`.

   This maps the resource property to the `CreateTime` entity field.

   The following code block depicts the completed property field:

```
{
  . . .
  "mappers": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        . . .
        "createTime_Ext": {
          "path": "Activity.CreateTime"
        },
        . . .
      }
    }
  }
}
```

8. Configure the `createUser_Ext` property.
   a. Add a `path` attribute and assign it the value `Activity.CreateUser`.

      This maps the resource property to the `CreateUser` entity field.
   b. Add a `mapper` attribute and assign it the value `#/mappers/SimpleReference`.

      Any property whose type is declared by a URI reference must have a mapping set to a URI reference for the related mappers schema.

   The following code block depicts the completed property field:

```
{
  . . .
  "mappers": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        . . .
        "createUser_Ext": {
          "path": "Activity.CreateUser",
          "mapper": "#/mappers/SimpleReference"
        },
        . . .
      }
    }
  }
}
```

9. In the `shortSubject_Ext` property, add a path attribute and assign it the value `Activity.ShortSubject`.

   This maps the resource property to the `ShortSubject` entity field.

   The following code block depicts the completed property field:

```
{
  . . .
  "mappers": {
    "Activity": {
```

```
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        . . .
        "shortSubject_Ext": {
          "path": "Activity.ShortSubject"
        },
        . . .
      }
    }
  }
}
```

10. In the `isAutogenerated_Ext` property, add a `path` attribute and assign it the value `Activity.IsAutogenerated_Ext`.

    This maps the resource property to the `IsAutogenerated_Ext` entity field.

    The following code block depicts the completed property field:

```
{
  . . .
  "mappers": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        . . .
        "isAutogenerated_Ext": {
          "path": "Activity.IsAutogenerated_Ext"
        }
      }
    }
  }
}
```

11. Save your changes.

Extend the updater

For the updater, you only need to add resource properties that can be updated by a POST or PATCH operation. If a resource extension does not have any such properties, then it is not necessary to create an updater extension.

To create an updater that supports POST or PATCH operations for the `isAutogenerated_Ext` properties, follow these steps.

1. In Studio, open the updater extension file associated with the Common API.

   This file is located at `Integration > updaters > ext > common > v1 > common_ext-1.0.updater.json`.

   The base file appears as follows:

```
{
  "schemaName": "ext.common.v1.common_ext-1.0",
  "combine": [
    "gw.content.pc.common.v1.common_content-1.0"
  ],
  "updaters": {}
}
```

2. In the `updaters` field, create an updater extension for `Activity`.

```
{
  . . .
  "updaters": {
    "Activity": {}
  }
}
```

3. In the `Activity` updater extension, add a `schemaDefinition` property and give it the value `Activity`.

   This associates the updater with the Activity resource.

```
{
  . . .
  "updaters": {
    "Activity": {
      "schemaDefinition": "Activity"
    }
```

```
  }
}
```

4. Add a `root` property, and give it the value of `entity.Activity`.

   This associates the `Activity` resource with the PolicyCenter `Activity` entity.

```
{
  . . .
  "updaters": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity"
    }
  }
}
```

5. Add a `properties` property, and within that create a field for each of the supported properties as defined in the schema definition extension.

```
{
  . . .
  "updaters": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        "shortSubject_Ext": {},
        "isAutogenerated_Ext": {}
      }
    }
  }
}
```

6. In the `isAutogenerated_Ext` property, add a `path` attribute and assign it the value `Activity.IsAutogenerated_Ext`.

   This maps the resource property to the `IsAutogenerated_Ext` entity field, enabling property data to be written to the InsuranceSuite database.

   The following code block depicts the completed property field:

```
{
  . . .
  "updaters": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        "isAutogenerated_Ext": {
          "path": "Activity.IsAutogenerated_Ext"
        },
        . . .
      }
    }
  }
}
```

7. In the `shortSubject_Ext` property, add a `path` attribute and assign it the value `Activity.ShortSubject`.

   This maps the resource property to the `ShortSubject` entity field, enabling property data to be written to the InsuranceSuite database.

   The following code block depicts the completed property field:

```
{
  . . .
  "updaters": {
    "Activity": {
      "schemaDefinition": "Activity",
      "root": "entity.Activity",
      "properties": {
        . . .
        "shortSubject_Ext": {
          "path": "Activity.ShortSubject"
        }
      }
    }
  }
}
```

8. Save your changes.

Verify the extended resource

After creating the schema definition extension, you can review the revised schema definition in Swagger UI.

1. Launch Swagger UI, and then load the Common API.

   For details, see the *Cloud API Business Flows and Configuration Guide*.
2. Select the GET `/common/v1/activities/{activityId}` endpoint.
3. Under **Responses**, select the **Model** view.
4. In the `Activity` schema associated with the `data.attributes` section, verify the presence of the extended properties.

Additionally, you can test drive the revised schema definition using Postman and some sample data. This tutorial assumes you have set up your environment with Postman and the correct sample data set. For more information, see the *Cloud API Business Flows and Configuration Guide*.

First, you can review the response object of a GET operation for the resource property extensions.

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify **Basic Auth** authorization using user `aapplegate` and password `gw`.
3. Enter the following call, and then click **Send**:

   GET `http://localhost:8180/pc/rest/common/v1/activites/pc:105`
4. Review the body of the response. It appears as follows:


```
{
    "data": {
            "attributes": {
            "activityClass_Ext": {
                "code": "task",
                "name": "Task"
            },
            "activityPattern": "premium_report_overdue",
            "activityType": {
                "code": "general",
                "name": "General"
            },
            "assignedByUser": {
                "displayName": "System User",
                "id": "systemTables:2"
            },
            "assignedGroup": {
                "displayName": "Los Angeles Branch UW",
                "id": "pc:7"
            },
            "assignedUser": {
                "displayName": "Alice Applegate",
                "id": "pc:105"
            },
            "assignmentStatus": {
                "code": "assigned",
                "name": "Assigned"
            },
            "createTime_Ext": "2020-04-23T15:28:26.677Z",
            "createUser_Ext": {
                "displayName": "System User",
                "id": "systemTables:2"
            },
            "description": "Premium report overdue",
            "dueDate": "2020-04-27T15:28:26.682Z",
            "escalated": true,
            "escalationDate": "2020-04-27T15:28:26.682Z",
            "externallyOwned": false,
            "id": "pc:105",
            "mandatory": true,
            "priority": {
                "code": "high",
                "name": "High"
            },
            "recurring": false,
            "status": {
                "code": "open",
                "name": "Open"
            },
            "subject": "Premium report overdue"
        },
        . . .
```

```
    }
}
```

You can test the updater by executing a PATCH operation on the same resource:

1. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
2. Specify **Basic Auth** authorization using user `aapplegate` and password `gw`.
3. Enter the following call, but do not click **Send**:

   PATCH `http://localhost:8180/pc/rest/common/v1/activites/pc:105`
4. 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": {
            "isAutogenerated_Ext": true,
            "shortSubject_Ext": "shortsub"
        }
    }
}
```

5. Click **Send**. The response payload appears below the request payload.

```
{
    "data": {
        "attributes": {
            "activityClass_Ext": {
                "code": "task",
                "name": "Task"
            },
            "activityPattern": "premium_report_overdue",
            "activityType": {
                "code": "general",
                "name": "General"
            },
            "assignedByUser": {
                "displayName": "System User",
                "id": "systemTables:2"
            },
            "assignedGroup": {
                "displayName": "Los Angeles Branch UW",
                "id": "pc:7"
            },
            "assignedUser": {
                "displayName": "Alice Applegate",
                "id": "pc:105"
            },
            "assignmentStatus": {
                "code": "assigned",
                "name": "Assigned"
            },
            "createTime_Ext": "2020-04-23T15:28:26.677Z",
            "createUser_Ext": {
                "displayName": "System User",
                "id": "systemTables:2"
            },
            "description": "Premium report overdue",
            "dueDate": "2020-04-27T15:28:26.682Z",
            "escalated": true,
            "escalationDate": "2020-04-27T15:28:26.682Z",
            "externallyOwned": false,
            "id": "pc:105",
            "isAutogenerated_Ext": true,
            "mandatory": true,
            "priority": {
                "code": "high",
                "name": "High"
            },
            "recurring": false,
            "shortSubject_Ext": "shortsub",
            "status": {
                "code": "open",
                "name": "Open"
            },
            "subject": "Premium report overdue"
        },
        . . .
```

```
    }
}
```

Providing feedback

The system APIs expose a subset of PolicyCenter base entities and associated fields. If there are entities or fields that you think should be added to system API resources, let your Guidewire representative know. The system APIs are in active development, and your feedback will be helpful to the system API development team.

chapter 34

Obfuscating Personally Identifiable Information (PII)

Generally, enterprises that handle personal data must abide by the 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.

One way to protect the privacy of individuals is to obfuscate Personally Identifiable Information (PII). This approach limits the exposure of designated PII, and is supported by the system APIs. PII can be obfuscated by either nullifying or masking. PII is nullified when its value is returned null. PII is masked when a portion of its value is returned with placeholder characters, such as 'XXXXXXX-3213' as a return value for an account number.

Nullifying PII

You can nullify the return value of PII by modifying the mapper for the relevant resource property. This can be done in a resource extension. For details on resource extensions, see “Extending system API resources” on page 233.

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 PII

You can mask the return value of PII 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*. The mapper can be modified through a resource extension. For details on extending resources, see “Extending system API resources” on page 233.

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 "Extending system API resources" on page 233. For details on allowlisting fields, see the section on API role files in *Cloud API Authentication Guide*.

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.

---

chapter 35

APIs for lines of business

Every product in PolicyCenter consists of one or more lines of business (LOBs). From a technical standpoint, every line of business is implemented through a set of LOB-specific artifacts. This can include the following:

- Database tables that store LOB-specific information
- LOB-specific PCFs
- LOB-specific reference tables, such as tables that store class codes

When a line of business is exposed to the system APIs, there is an additional set of LOB-specific artifacts: the LOB-specific APIs. An *LOB-specific API* is a set of endpoints that can create, read, update, or delete information for a specific line of business. Every LOB-specific API includes endpoints to work with `Line` resources. Depending on the structure of the line of business, the API may include other endpoints, such as:

- Endpoints to work with coverables
- Endpoints to work with locations

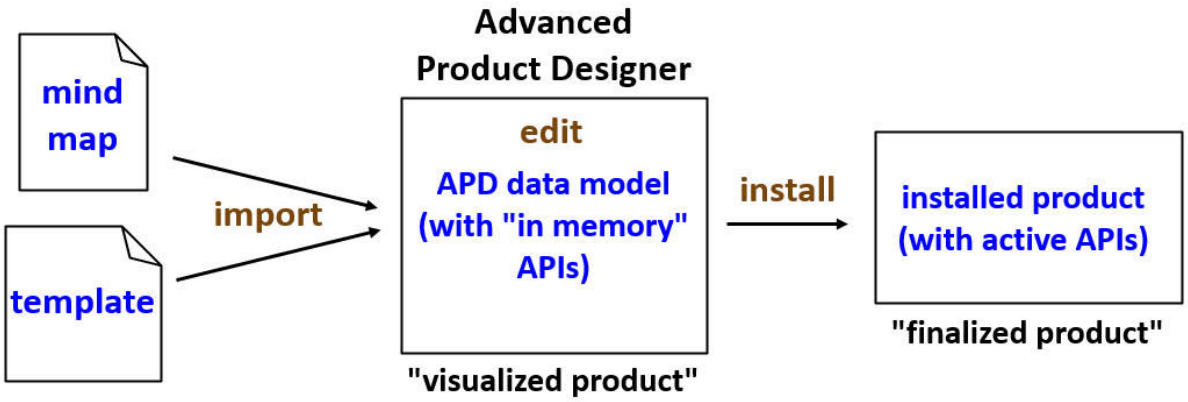
Every insurer develops lines of business for their specific needs. This can involve modifying base configuration LOBs or creating new ones. Because of this, the base configuration does not include any LOB-specific APIs. Insurers must generate the APIs for themselves after the structure of the line of business has been sufficiently developed. PolicyCenter provides functionality that lets you generate APIs for the LOBs that support API generation.

Note:

Line of business configuration can vary depending on the LOB. For further assistance, contact Guidewire Support.

Generating and installing LOB-specific APIs

The following diagram provides a high-level overview for how a line of business is typically developed using Advanced Product Designer.



1. The product and its lines of business start as metadata that is captured in either a mind map or a template.
2. The insurer creates a *visualized product* by **importing** the LOB metadata into PolicyCenter.
   a. During the import, PolicyCenter generates a set of "in memory" APIs that reflect the structure of the LOB.
3. While in Advanced Product Designer, the insurer typically **edits** the product.
   a. This is typically an iterative process where the insurer refines the metadata as needed.
   b. Every time the product is modified, the "in memory" APIs are regenerated.
4. Once the refining is complete, the insurer creates a *finalized product* by **installing** the product. The finalized products consists of:
   a. The necessary artifacts for the product, including database tables and PCFs.
   b. A set of "active" APIs that are now part of the system APIs.

Note: When importing, editing, or installing a line of business through Advanced Product Designer, there is no separate step to generate LOB-specific APIs. The APIs are generated automatically whenever the product is imported, edited, or installed.

For more information on Advanced Product Designer, refer to the *Advanced Product Designer Guide*.

Related developer tasks

When working with LOB-specific APIs, developers can do the following:

- Generate LOB-specific APIs as part of an entire product (through Advanced Product Designer)
- Generate a template from an existing finalized product
- Install LOB-specific APIs without modifying other product-specific artifacts

Generating LOB-specific APIs through APD

When you import, edit, or install a product through Advanced Product Designer, LOB-specific APIs are automatically generated. For more information on how to use Advanced Product Designer, refer to the *Advanced Product Designer Guide*.

Note that the generation of APIs through Advanced Product Designer can be either seamless or bootstrapped.

- If the line of business supports seamless generation, then the LOB-specific APIs are generated automatically from the visualized product. No manual modifications are required.
- If the line of business supports only bootstrapped generation, then some manual modification of the generated APIs is required.

Generate LOB-specific APIs

About this task

You can generate an LOB-specific API from an APD template. This process applies to both creating new LOB-specific APIs as well as updating existing APIs.

Note:

PolicyCenter upgrades include new features for use with the Cloud APIs. In order to apply these new features to your LOB-specific APIs, you must update each of those APIs after upgrading PolicyCenter itself.

To execute this task, you must have an APD template for the LOB. If necessary, you can generate an APD template from the existing visualized product and import the template. For more information, see "Generate a template using the reverse template generator" on page 257

Procedure

1. In PolicyCenter, set **Options** > **Preferences** > **Product Design Mode** to **Developer**, and then click **Update**.
2. To add the template, click **Administration** > **Product Management** > **Import From Template**, browse and select your template, and then click **Update**.
3. To generate the API, from the **Details** pane click **Edit Product** > **Generate Product Code** > **System APIs** > **System APIs - Code**
4. Review the model, and if acceptable, click **Complete Generation**.
5. Click **Return to Product Definition**, and then click **Save**.
6. Restart PolicyCenter.

Results

To verify the result, browse to the **APD Managed** pane on the **Product Management** page. The LOB product will be displayed, and the **Last Updated** column will have a value, indicating that the product has been installed. For details on viewing the API in Swagger, see "View an API definition using Swagger UI" on page 25.

Generating templates from a finalized product

Products can be created in PolicyCenter using approaches other than Advanced Product Designer. These approaches create a product with several LOB-specific artifacts (such as LOB-specific database tables or PCFs). But, they do not create any LOB-specific APIs. If you want to expose these lines of business to the system APIs, you need to generate LOB-specific APIs.

You can achieve this by doing the following:

1. Create a template for the product using the reverse template generator.
2. Import the product into Advanced Product Designer.
3. From the imported product, install only the LOB-specific APIs.

The *reverse template generator* is a script that creates an XML template from an installed product. Because the generator generates XML based on the installed product, the resulting template typically has more information in it than what exists in a mind map or in a product that is only visualized.

Advanced Product Designer is not required to run the reverse template generator. Therefore, it is possible to run the reverse template generator in older versions of PolicyCenter to generate a template. However, Advanced Product Designer is required to import the template and install the corresponding APIs.

Generate a template using the reverse template generator

Procedure

1. In PolicyCenter, select **Administration** > **Product Management** > **Externally Managed**. This pane contains a list of products that have been installed by means other than APD.
2. In the **Installed Products** pane, select the product for which you wish to generate an APD template.
3. In the **Details** pane, click **Extract APD Representation**. PolicyCenter generates the APD template and stores it in the `<USER_HOME>/Downloads` directory.

Results

Once the product is imported as a visualized product and any inconsistencies have been corrected, you can install just the APIs. For details, see “Generate LOB-specific APIs” on page 256.

Installing LOB-specific APIs without installing other artifacts

In some situations, you may have an installed product that has no LOB-specific APIs. For example, this can occur when a product has been created outside of Advanced Product Designer, such as:

- A base configuration product
- A product installed from a Standards Based Template (SBT)
- A product created in Product Designer

You can create a template from the installed product using the reverse template generator, and then import the template into Advanced Product Designer. Once the product has been imported, you can install only the APIs, without modifying any of the other existing product-specific artifacts.

API codegen configuration

APD-generated products adhere to certain naming conventions and behaviors that simplify API generation. However, when generating LOB-specific APIs from an APD template that has been exported from a non-APD product, it is possible for naming conflicts to arise during code generation that cause build failure, since the existing product may have irregular entity names or other functionality not yet supported by APD.

In such a circumstance, you can apply an API codegen configuration file for the non-APD product. The codegen configuration can be used to map any mismatches of type or field names between the non-APD product with those used in the associated APD template. Additionally, it is possible to override certain behaviors, such as constraints, that exist in the non-APD product but are not reflected in its APD template.

Note:

API codegen configuration can become quite complex, depending on the LOB. For assistance, contact Guidewire Support.

Applying API codegen overrides

In order to apply API codegen overrides to a legacy product, the following criteria must be met:

- The directory **config** > **Integration** > **apis** > **installedlobs** must be present in your system. If it is not present, then you must create it.
- That directory must contain the API codegen configuration file, in YAML format.
- The file name must be in the form `<xx>_codegen_config.yaml`, in which `<xx>` is the product suffix.

After meeting the above conditions, you must regenerate the product API.

API codegen configuration syntax

Resolving mismatched type/field names

Sometimes, there can be a mismatch between APD-generated types and the fields in the corresponding product, particularly when the template has been exported from a non-APD product. The API codegen configuration file can be used to specify mapping information that resolve these mismatches.

The API codegen configuration file must have a top-level `types` property followed by one or more APD-generated types. Each APD-generated type can specify a `nameOverride` property, which identifies the relevant type name in the product. Product values are case-insensitive, but APD-generated values are case-sensitive and the case must match. The syntax for this is as follows:

```
types:
  <APDGeneratedName>:
    nameOverride: <TypeNameInProduct>
```

For example, suppose you have an APD-generated type `PALine`, which corresponds to the product's `PersonalAutoLine`. The following code maps these values.

```
types:
  PALine:
    nameOverride: PersonalAutoLine
```

For each type, you can also provide type-to-field mapping by specifying a `fields` property. The `fields` property includes an array of one or more APD-generated types to override. Each type name contains a `nameOverride` property that identifies the field name as used in the product. As is the case with the parent type value, APD-generated field-level values must match the case of the APD-generated type values. The syntax for this is as follows:

```
types:
  <APDGeneratedName>:
    nameOverride: <TypeNameInProduct>
    fields:
      <APDGeneratedName>:
        nameOverride: <FieldNameInProduct>
```

For example, suppose you have an APD-generated type `PALine`, which corresponds to the product's `PersonalAutoLine`. There is also an APD-generated `Year` type which corresponds to the product's `ModelYear` field. The following code identifies how to map these values.

```
types:
  PALine:
    nameOverride: PersonalAutoLine
    fields:
      Year:
        nameOverride: ModelYear
```

Modifying field constraints

In addition to applying naming overrides, it is also possible to apply or relax type or field constraints enforced in the APD template that differ from the behavior expected in the originating product.

The following optional properties can be used to override specific aspects of a type or field:

- `autonumber`: Accepts resource name of the schema and mapping to be used in lieu of the default sorting
- `canDelete`: Enable or disable DELETE method (Boolean)
- `canPatch`: Enable or disable PATCH method (Boolean)
- `canPost`: Enable or disable POST method (Boolean)
- `canSplit`: Enable or disable the split custom action (Boolean)
- `createOnly`: Enable or disable modifiers (Boolean)
- `identifier`: Override default identifying property
- `ignored`: Hide or expose the field (Boolean)
- `nullable:`: Allow a null value (Boolean)
- `oneToOne`: Enforce a one-to-one relationship by removing the ability to add or remove values
- `requiredForCreate`: Override the APD-derived `requiredForCreate` value (Boolean)
- `resourceName`: Override default schema and resource name. Apply this value when the resource name differs from the `nameOverride` name.
- `toCreateAndAdd`: Override default `createAndAdd` method
- `toRemove`: Override default `remove` method
- `wizardStepIds`: Add or drop wizard step identifiers (Boolean)

The following code block is an example of an API codegen configuration for a Personal Auto line that includes optional properties:

```
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
```

Disabling product artifacts during testing

Multiple sets of product artifacts

During the development of an APD product, the product moves from a *visualized product* to a *finalized product*. However, product development does not always move in a single direction. Rather, it is typically an iterative process. Testing of the finalized product may uncover issues that require fixes to the visualized product. Once the visualized product has been fixed, it must be reinstalled as a finalized product and retested.

Thus, during product testing, it is not unusual to have two sets of product artifacts: one for the visualized product and one for the finalized product. Furthermore, these two sets of product artifacts are not necessarily in sync at all times.

Disabling product artifacts

The LOB-specific APIs use the product artifacts as the backing data source for the APIs. However, the LOB-specific APIs can only access one set of product artifacts at a time. Because there might be multiple sets of product artifacts, the system APIs give you the ability to disable either set of product artifacts. Disabling a set of product artifacts can also be useful when you have inadvertently created a line with internal errors that are significant enough that it could prevent PolicyCenter from running.

Every product has an **Enable for REST API** flag. When this flag is set to *Disabled*, the system APIs will not use the visualized product artifacts. This field is visible in Advanced Product Designer on the **Product Definition** screen only when the user's **Product Design Mode** preference is set to *Developer*.

Every installed product has a product file in the `integration/apis/installedlobs` directory. This is a YAML file whose name starts with the product's **Abbreviation** as defined on the **Product Definition** screen. If this file is renamed or removed, the system APIs will not use the finalized product artifacts.

Disabling product artifacts typically occurs in development environments only. This is because it is unlikely that a visualized product (or a product with significant errors) would be brought into a production environment.

Determining which product artifacts to use

If there is only a visualized set of product artifacts, the system APIs will use these artifacts. (However, if the visualized artifacts have been disabled, the system APIs will throw an error.)

If there is only a finalized set of product artifacts, the system APIs will use these artifacts. (However, if the finalized artifacts have been disabled, the system APIs will throw an error.)

If there is both a visualized set of product artifacts and a finalized set of product artifacts, the system APIs check the product's **Enable for REST API** flag.

- If the flag is set to *Enabled*, the visualized set is used.
- If the flag is set to *Disabled* (and the product file is in the `installedlobs` directory), the finalized set is used.

Disable a product's visualized artifacts

About this task

Disabling a visualized product is useful when you want to test the finalized product. It can also be useful when you have inadvertently created a line with internal errors that are significant enough that it could prevent PolicyCenter from running.

Procedure

1. Set your **Product Design Mode** preference to *Developer*. (You can do this by selecting **Preferences** from the **Options** menu. The **Options** menu is represented by the gear icon in the upper right corner.)
2. Navigate to the product's **Product Definition** screen.
3. Set **Enabled for REST API** to *Disabled*.
4. Restart PolicyCenter.

Results

PolicyCenter immediately disables the product's visualized artifacts.

Enable a product's visualized artifacts

About this task

Enabling a visualized product is useful when you have previously disabled the visualized product (for example, when testing the finalized product) and you now want to re-enable the visualized product (for example, to conduct further testing on the visualized product).

Procedure

1. Set your **Product Design Mode** preference to *Developer*. (You can do this by selecting **Preferences** from the **Options** menu. The **Options** menu is represented by the gear icon in the upper right corner.)
2. Navigate to the product's **Product Definition** screen.
3. Set **Enabled for REST API** to *Enabled*.
4. Restart PolicyCenter.

Results

PolicyCenter immediately enables the product's visualized artifacts.

Disable a product's installed artifacts

About this task

Disabling a finalized product can also be useful when you have inadvertently created a line with internal errors that are significant enough that it could prevent PolicyCenter from running.

Procedure

1. In Windows Explorer, navigate to the `integration/apis/installedlob` directory.
2. Either rename or delete the product file corresponding to the installed product.
3. Restart PolicyCenter.

Results

PolicyCenter immediately disables the product's installed artifacts.

Managing LOB-specific APIs for testing and integration

The Product Definition API provides endpoints that can be used to import, visualize, and generate code for products from Advanced Product Designer product templates. These APIs are designed to assist you in managing your product testing and integration. For building product models, Guidewire recommends using Advanced Product Designer directly.

Importing a product template through the API

The Product Definition API provides a set of endpoints that can be used to import a product from an Advanced Product Designer (APD) product template.

Endpoint	Template type	Input file format
`/productdefinition/v1/import-template`	An APD product template	XML
`/productdefinition/v1/import-edition`	An APD product template, imported as a product edition	XML
`/productdefinition/v1/import-xmind`	An XMind mind map that can be used to generate an APD product template	XMIND
`/productdefinition/v1/import-json`	An APD product template	JSON

In the request body, `Content-Type` must be set to `multipart/form-data`, and the product template file must be passed through the `content` parameter.

The following example shows a partial request header for a call that imports a `BPP-PetBusinessOwners-template.xml` file:

```
> POST /pc/rest/productdefinition/v1/import-template HTTP/1.1
> Accept-Encoding: gzip,deflate
> Authorization: Basic c3U6Z3c=
> Connection: Keep-Alive
> Content-Type: multipart/form-data; boundary=--------iSl9mNZkwq31beJX5v
> Transfer-Encoding: chunked

> --------iSl9mNZkwq31beJX5v
> Content-Type: text/xml
> Content-Disposition: form-data; content="BPP-PetBusinessOwners-template.xml"

   (content of BPP-PetBusinessOwners-template.xml)
 > --------iSl9mNZkwq31beJX5v--
```

A successful request returns the following response body:

```
{
    "data": {
        "attributes": {
            "id": "Petbusinessowners"
        }
    }
}
```

By default, the imported product template is loaded in PolicyCenter as a visualized product, and its LOB-specific APIs are enabled for inspection.

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 as described above, 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": "Petbusinessowners-2"
    }
  }
}
```

Querying for product templates

Authorized internal users can query for product templates. Authorized callers can use the following endpoints to retrieve product template resources:

- `/productdefinition/v1/product-templates`
- `/productdefinition/v1/product-templates/{productId}`

For example, a ProductTemplate resource for a Pet Business Owners product appears as follows:

```
{
    "data": {
        "attributes": {
            "abbreviation": "PBP",
            "codeIdentifier": "PetBusinessOwners",
            "description": "Pet Business Owners",
            "enabled": true,
            "id": "Petbusinessowners",
            "name": "Petbusinessowners",
            "productAccountType": {
                "code": "Any",
                "name": "Any"
            }
        },
        . . .
        "links": {
            "codegen": {
                "href": "/productdefinition/v1/product-templates/Petbusinessowners/codegen",
                "methods": [
                    "post"
                ]
            },
            "disable": {
                "href": "/productdefinition/v1/product-templates/Petbusinessowners/disable",
                "methods": [
                    "post"
                ]
            },
            "self": {
                "href": "/productdefinition/v1/product-templates/Petbusinessowners",
                "methods": [
                    "delete",
                    "get"
                ]
            }
        }
    }
}
```

Toggling the active state of a visualized product

By default, importing a product template creates a product in visualized mode and enables its API for queries. The Product Definition API provides endpoints for toggling this state:

- `/productdefinition/v1/product-templates/{productId}/enable`
- `/productdefinition/v1/product-templates/{productId}/disable`

The request body for either call is the same. It contains a `productId` property whose value is the product identifier.

```
{
    "data": {
        "attributes": {
            "productId": "Petbusinessowners"
        }
    }
}
```

The response body contains the associated ProductTemplate resource. The `enabled` property displays a Boolean value indicating whether the product is enabled or disabled.

```
{
    "data": {
```

```
        "attributes": {
            "abbreviation": "PBP",
            "codeIdentifier": "PetBusinessOwners",
            "description": "Pet Business Owners",
            "enabled": false,
            "id": "Petbusinessowners",
            "name": "Petbusinessowners",
            . . .
        },
        . . .
    }
}
```

Generating code from a visualized product

In PolicyCenter, authorized users can generate several types of code from a visualized product.

Code type	Description	Installed location in PolicyCenter

Base product code	All code necessary to enable complete product functionality in PolicyCenter	`/pc/app-pc/pc-apd-genlob-content/config/extensions/`
		`/pc/app-pc/pc-apd-genlob-content/config/locale/`
		`/pc/app-pc/pc-apd-genlob-content/config/lookuptables/`
		`/pc/app-pc/pc-apd-genlob-content/config/resources/`
		`/pc/app-pc/pc-apd-genlob-content/config/web/`
		`/pc/app-pc/pc-apd-genlob-content/displaynames/`
		`/pc/app-pc/pc-apd-genlob-content/gsrc/`

API code	API-related code (Swagger, schema, mapper, updater, and resource files)	`/pc/app-pc/pc-lob-{lobId}`
API test code	API test stubs, using the Karate testing framework *(these tests belong to the customer to maintain and are not considered product)*	`/pc/apitestbench/customer-api/src/test/java/gw/pc/customer/api/lob/{lobId}`
External	A function stub that can be used to hook in code extensions	`/pc/app-pc/pc-apd-genlob-content/config/extensions/`

(`{lobId}` refers to the three-character abbreviation used in LOB names, such as "BOP")

To generate code from a visualized product, authorized callers can submit a POST request to the `/productdefinition/v1/product-templates/{productId}/codegen` endpoint.

The request body can include the `generationMode` property, set to one of the following values:

- `ALL`: Generate API and product base code
- `API_CODE`: Generate code for the API only
- `BASE_CODE`: Generate product base code only
- `EXTERNAL`: Generate extensions code only

By default, this value is set to ALL.

The following code block displays a request body for generating the API code for a Pet Business Owners product:

```
{
    "data": {
        "attributes": {
            "productId": "Petbusinessowners"
            "generationMode": "API_CODE"
        }
    }
}
```

Deleting a product template

To delete a product template, authorized callers can submit a business action POST to the `/productdefinition/v1/product-templates/{productId}/delete` endpoint.