Guidewire ClaimCenter™ for Guidewire Cloud

Cloud API Business Flows Guide

Release: 2021.04

GUIDEWIRE

Product Name: Guidewire ClaimCenter for Guidewire Cloud
Product Release: 2021.04
Document Name: Cloud API Business Flows Guide
Document Revision: 18-March-2023

Contents

Part 2
ClaimCenter business flows

Part 3
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.

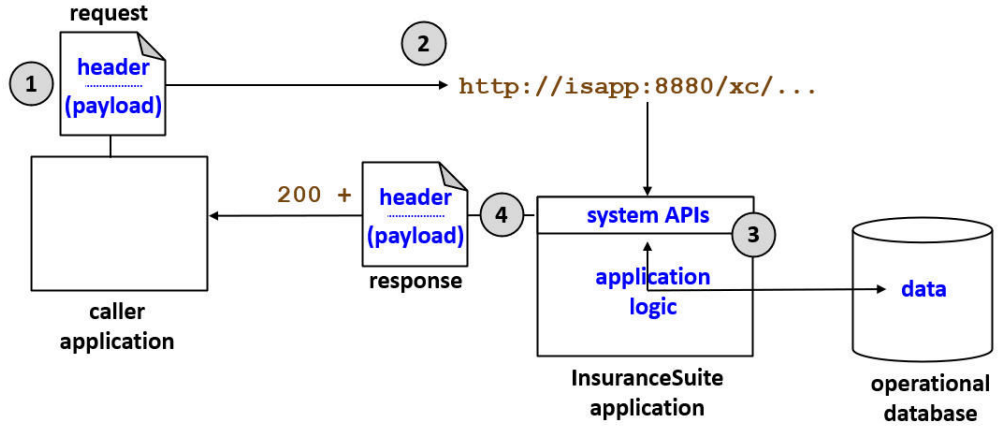
The InsuranceSuite Cloud API

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 ClaimCenter 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 `ClaimContact` resource maps to three data model entities in ClaimCenter: `ClaimContact`, `Contact`, and `ClaimContactRole`.

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 ClaimCenter. For example, the `/common/v1/activities` endpoint can be used to either request data about ClaimCenter activities or trigger actions related to ClaimCenter 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 `/fixed-property-incidents` endpoint is `FixedPropertyIncident`)
  - In some cases, the endpoint name may differ from the root resource name (for example, the root resource for the `/contacts` endpoint is `ClaimContact`)

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.

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 ClaimCenter installed on your machine that contains only the Demo sample data set.
- An instance of Postman.

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

Note: If your instance of ClaimCenter 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 Demo 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 ClaimCenter and load the Demo 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:8080/cc/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": 25,
    "data": [
        {
        "attributes": {
            "activityPattern": "contact_claimant",
            "assignedGroup": {
                "displayName": "Auto1 - TeamA",
                "id": "demo_sample:31"
            },
            "assignedUser": {
                "displayName": "Andy Applegate",
                "id": "demo_sample:1"
            },
```

Troubleshooting: No response

Requests can be sent only to running applications. All of the tutorials in this documentation require that ClaimCenter 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:8080/cc/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
Claim	API for claims and claim-specific objects	`/claim/v1`
Admin	API for administration objects	`/admin/v1`
Testsupport	API for testing during development (Available only when ClaimCenter is started in the `ci-test` environment)	`/testsupport/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 Cloud API information

In order to write system API calls, developers need detailed information about the APIs. This includes:

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

This information is often referred to as the *API metadata*. The metadata is defined in a series of swagger files. These are the common approaches for viewing the metadata:

- Swagger UI

- Calling either the `/openapi.json` endpoint or the `/swagger.json` directly through a request tool, such as Postman

Swagger UI

There is an open source tool named *Swagger UI* that automatically presents this information as an interactive web page. For information on Swagger UI, refer to the Swagger web site: https://swagger.io/tools/swagger-ui/

Swagger UI is automatically bundled with InsuranceSuite applications that have system APIs.

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 API and observe responses. However, Guidewire recommends using Swagger UI only to view system API documentation. The system APIs 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.

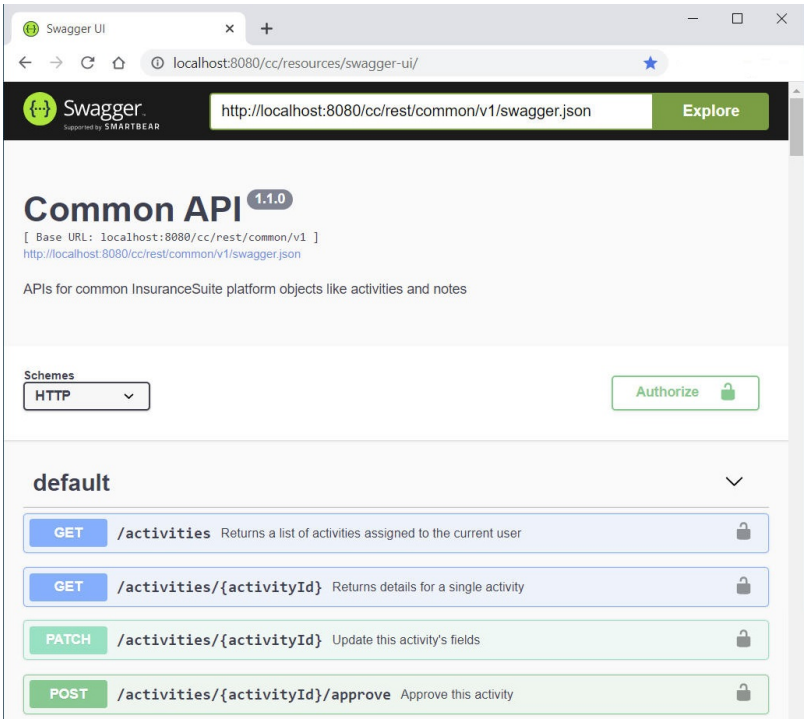
View a system API using Swagger UI

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 ClaimCenter.
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 ClaimCenter, use: `http://localhost:8080/cc/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 metadata endpoints and Postman

In some situations, it is useful to view raw data about the endpoints of an API. Every system API includes two endpoints that return metadata about the API: `/openapi.json` and `/swagger.json`.

- `/openapi.json` returns metadata information using the OpenAPI 3.0 specification, often referred to as "OpenAPI 3.0"
- `/swagger.json` returns metadata information 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, metadata about 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 metadata endpoints can be helpful when you want to view all metadata about an endpoint, including metadata that Swagger UI might strip out. However, the metadata 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 metadata endpoints are available to any caller, including unauthenticated callers.

Postman

You can call the metadata 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 a system API 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 ClaimCenter.
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 ClaimCenter, enter the following:
   - http://localhost:8080/cc/rest/common/v1/openapi.json (OR http://localhost:8080/cc/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 metadata endpoint output

The output of the metadata 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 API 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 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

Beta functionality for ClaimCenter

In this release, the following endpoints in published ClaimCenter APIs are beta:

- Endpoints that are enabled through the `feature.ClaimFinancialsApisBeta` toggle:
  - POST `/claims/{claimId}/check-sets`
  - GET `/claims/{claimId}/payments`
  - GET `/claims/{claimId}/payments/{transactionId}`
  - GET `/claims/{claimId}/payments/{transactionId}/approvals`
  - POST `/claims/{claimId}/reserve-sets`
  - GET `/claims/{claimId}/reserves`
  - GET `/claims/{claimId}/reserves/{transactionId}`
  - GET `/claims/{claimId}/reserves/{transactionId}/approvals`
  - GET `/claims/{claimId}/reserves/{transactionId}/group-reserves`
- Endpoints that are enabled through the `feature.ClaimServicingApisBeta` toggle:

- POST `/claims/{claimId}/service-requests/{serviceRequestId}/add-quote`
- POST `/claims/{claimId}/service-requests/{serviceRequestId}/completion-date`
- GET `/claims/{claimId}/service-requests/{serviceRequestId}/history`
- GET `/claims/{claimId}/service-requests/{serviceRequestId}/history/{serviceRequestChangeId}`
- GET `/claims/{claimId}/service-requests/{serviceRequestId}/instructions`
- POST `/claims/{claimId}/service-requests/{serviceRequestId}/instructions`
- GET `/claims/{claimId}/service-requests/{serviceRequestId}/instructions/{instructionId}`
- POST `/claims/{claimId}/service-requests/{serviceRequestId}/invoices/{invoiceId}/approve`
- `POST/claims/{claimId}/service-requests/{serviceRequestId}/invoices/{invoiceId}/pay`
- POST `/claims/{claimId}/service-requests/{serviceRequestId}/invoices/{invoiceId}/reject`
- GET `/claims/{claimId}/service-requests/{serviceRequestId}/messages`
- POST `/claims/{claimId}/service-requests/{serviceRequestId}/messages`
- GET `/claims/{claimId}/service-requests/{serviceRequestId}/messages/{messageId}`
- POST `/claims/{claimId}/service-requests/{serviceRequestId}/quote-date`
- GET `/claims/{claimId}/service-requests/{serviceRequestId}/quotes`
- GET `/claims/{claimId}/service-requests/{serviceRequestId}/quotes/{quoteId}`
- PATCH `/claims/{claimId}/service-requests/{serviceRequestId}/quotes/{quoteId}`
- POST `/claims/{claimId}/service-requests/{serviceRequestId}/quotes/{quoteId}/approve`
- POST `/claims/{claimId}/service-requests/{serviceRequestId}/resume`
- POST `/claims/{claimId}/service-requests/{serviceRequestId}/suspend`
- GET `/service-messages`

This release contains no ClaimCenter APIs that are entirely beta.

Beta functionality for ContactManager

This release contains no ContactManager APIs that are entirely beta.

This release contains no beta endpoints in any of the published ContactManager APIs.

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:8080/cc/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:8080/cc/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 ClaimCenter.
- `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 ClaimCenter. 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 `ClaimantType` typelist. You will then use a typekey filter to query for all claimant types that are related to a claim loss type of `PR` (which means the claim's policy is a property policy).

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:8080/cc/rest/common/v1/typelists/ClaimantType`
4. The response payload contains all non-retired claimant types. Verify that the first three codes in the payload are: `insured`, `householdmember`, `veh_ins_driver`.)
5. Modify the call by adding the following query parameter to the end, and then click **Send**:
   - `?typekeyFilter=category:cn:LossType.PR`
6. The response payload now contains only claimant types relevant to property claims. Verify that the first three codes in the payload are now: `insured`, `householdmember`, `propertyowner`. (`veh_ins_driver` no longer appears because it is not a valid claimant type for a property claim.)

Routing related API calls in clustered environments

To improve performance and reliability, you can install multiple ClaimCenter servers in a configuration known as a cluster. A ClaimCenter cluster distributes client connections among multiple ClaimCenter 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 ClaimCenter 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 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 ClaimCenter 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.

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, 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 ClaimCenter 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.

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.

However, Guidewire supports both approaches.

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 ClaimCenter.
2. In a web browser, navigate to the Swagger UI for the appropriate API.
   - For more information, see “View a system API 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 ClaimCenter 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 ClaimCenter on your machine, enter: `http://localhost:8080/cc/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:8080/cc/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": 25,
        "data": [
            {
            "attributes": {
            "activityPattern": "contact_claimant",
            "assignedGroup": {
                "displayName": "Auto1 - TeamA",
                "id": "demo_sample:31"
            },
            "assignedUser": {
                "displayName": "Andy Applegate",
                "id": "demo_sample:1"
            },
```

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 ClaimCenter is installed on your local machine.
- The Demo 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 "Contact claimant" (whose Public ID is cc:20)
   - GET `http://localhost:8080/cc/rest/common/v1/activities/cc:20`
2. Claim 235-53-365870 (whose Public ID is demo_sample:1)
   - GET `http://localhost:8080/cc/rest/claim/v1/claims/demo_sample:1`

Response payload for a collection:

1. All activities assigned to Andy Applegate
   - GET `http://localhost:8080/cc/rest/common/v1/activities`
2. All claims assigned to Andy Applegate
   - GET `http://localhost:8080/cc/rest/claim/v1/claims`

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 ClaimCenter. 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",
```

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, monetary 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"
},
```

Monetary amount values are complex values with a currency field and an amount field. For example:

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

(Note that in the system APIs, the datatype is referred to as `MonetaryAmount`. But in ClaimCenter, these values are actually stored using the `CurrencyAmount` datatype.)

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 99.

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 `/claims?include=exposures,activities` returns all claims assigned to the current user, and all exposures and activities associated with those claims.

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 Claim 235-53-365870 (Public ID demo_sample:1) and its contacts, enter the following and click **Send**:

   GET `http://localhost:8080/cc/rest/claim/v1/claims/demo_sample:1?include=contacts`

   In the response payload:

   - In the data section, line 254 identifies a related contact with the ID of cc:2
   - In the related section, lines 410 to 449 provides the details about contact cc:2. (The ID is in the related section on line 424.)
4. To get a response payload for Claim 235-53-365870 (Public ID demo_sample:1) and its main contact, enter the following and click **Send**:

   GET `http://localhost:8080/cc/rest/claim/v1/claims/demo_sample:1?include=mainContact`

   In the response payload:

   - In the data section, line 250 identifies a related contact with the ID of cc:1
   - In the related section, lines 260 to 372 provides the details about contact cc:1. (The ID is in the related section on line 274.)

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",
            "relatedTo": {
                "displayName": "235-53-373871",
                "id": "demo_sample:8002",
                "type": "Claim"
            },
            "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 `/claims/{claimId}?include=contacts`. This call returns the claim with the given Claim ID and all ClaimContacts related to that claim. There are theoretically several related resources (ClaimContacts) for each primary resource (the claim with the given Claim ID).

An example of the second case is GET `/claims/{claimId}?include=mainContact`. This call returns the claim with the given Claim ID and the ClaimContact who is designated as the main contact. There is only a single related resource (the main ClaimContact) for each primary resource (the claim with the given Claim ID).

You can also specify multiple include options, as in GET `/claims/{claimId}?include=contacts,mainContact`. In this case, for each claim, the related section specifies the IDs of all related contacts and it also explicitly identifies the ID of the main contact.

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, `?filter=subject:sw:Contact%20claimant`)
- Specify Booleans as either true or false. (For example, `?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 "::". The first colon acts as an escape character. 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`.

Do not filter on the id property

In general, endpoints do not have an `id` property that is filterable. When you want to retrieve a specific resource, use an element endpoint rather than a collection endpoint with a filter on `id`. For example, do not attempt to query for activity xc:20 with this call:

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

Use this call:

GET `/activities/xc:20`

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 `/claims` endpoint has a default filter that returns only claims assigned to the caller making the API call.

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 `/claims` endpoint says: "Retrieve a list of claims, by default those assigned to the current user."

You can use the `filter` query parameter to override default filters. For example, if you are a caller who is authorized to view claims not assigned to you (such as the super user `su`), the following GET returns claims assigned to Betty Baker: GET `claims?filter=assignedUser:eq:demo_sample::8`

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:8080/cc/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:8080/cc/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
- `*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 claim resource (for example, the default fields for the `/claims/{claimId}` endpoint) and the summary fields returned for a claim collection (for example, the default fields for the `/claims` endpoint). Fields included in the Detail only are in bold:

- Detail: `assignedGroup`, `assignedUser`, `assignmentStatus`, `claimNumber`, **`description`**, `faultRating`, `flagged`, `id`, `incidentOnly`, `insured`, `jurisdiction`, `lobCode`, `lossCause`, `lossDate`, **`lossLocation`**, `lossType`, `mainContact`, **`policyAddresses`**, `policyNumber`, `reportedByType`, `reportedDate`, `reporter`, `segment`, `state`, `strategy`, `validationLevel`
- Summary: `assignedGroup`, `assignedUser`, `assignmentStatus`, `claimNumber`, `faultRating`, `flagged`, `id`, `incidentOnly`, `insured`, `jurisdiction`, `lobCode`, `lossCause`, `lossDate`, `lossType`, `mainContact`, `policyNumber`, `reportedByType`, `reportedDate`, `reporter`, `segment`, `state`, `strategy`, `validationLevel`

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:8080/cc/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:8080/cc/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.

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:8080/cc/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:8080/cc/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 `claim-1.0apiconfig.yaml`, the `ActivityPatterns` 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:8080/cc/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:8080/cc/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 25. 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 30. (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 claims. But you want the response payload to include only the main contact and reporter, and you want only the id, primary phone, and work phone for these contacts. To get this response, you must send the following:

```
GET /claims?include=mainContact,reporter
           &fields=mainContact,reporter
           &fields=mainContact:id,primaryPhone,workPhone
           &fields=reporter:id,primaryPhone,workPhone
```

Note that, in this example, the logic to restrict the returned fields to only id, primary phone, and work phone 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 claim/v1/claims/demo_sample:1?
       include=activities&
       filter=activities:escalated:eq:true
```

- **Returns:** Claim demo_sample:1 and its included activities that have been escalated

The fields parameter

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

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

```
GET claim/v1/claims/demo_sample:1?
          include=activities&
          fields=activities:id,dueDate
```

- **Returns:** Claim demo_sample:1 and its included activities. For the activities, return only `id` and `dueDate`.

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 claim/v1/claims/demo_sample:1?
          include=activities&
          sort=activities:dueDate
```

- **Returns:** Claim demo_sample:1 and its included activities, sorted by their due date.

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 claim/v1/claims/demo_sample:1?
          include=activities&
          pageSize=activities:5
```

- **Returns:** Claim demo_sample:1 and up to 5 of its included activities.

The includeTotal parameter

You can include the total number of included resources.

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

```
GET claim/v1/claims/demo_sample:1?
          include=activities&
          includeTotal=activities:true
```

- **Returns:** Claim demo_sample:1, its included activities, and the total number of included activities for demo_sample:1.

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:8080/cc/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:8080/cc/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 ClaimCenter. 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 ClaimCenter, 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 ClaimCenter.
2. In a web browser, navigate to the Swagger UI for the appropriate API.
   - For more information, see “View a system API 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 ClaimCenter. (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 ClaimCenter. (For example, there could be a validation rule in ClaimCenter 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, ClaimCenter 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 ClaimCenter. 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"
}
```

(Note that in the system APIs, the datatype is referred to as MonetaryAmount. But in ClaimCenter, these values are actually stored using the CurrencyAmount datatype.)

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 cc:2 on an instance of ClaimCenter on your machine, enter: `http://localhost:8080/cc/rest/common/v1/activities/cc:2/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:8080/cc/rest/common/v1/activities`
4. Identify the `id` of the first activity in the payload. (It is cc:20.)
5. Enter the following call, but do not click **Send** yet:

POST `http://localhost:8080/cc/rest/common/v1/activities/cc:20/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 ClaimCenter.
   a. In the response payload, note the claim number of the claim this note is related to. (It is on line 13. The claim number is 235-53-365889.)
   b. Log on to ClaimCenter as `aapplegate` and navigate to the claim.
   c. Click **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 "Litigation").

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:8080/cc/rest/common/v1/activities`
4. Identify the `id` of the first activity in the payload. (It is cc:20.)
5. Enter the following call, but do not click **Send** yet:

   POST `http://localhost:8080/cc/rest/common/v1/activities/cc:20/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": "litigation"
                }
            }
        }
}
```

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

Checking your work

1. View the new note in ClaimCenter.
   a. In the response payload, note the claim number of the claim this note is related to. (It is on line 13. The claim number is 235-53-365889.)
   b. Log on to ClaimCenter as `aapplegate` and navigate to the claim.
   c. Click **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 ClaimCenter 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 claim 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 ClaimCenter. 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 `/claim/v1/claims/{claimId}/cancel`, which cancels the corresponding draft claim
- POST `/claim/v1/claims/{claimId}/submit`, which submits the corresponding draft claim, thereby promoting it to an open claim

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 99.

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 108.

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 ClaimCenter.

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 Claim API, the Claim resource has a `witnesses` array. This is an array of ClaimContacts who are witnesses to the loss. The following PATCH payload will set the witnesses array to a single witness, the ClaimContact whose id is cc:1306. If there were witnesses in this array before the PATCH, those witnesses will be removed and the only witness will be ClaimContact cc:1306.

```
{
  "data": {
    "attributes": {
      "witnesses": [
        {
          "contact": {
            "id": "cc:1306"
          }
        }
      ]
    }
  }
}
```

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 cc:2 on an instance of ClaimCenter on your machine, enter: `http://localhost:8080/cc/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:8080/cc/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:8080/cc/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 ClaimCenter.
   a. Log on to ClaimCenter as the user `aapplegate`. Andy's landing page is the **Activities** screen, which shows the open activities assigned to him.
   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 99.

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 ClaimCenter. What it means to "remove" the resource depends on the resource type. The DELETE operation federates to the ClaimCenter 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, ClaimCenter 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 Elizabeth Lee (user name `elee`). In the base configuration, Elizabeth Lee is a manager who has permission to delete notes. As Elizabeth Lee, 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 `elee` and password `gw`.
2. Enter the following call, but do not click **Send** yet:
   a. POST `http://localhost:8080/cc/rest/common/v1/activities/cc:20/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 `elee` and password `gw`.
6. Verify that the new note exists by entering the following call and click **Send**:
   a. GET `http://localhost:8080/cc/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 `elee` and password `gw`.
8. Delete the new note by entering the following call and click **Send**:
   a. DELETE `http://localhost:8080/cc/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 99.

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 84.

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 88.

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

There also may be some situations where a given technique is required. For example, unverified policies can only be created through composite requests.

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 `/claims` to create a new claim, a set of ClaimContacts for that claim, a set of incidents for that claim, and a set of exposures for that claim.

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
- A claim and its incidents

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 /claims call to create a claim and a note. The note is the `included` resource. The included section would contain code similar to this:

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

This specifies that in order to create the note, use the POST `/claim/v1/claims/{claimId}/notes` endpoint.

The uri must start with the API name, such as "/claim/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 a claim and a note for the claim. The payload assumes there is an existing policy whose number is "FNOL-POLICY". For more information on creating policies, see “Executing FNOL” on page 117.

```
POST http://localhost:8080/cc/rest/claim/v1/claims

{
  "data" : {
    "attributes": {
      "lossDate": "2020-02-01T07:00:00.000Z",
      "policyNumber": "FNOL-POLICY"
    }
  },
  "included": {
    "Note": [
      {
        "attributes": {
            "subject": "Initial phone call",
            "body": "Initial phone call with claimant"
        },
        "method": "post",
        "uri": "/claim/v1/claims/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 claim has a "reporter". This is the ClaimContact who first reported the claim to the insurer. A claim can have any number of child ClaimContacts, but only one of those ClaimContacts can be labeled as the *reporter*.

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 /claims call to create a claim and a ClaimContact who is the "reporter". The ClaimContact is the `included` resource. The included section would contain code similar to this:

```
"included": {
  "ClaimContact": [
    {
      "attributes": {
        ...
      },
      "refid": "...",
      "method": "post",
      "uri": "/claim/v1/claims/this/contacts"
    }
  ]
}
```

This specifies that in order to create the ClaimContact, use the POST `/claim/v1/claims/{claimId}/contacts` endpoint.

The uri must start with the API name, such as "/claim/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 a claim and a ClaimContact for the claim whose relationship is "reporter". The payload assumes there is an existing policy whose number is "FNOL-POLICY". For more information on creating policies, see “Executing FNOL” on page 117.

```
POST http://localhost:8080/cc/rest/claim/v1/claims

{
  "data" : {
    "attributes": {
      "lossDate": "2020-02-01T07:00:00.000Z",
      "policyNumber": "FNOL-POLICY",
      "reporter": {
            "refid": "robertFarley"
        }
    }
  },
  "included": {
    "ClaimContact": [
      {
        "attributes": {
          "firstName": "Robert",
          "lastName": "Farley",
          "contactSubtype": "Person"
        },
        "refid": "robertFarley",
        "method": "post",
        "uri": "/claim/v1/claims/this/contacts"
      }
    ]
  }
}
```

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 a claim and a note. The uri for the exposure would have a value such as "`/claim/v1/claims/this/notes`".

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 note could reference an existing claim, such as "`/claim/v1/claims/cc:200/notes`". This uri says "create a note and attach it not to the root resource of this POST, but rather to the existing claim cc:200".

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. ClaimCenter 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 of 25 subrequests. Batch requests with more than 25 subrequests fail with a `BadInputException`.

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, ClaimCenter 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 Claim API batch from an instance of ClaimCenter on your local machine, the call would be:

```
POST http://localhost:8080/cc/rest/claim/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:8080/cc/rest/claim/v1/claims/cc:22/activities/cc:55`. The path within a batch is: `/claims/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 claim.

```
{
  "requests": [
    {
      "method": "get",
      "path": "/claims/demo_sample:1"
    },
    {
      "method": "get",
      "path": "/claims/demo_sample:2"
    },
    {
      "method": "get",
      "path": "/claims/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": "/claims/demo_sample:1",
      "query": "sort=lossDate"
    },
    {
      "method": "get",
      "path": "/claims/demo_sample:2",
      "query": "fields=*all"
    },
    {
      "method": "get",
      "path": "/claims/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"
          }
        }
      }
    }
  ]
}
```

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. ClaimCenter 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 of 25 subrequests. Composite requests with more than 25 subrequests and/or subselections fail with a `BadInputException`.

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 97.

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": "/claim/v1/claims/cc: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": "/claim/v1/claims/cc: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": "/claim/v1/claims/cc: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 97.

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.

Composite request limitations

Composite requests have the following general limitations:

- The number of subrequests and subselections in a single composite request must be 25 or less.
- 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 may be some business requirements where you are required to use a composite request. For example, when creating a new claim with an unverified policy, you must create the policy and claim in a composite request.

There are also specific business requirements where you cannot use a composite request. For example:

- You cannot have a single composite request operate on more than one claim.
- You cannot create or update the child objects of a service request.
- In a composite request, you can create and submit a service request. But you cannot advance a service request to any other stage in its life cycle (such as in progress, declined, or canceled).

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.

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 101
- “Tutorial: Assign an activity using checksums” on page 102
- “Tutorial: DELETE a note using checksums” on page 103

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 ClaimCenter 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, ClaimCenter calculates the checksum and compares that value to the submitted checksum value.

- If the values match, ClaimCenter 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, ClaimCenter determines the resource has been changed since the caller application last acquired the data. The request is not executed, and ClaimCenter 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 102.)

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:8080/cc/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:8080/cc/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 ClaimCenter. 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 ClaimCenter. 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:8080/cc/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:8080/cc/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 ClaimCenter. 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 ClaimCenter. (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 Elizabeth Lee (user name `elee`). In the base configuration, Elizabeth Lee is a manager who has permission to delete notes. As Elizabeth Lee, 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 `elee` and password `gw`.
2. Enter the following call, but do not click **Send** yet:
   a. POST `http://localhost:8080/cc/rest/common/v1/activities/cc:20/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 `elee` and password `gw`.
6. Enter the following call, but do not click **Send** yet:
   a. DELETE `http://localhost:8080/cc/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 ClaimCenter. 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 ClaimCenter. 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 ClaimCenter. For more information, see “Checksums” on page 100.
`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 ClaimCenter database and must be globally unique across all clients, APIs and web services. For more information, see “Preventing duplicate database transactions” on page 107.

Header	Datatype	Description
`GW-DoNotCommit`	Boolean	This can be used to warm up a server connection. 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 108.
`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 111.
`GW-Locale`	String	This sets the locale used when processing the request. For more information, see "Globalization" on page 111.
`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 108.
`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 108.
`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. For more information, see "Validating response payloads against additional constraints" on page 109.
`x-gwre-session`	String	This controls how related calls are routed on instances of ClaimCenter 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 32.

Header	Datatype	Description
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 unverified policy, and then a claim for that policy. This would include POSTs to other child objects as well, such as contacts, incidents, exposures, and service requests.

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 ClaimCenter 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. ClaimCenter 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 a locale-based 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 locale, the name of which is derived from the relevant ISO 3166-1 alpha-2 locale code. The previous code block displays two such properties, `JP` and `US`, for Japan and the United States, respectively. Each locale 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 locale
- `addressRequire`: The minimum subset of address fields that must be included in an address for the locale

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

ClaimCenter business flows

The *InsuranceSuite Cloud API* is a set of RESTful system APIs that expose functionality in ClaimCenter so that caller applications can request data from or initiate action within ClaimCenter.

The following topics discuss how caller applications can initiate specific ClaimCenter business flows or interact with specific types of ClaimCenter resources. This includes:

- Executing FNOL
- Working with existing claims
- Working with ClaimContacts
- Working with incidents
- Working with exposures
- Working with service requests

chapter 12

Executing FNOL

This topic describes how to create claims for FNOL (First Notice of Loss) through system APIs. This topic assumes you are familiar with the ClaimCenter FNOL process. For a more detailed discussion of the ClaimCenter FNOL process, refer to the *Application Guide*.

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

- “Tutorial: Creating a policy using the Testsupport API” on page 125
- “Tutorial: POSTing a minimal draft claim for personal auto” on page 126
- “Tutorial: PATCHing a draft claim for personal auto” on page 127
- “Tutorial: POSTing a typical draft claim for personal auto” on page 128
- “Tutorial: Submitting a draft claim” on page 138

Overview of the FNOL process

FNOL (First Notice of Loss) is the event in which the insurer is informed of a potentially covered loss.

The following section provides an overview of FNOL behavior in ClaimCenter.

Draft claims and open claims

During the FNOL process, the claim is first created. The claim passes through two states: draft and open.

- A *draft claim* is a claim that has been saved to the ClaimCenter database, but there is not yet enough information for the claim to enter the adjudication process. Draft claims are not assigned to any user.
- An *open claim* is a claim that has been saved to the ClaimCenter database with enough information to enter the adjudication process. Once a claim becomes open, it is assigned to an adjuster. (Open claims are often referred to simply as "claims".)

During the FNOL process, a claim can have two different claim numbers: a draft claim number and an open claim number.

- *Draft claim numbers* are assigned when the draft is initially saved. In the base configuration, draft claim numbers typically start with "999".
- *Open claim numbers* are assigned when the claim moves from draft to open. In the base configuration, open claim numbers typically start with "000".

All claims move from draft to open, whether they are entered by a user using the **New Claim Wizard** or are created through system APIs. However, the **New Claim Wizard** hides most of the distinction between draft and open claims. For example, in the base configuration, draft claim numbers are not shown in the user interface.

Prior to finishing a draft claim in the **New Claim Wizard**, you can cancel the draft claim by clicking the **Cancel** button. This discards the claim information from the database. This action can be taken only on draft claims. Once the **New Claim Wizard** is finished, the claim cannot be canceled.

Verified and unverified policies

When a claim is created, ClaimCenter creates a policy and attaches it to the claim. This can be either a verified policy or an unverified policy.

Verified policies

A *verified policy* is a policy that is based on information retrieved from the PAS. In a production system, verified policies are the most common types of policies.

When a claim is created, if ClaimCenter can find the corresponding policy in the PAS, it creates a copy of the policy and attaches it to the claim. This is a snapshot of the policy at the point in time that the loss occurred.

Every claim has its own copy of the policy. If two claims are created from the same policy, they will each have their own copy.

Verified policies can be used in either a test or production system. However, they require a PAS (or test PAS) and an integration point to that PAS.

Unverified policies

An *unverified policy* is a policy that is created during the FNOL process based on information supplied by an adjuster or by the caller application. This information may or may not correctly correspond to information in a PAS. Unverified policies let adjusters start the FNOL process without information from the PAS. This could be necessary if the reporter does not know or cannot recall enough information to find the policy.

Eventually, the unverified policy must be refreshed with data from the PAS, thereby converting it to a verified policy. You cannot complete the claim process and make payments on a claim while the policy is still unverified.

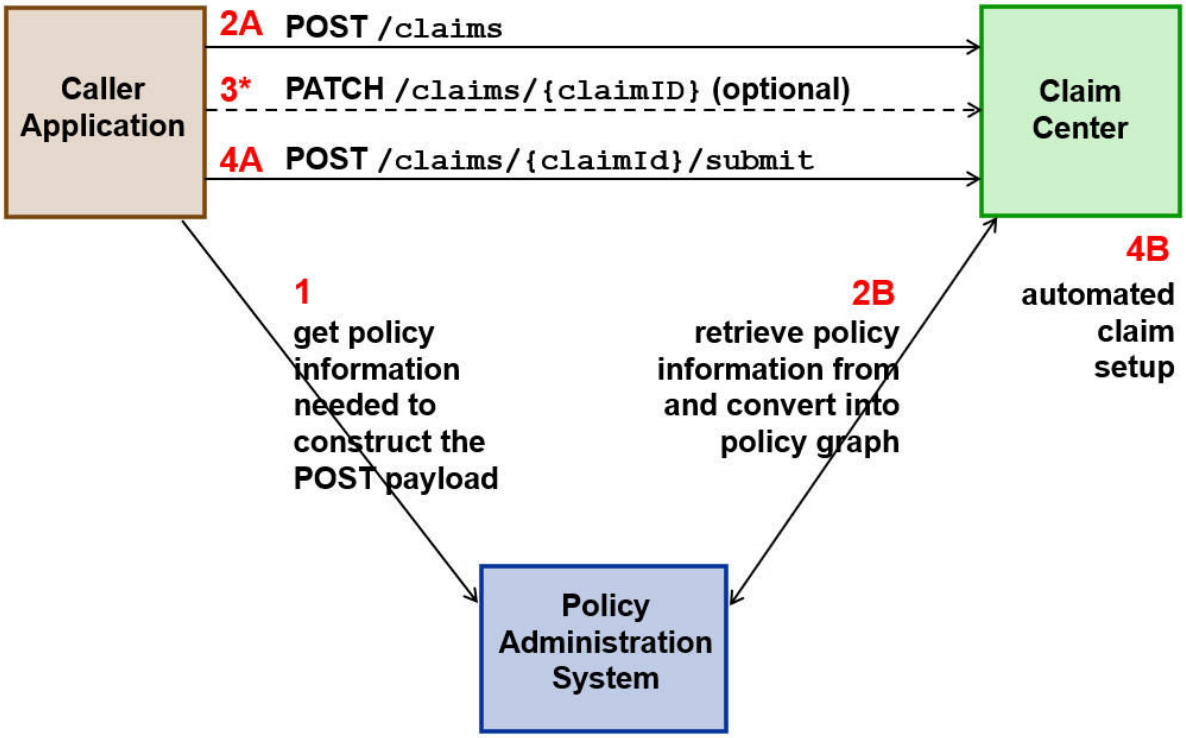
Unverified policies can be used in either a test or production system. In a test system, unverified policies may be a useful way to create policies without having an integration point to a PAS and without needing to enable the Testsupport API.

Overview of the FNOL process in the system APIs

FNOL can be accomplished entirely through the use of system APIs. The following section provides an overview of the FNOL process when it is executed through the system APIs.

The system API FNOL process

The following diagram illustrates the FNOL process as executed through the system APIs:



Prior to making any system API call to ClaimCenter:

1. The caller application queries the Policy Administration System for information on the relevant policy and its contacts, covered items, and coverages.

To execute the FNOL process with ClaimCenter:

2A. The caller application executes a POST `/claims` with a payload that describes the claim. If the call is successful, ClaimCenter saves a draft claim to the database with a draft claim number. The response object returned to the caller application includes the claim's ID and its draft claim number.

2B. During the creation of the draft claim, ClaimCenter retrieves information about the policy from the PAS and copies it into the ClaimCenter policy graph.

3. Optionally, the caller application can execute a PATCH `/claims/{claimID}` one or more times. This may be appropriate when you want to create a draft claim before you have all of the information needed to submit it, and you later want to update that claim with additional information.

4A. The caller application executes a POST `/claims/{claimId}/submit` with no payload. If the call is successful, ClaimCenter promotes the claim to being open and assigns it an open claim number. The response object returned to the caller application includes the open claim number.

4B. As part of the process to promote the claim, ClaimCenter executes the automated claim setup rules. These rules can segment the claim, assign the claim, and create and assign activities for the claim.

The entire process always consists of at least two system API calls to ClaimCenter:

- A POST `/claims` that creates a draft claim
- A POST `/submit` that submits the claim and promotes it to being open.

FNOL use cases by policy state

You cannot create a claim without a policy, and every policy belongs uniquely to one claim. (If two of more claims are based on the same policy, each has its own copy of the policy.)

There are three use cases for how FNOL can be executed through the system APIs. Each use case involves policies in a different state.

Claims with Testsupport API policies

The *Testsupport API* is an API that provides functionality to facilitate testing during development. You can use the Testsupport API to create and test policies. Testsupport API policies are appropriate for development environments that are not connected to a PAS.

You cannot PATCH a Testsupport API policy. Because the policy reflects test data, the expectation is that it is created correctly in the initial POST.

To create an open claim with a Testsupport API policy, you must execute the following calls:

1. POST `/testsupport/v1/policies` to create the test policy.
2. POST `/claims/v1/claim` to create the draft claim.
3. POST `/claims/v1/{claimId}/submit` to submit the draft claim, thereby converting it to an open claim.

Claims with verified policies

You can create claims with verified policies. This approach requires an environment connected to a PAS.

There is current no endpoint to PATCH a verified policy.

To create an open claim with a verified policy, you must execute the following calls:

1. POST `/claims/v1/claim` to create the draft claim. The policy is automatically copied over from the PAS as part of the processing of this call.
2. POST `/claims/v1/{claimId}/submit` to submit the draft claim, thereby converting it to an open claim.

Note: In ClaimCenter, the New Claim Wizard permits the creation of a claim on a verified policy that is not in effect on the loss date. This type of claim would not be allowed to complete the adjudication process unless the coverage was somehow verified. But, it can be created. In contrast, Cloud API does not permit the creation of claims on verified policies if the policy is not in force on the loss date.

Claims with unverified policies

You can create claims with unverified policies. This approach can be done in either a test or production system. The policy is based on information provided by the call, so it does not require an environment connected to a PAS. However, you typically cannot make payments on a claim while the claim's policy is still unverified.

You can PATCH an unverified policy. However, there is currently on endpoint to refresh an unverified policy, which is the action that converts an unverified policy into a verified policy.

Claims with unverified policies can only be created in a composite request. Thus, to create an open claim with a unverified policy, you must execute the following call:

1. POST `/composite/v1/composite` to create a composite request with the following subrequests:
   a. POST `/claim/v1/unverified-policies` to create an unverified policy.
   b. POST `/claims/v1/claim` to create the draft claim.
   c. POST `/claims/v1/{claimId}/submit` to submit the draft claim, thereby converting it to an open claim.

Canceling claims

You can cancel a draft claim through the system APIs. This is done using the `/claim/{claimId}/cancel` endpoint. This has the same effect as when a user clicks the **Cancel** button in the **New Claim Wizard**.

You cannot cancel a claim after it has been submitted and therefore promoted to being an open claim.

Claim modes

In the base configuration, some lines of business let you create claims in different "modes". For example, for personal auto, you can create a "regular" claim, a "quick" claim, or a "first and final" claim. These modes are distinctions that

exist primarily in the user interface. Each mode may require a greater or lesser amount of information, and some modes may assist in executing tasks beyond simply reporting the claim. But from a technical standpoint, the claims created in each mode are not fundamentally different from one another.

Thus, the system APIs do not have an analog for "modes". When you create a claim through the system APIs, as long as you provide the minimum required information, you can provide whatever amount of information is appropriate for the use case.

The Testsupport API

To create a claim, you must have information from a Policy Administration System. In some situations, you may also need information from an external user management system or contact management system. However, during development, your instance of ClaimCenter may not have access to any of these systems.

To facilitate development, Cloud API includes a Testsupport API. The *Testsupport API* is an API that provides functionality to facilitate testing during development. You can use the Testsupport API to:

- Create and search for test policies
- Create and search for test contacts and test user roles
- Create and search for test users

WARNING: The Testsupport API is intended for use in a development environment only. Do not use the Testsupport API on a production system.

Viewing Testsupport API information

The Testsupport API is available only when the ClaimCenter environment is set to `ci-test`. The environment can be set only during start-up. For more information on how to start ClaimCenter in a given environment, refer to the *System Administration Guide.*

Set the ClaimCenter environment in Studio

About this task

In Studio, you can specify an environment for ClaimCenter to use whenever it is started from Studio.

Procedure

1. In Studio, select **Run** > **Edit Configurations**. Studio opens the **Run/Debug Configurations** dialog box.
2. In the left pane, click **Server**.
3. Add the following to the end of the **VM Options** field: `-Dgw.cc.env=ci-test`
   - If the **VM Options** field already contains an `env` setting, replace the existing setting with `ci-test`.
4. Click **OK**.

Results

Whenever the server is started from Studio, it will be started using the `ci-test` environment. This includes:

- Starting the server from the **Run** menu.
- Starting the server using the **Run** tools in the upper right corner of the Studio user interface.

View the Testsupport API in Swagger UI

About this task

Once ClaimCenter has been started using the `ci-test` environment, you can use Swagger UI to view the Testsupport API.

Procedure

1. Start ClaimCenter using the `ci-test` environment.
2. 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 ClaimCenter, use: `http://localhost:8080/cc/resources/swaggerui/`
3. In the text field at the top of the Swagger UI interface, enter the following URL:
   - `<applicationURL>/rest/testsupport/v1/swagger.json`
4. Click **Explore**.

Creating test policy data

The TestSupportPolicyPlugin class

At the beginning of the FNOL process, the user or service that is executing FNOL must identify the relevant policy. Once identified, ClaimCenter copies information about that policy to its own policy graph. These processes are referred to as *policy search* and *policy retrieval*. The details for how to execute these processes are specified by the `IPolicySearchAdapter` plugin.

The base configuration includes an implementation of the `IPolicySearchAdapter` plugin that points to a Gosu class named `TestSupportPolicyPlugin`. This implementation is used only when ClaimCenter is started in the `ci-test` environment.

The Testsupport endpoints

In the base configuration, none of the API roles provide access to the endpoints in the Testsupport API. Therefore, the only user who can create test policies is the super user `su`, who inherently has access to all endpoints.

You can configure the existing API roles to extend access to these endpoints to other users. For more information on configuring API roles, see the *Cloud API Authentication Guide*.

Creating test policies

You can use the `/testsupport/v1/policies` endpoint to create test policies. When you execute this endpoint, the system APIs take the data specified in the payload and add it to the data in the `TestSupportPolicyPlugin` class. This allows you to reference test policies and their data in later calls that create claims and claim data.

There is no minimum data needed to create a test policy. If you create a policy with an empty request payload, the policy will have the following attributes:

- The policy currency will be set to the ClaimCenter default currency.
- The effective date will be set to the current date.
- The expiration date will be set to one year from the current date.
- The policy will be an unverified policy.

You can use the policies in the sample data as models for how to build test policies. To do this, identify the claim ID of a claim whose policy is an appropriate model. Then, execute a GET on any of the endpoints starting with `/claims/{claimId}/policy` to view how to structure data in a JSON payload. For a complete description of the data that can be specified in the request payload, refer to Swagger UI.

Use a unique policy number for each policy

When creating a test policy, the system APIs do not require that the policy number be unique from any existing policy numbers. However, if there are two or more policies with the same policy number, you will not be able to create a claim using that policy number. This is because the system API that creates claims will find multiple policies and will not be able to identify which policy to use.

Examples of test policy data

The following sections provide examples of some of the more frequently used fields. To access these examples compiled into a single payload, see “Sample policy payload” on page 139.

Common scalar fields

The following code block creates a policy that is effective from January 1, 2020 to January 1, 2021. Its policy number is FNOL-POLICY, and it is a verified policy.

```
{
  "data": {
    "attributes": {
      "effectiveDate": "2020-01-01T07:00:00.000Z",
      "expirationDate": "2021-01-01T07:00:00.000Z",
      "policyNumber": "FNOL-POLICY",
      "verifiedPolicy": true
    }
  }
}
```

Common typekey fields

The following code block creates a personal auto policy that is in force. (In other words, the policy has not been canceled.)

```
{
  "data": {
    "attributes": {
      "policyType": {
        "code": "PersonalAuto"
      },
      "status": {
        "code": "inforce"
      }
  }
}
```

Policy contacts

The following code block creates a policy where the insured is a person named Ray Newton who lives at 287 Kensington Rd. #1A, South Pasadena, CA. The ID in the Policy Administration for this contact is `ab:0001-1`.

When creating test policies, policy contacts are specified using request inclusion. For more information on this approach, see “Request inclusion” on page 80.

```
{
  "data": {
    "attributes": {
      "policyContacts": [
        {
          "contact": {
            "refid": "rayNewton"
          },
          "roles": [
            {
              "code": "insured"
            }
          ]
        }
      ]
    }
  },
  "included": {
    "Contact": [
      {
        "attributes": {
          "firstName": "Ray",
          "lastName": "Newton",
          "primaryAddress": {
            "addressLine1": "287 Kensington Rd. #1A",
            "city": "South Pasadena",
            "country": "US",
            "postalCode": "91145",
            "state": {
              "code": "CA"
            }
          },
```

```
          "subtype": {
            "code": "Person"
          },
          "policySystemId": "ab:0001-1"
        },
        "method": "post",
        "refid": "rayNewton",
        "uri": "/testsupport/v1/contacts"
      }

    ]
  }
}
```

Policy locations

The following code block creates a policy location.

```
{
  "data": {
    "attributes": {
      "policyLocations": [
        {
          "address": {
            "addressLine1": "287 Kensington Rd. #1A",
            "city": "South Pasadena",
            "postalCode": "91145",
            "state": {
              "code": "CA"
            }
          }
        }
      ]
    }
  }
}
```

Policy-level coverages

The following code block creates a policy-level coverage. Note that the coverage has a coverage type (a typekey field), and two currency amount fields. A policy can also contain risk unit coverages, which are shown in the next section.

```
{
  "data": {
    "attributes": {
      "policyCoverages": [
        {
          "coverageType": {
            "code": "PALiabilityCov"
          },
          "incidentLimit": {
            "amount": "30000.00",
            "currency": "usd"
          },
          "exposureLimit": {
            "amount": "15000.00",
            "currency": "usd"
          }
        }
      ]
    }
  }
}
```

Risk units

A **risk unit** is something on a policy to which coverages are attached, such as a vehicle or a building. The following code block creates a risk unit. There are different types of risk units, depending on the policy's product type. Because the other examples are from a personal auto policy, this example is a vehicle risk unit.

Risk units typically include risk unit coverages (such as Collision coverage on a vehicle). For each coverage, you must specify a `coverageType`, which must be set to a value from the `coverageType` typelist. The coverage can also have coverage terms (such as a deductible). If you include coverage terms, each coverage term must have a `covTermPattern` (set to a code from the `covTermPattern` typelist) and a `covTermSubtype` (set to a code from the

`covTerm` typelist). Additional fields may be necessary based on the type of coverage term. For example, coverage terms that are financial amounts require a `financialAmount` field.

```
{
  "data": {
    "attributes": {
      "vehicleRiskUnits": [
        {
          "RUNumber": 1,
          "vehicle": {
            "licensePlate": "1HGJ465",
            "make": "Saturn",
            "model": "SL",
            "policySystemId": "pcveh:0001-1",
            "state": {
              "code": "CA"
            },
            "vin": "1GV234TV347463345",
            "year": 1997
          },
          "coverages": [
            {
              "coverageType": {
                "code": "PACollisionCov"
              },
              "covTerms": [
                {
                  "covTermPattern": {
                    "code": "PACollDeductible"
                  },
                  "covTermSubtype": "FinancialCovTerm",
                  "financialAmount": {
                    "amount": "500.00",
                    "currency": "usd"
                  }
                }
              ],
              "incidentLimit": {
                "amount": "15000.00",
                "currency": "usd"
              }
            }
          ]
        }
      ]
    }
  }
}
```

Tutorial: Creating a policy using the Testsupport API

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 add a policy to the policy data stored in the `TestSupportPolicyPlugin` class. You can then reference this policy when creating a new claim.

1. If you have not already done so, configure Studio to start ClaimCenter using the `ci-test` environment.
   a. In Studio, select **Run** > **Edit Configurations**. Studio opens the **Run/Debug Configurations** dialog box.
   b. In the left pane, click **Server**.
   c. Add the following to the end of the **VM Options** field: `-Dgw.cc.env=ci-test`
   d. Click **OK**.
2. Start ClaimCenter from Studio by selecting **Run** > **Run** and selecting **Server**.
3. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
4. Specify *Basic Auth* authorization using user `su` and password `gw`.
5. Enter the following call, but do not click **Send** yet:
   - POST `http://localhost:8080/cc/rest/testsupport/v1/policies`
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 text in the “Sample policy payload” on page 139 into the text field underneath the radio buttons.

7. Click **Send**.

Checking your work

ClaimCenter does not show data related to policies that are not associated with a claim. Therefore, there is no independent way to check your work for this tutorial, other than confirming that you get a response code and then attempting to create a claim using the policy.

Creating test data for contacts, user roles, and users

You can also use the Testsupport API to create contacts, user roles, and users. This may be useful when you want to test functionality during development and it is too cumbersome to create the required contacts, user roles, or users manually.

POSTing a minimal draft claim

You can use the Claim API's POST `/claims` endpoint to create a draft claim. The minimum amount of information to create a draft claim is:

- Policy number
- Loss date

A policy with the given policy number must exist in the system or class that is acting as the Policy Administration System, and the loss date must occur while the policy is in force. If you provide a policy number that does not exist or is not in effect on the given date, you will get a 400 response with an error message similar to one of the following:

```
"userMessage": "No policy was found with policy number ABC123 for loss date
2020-01-01T07:00:00.000Z"
```

Tutorial: POSTing a minimal draft claim for personal auto

This tutorial assumes you have completed the following prerequisite tutorials:

- “Tutorial: Set up your Postman environment” on page 21
- “Tutorial: Creating a policy using the Testsupport API” on page 125

In this tutorial, you will create a draft claim for a personal auto policy using the minimum amount of information needed.

1. Start ClaimCenter using the `ci-test` environment.
2. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
3. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
4. Enter the following call, but do not click **Send** yet:
   - POST `http://localhost:8080/cc/rest/claim/v1/claims`
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 following into the text field underneath the radio buttons:

```
{
  "data" : {
    "attributes": {
      "lossDate": "2020-02-01T07:00:00.000Z",
      "policyNumber": "FNOL-POLICY"
    }
  }
}
```

e. If necessary, modify the payload's `lossDate` to ensure the loss occurred while the policy from the previous tutorial is in force, but not in the future.

6. Click **Send**.
7. The response payload includes the claim's ID and claim number. Copy both of these values to a separate location, as they are needed for later tutorials.

Checking your work

1. View the new draft claim in ClaimCenter.
   a. In the response payload, note the claim number of the new draft claim. (It is on or near line 8, and it likely starts with "999-99".)
   b. Log on to ClaimCenter as `su`.
   c. Click the **Claim** tab, enter the claim number in the **Claim #** menu item, and press Enter.

ClaimCenter navigates to the draft claim. Because the claim has minimal information only, ClaimCenter navigates to the **Basic Info** step of the **New Claim Wizard**. The policy number and loss date are listed in the Info Bar. (Policy number is labeled **Pol**, and loss date is labeled **DoL**.) The policy number and loss date should match the data in the POST `/claims` request payload.

Note that, in the base configuration, the **New Claim Wizard** does not display claim numbers while the claim is in a draft state.

PATCHing a draft claim

You can use the PATCH `/claim/v1/claims/{claimId}` endpoint to PATCH a draft or open claim. PATCHing a draft claim may be appropriate when you want to create the draft claim before you have all of the information needed to submit it, and you later need to update that claim with additional information. (It is also possible to create a draft claim with a single POST. For more information, see “POSTing a typical draft claim” on page 128.)

Tutorial: PATCHing a draft claim for personal auto

This tutorial assumes you have completed the following prerequisite tutorials:

- “Tutorial: Set up your Postman environment” on page 21
- “Tutorial: Creating a policy using the Testsupport API” on page 125
- “Tutorial: POSTing a minimal draft claim for personal auto” on page 126
  - That ID of the resulting claim is referred to below in this tutorial as *TutorialClaimID*.

In this tutorial, you will patch a draft claim for a personal auto policy.

1. Start ClaimCenter using the `ci-test` environment.
2. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
3. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
4. Enter the following call, but do not click **Send** yet:
   - PATCH `http://localhost:8080/cc/rest/claim/v1/claims/{`*`TutorialClaimID`*`}`
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 following into the text field underneath the radio buttons. (You may need to adjust the loss date to match the effective and expiration dates of the test policy.)

```
{
  "data" : {
    "attributes": {
      "howReported": {
        "code": "internet"
      }
```

```
    }
  }
}
```

6. Click **Send**.

Checking your work

1. View the draft claim in ClaimCenter.
   a. In the response payload, note the claim number of the new draft claim. (It is on or near line 8, and it likely starts with "999-99".)
   b. Log on to ClaimCenter as `aapplegate`.
   c. Click the **Claim** tab, enter the claim number in the **Claim #** menu item, and press Enter.

ClaimCenter navigates to the draft claim. Because the claim has minimal information only, ClaimCenter navigates to the **Basic Info** step of the **New Claim Wizard**. The **How Reported** field should be set to *Internet*.

Note that, in the base configuration, the **New Claim Wizard** does not display claim numbers while the claim is in a draft state.

POSTing a typical draft claim

You can create a typical draft claim in a single POST. This approach usually involves request inclusion. *Request inclusion* is a technique you can use with POSTs where you specify a root resource (such as a claim), one or more related child resource (such as one or more ClaimContacts), and the relationship between the root and the children. For more information, see “Request inclusion” on page 80.

New claims often include ClaimContacts, incidents, exposures, and service requests. For more information on how to work with each of these resource types, see the following:

- “Working with ClaimContacts” on page 153
- “Working with incidents” on page 163
- “Working with exposures” on page 171
- “Working with service requests” on page 181

Tutorial: POSTing a typical draft claim for personal auto

This tutorial assumes you have completed the following prerequisite tutorials:

- “Tutorial: Set up your Postman environment” on page 21
- “Tutorial: Creating a policy using the Testsupport API” on page 125

In this tutorial, you will create a draft claim for a personal auto policy using a typical amount of information needed. This claim is for Ray Newton, who has a personal auto policy that covers his Toyota Prius. On March 1, Ray hit a Honda Civic driven by Robert Farley. Both vehicles suffered minor damage.

1. Start ClaimCenter using the `ci-test` environment.
2. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
3. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
4. Enter the following call, but do not click **Send** yet:
   - POST `http://localhost:8080/cc/rest/claim/v1/claims`
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 text in the “Sample typical claim payload” on page 140 addendum into the text field underneath the radio buttons. If necessary, modify the payload's `lossDate` to ensure the loss occurred while the policy from the previous tutorial is in force, but not in the future.

6. Click **Send**.
7. The response payload includes the claim's ID and claim number. Copy both of these values to a separate location, as they are needed for later tutorials.

Checking your work

1. View the new draft claim in ClaimCenter.
   a. In the response payload, note the claim number of the new draft claim. (It is on or near line 8, and it likely starts with "999-99".)
   b. Log on to ClaimCenter as `su`.
   c. Click the **Claim** tab, enter the claim number in the **Claim #** menu item, and press Enter.

ClaimCenter navigates to the draft claim. Because the claim has minimal information only, ClaimCenter navigates to the **Basic Info** step of the **New Claim Wizard**. The policy number and loss date are listed in the Info Bar. (Policy number is labeled **Pol**, and loss date is labeled **DoL**.) The policy number and loss date should match the data in the POST `/claims` request payload.

Note that, in the base configuration, the **New Claim Wizard** does not display claim numbers while the claim is in a draft state.

Creating claims with unverified policies

An *unverified policy* is a policy that is created during the FNOL process based on information supplied by an adjuster or by the caller application. This information may or may not correctly correspond to information in a Policy Administration System.

Unverified policies let an adjuster start the FNOL process without information from the PAS. Eventually, the policy must be refreshed with data from the PAS, thereby converting it to a verified policy. You cannot complete the claim process and make payments on a claim while the policy is still unverified.

Unverified policies can be used in either a test or production system. In a test system, unverified policies may be a useful way to create policies without having an integration point to a PAS and without needing to enable the Testsupport API. In a production system, unverified policies are useful when there is a need to start the FNOL process and, for some reason, the policy is unavailable to ClaimCenter or cannot be found.

Unverified policies and composite requests

In ClaimCenter, you cannot create a policy that is not attached to a claim. Also, you cannot create a claim that has no policy. Therefore, when creating claims with unverified policies, you must create the unverified policy and the claim in the same call. The only way to do this is in the context of a composite request.

For more information on how to work with composite requests, see “Composite requests” on page 88.

The following sections discuss the endpoints used to create unverified policies and their child objects. In most cases, these endpoints are used as URIs inside a composite request, and not as the main URL for the request itself. The main URL for composite requests is always POST `/composite/v1/composite`.

Verifying unverified policies

As of this release, there are no endpoints to refresh a policy. Thus, you can create a claim with an unverified policy through the system APIs. But to complete the claims process, you must verify the policy either through the user interface or through some other integration point.

Minimum criteria for an unverified policy and claim

To create an unverified policy, use the following endpoint:

- POST `/claim/v1/unverified-policies`

To create a draft claim (regardless of whether the policy is verified or unverified), use the following endpoint:

- POST `/claim/v1/claims`

Minimum creation criteria

At a minimum, an unverified policy must have:

- A policy number (a String value)
- A policy type (a typecode from the `PolicyType` typelist)

At a minimum, a claim with an unverified policy must have:

- A policy number (which must match the unverified policy's policy number)
- A loss date

The following composite request creates an unverified policy and claim with the minimum amount of data. As is always the case with JSON, the fields can be listed in any order. Response payloads list fields in alphabetic order. However, the examples in the documentation list these fields in the most human readable order.

```
POST /composite/v1/composite

{
  "requests": [
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies",
      "body": {
        "data": {
          "attributes": {
            "policyNumber": "unverified-minimum",
            "policyType": {
              "code": "PersonalAuto"
            }
          }
        }
      }
    },
    {
      "method": "post",
      "uri": "/claim/v1/claims",
      "body": {
        "data": {
          "attributes": {
            "lossDate": "2021-03-04T07:00:00.000Z",
            "policyNumber": "unverified-minimum"
          }
        }
      }
    }
  ]
}
```

Contacts on an unverified policy

The only time you can add contacts to an unverified policy is in the composite request after the unverified policy has been created and before the claim is created. If you need to add contacts after the claim has been created, you must add them to the claim directly.

This requirement exists because all contacts in the ClaimCenter database must be ClaimContacts associated with a claim. When an unverified policy is created, any contacts associated with it are in a temporary state. When the associated claim is created, the contacts are copied over to the claim and become ClaimContacts. This occurs before the claim is committed to the database. If the system APIs gave you the ability to add contacts to the unverified policy after this point, those contacts would be associated only with the policy and would not be ClaimContacts, and ClaimCenter does not allow this.

To create a policy contact, use the following endpoint:

- POST `/claim/v1/unverified-policies/policyId/contacts`

When creating a policy contact, you must specify a `contactSubtype`. This is a typecode from the `Contact` typelist. Based on the chosen value, there may be additional required fields. For example, a contact whose `contactSubtype` is Person also requires a last name.

The following example creates an unverified policy with a policy contact (and a claim for the unverified policy). Note that the contact is created after the unverified policy and before the claim.

```
POST /composite/v1/composite

{
  "requests": [
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies",
      "body": {
        "data": {
          "attributes": {
            "policyNumber": "unverified-with-contact",
            "policyType": {
              "code": "PersonalAuto"
            }
          }
        }
      },
      "vars": [
        {
          "name": "policyId",
          "path": "$.data.attributes.id"
        }
      ]
    },
        {
      "method": "post",
      "uri": "/claim/v1/unverified-policies/${policyId}/contacts",
      "body": {
        "data": {
          "attributes": {
            "contactSubtype": "Person",
            "firstName": "Ray",
            "lastName": "Newton"
          }
        }
      }
    },
    {
      "method": "post",
      "uri": "/claim/v1/claims",
      "body": {
        "data": {
          "attributes": {
            "lossDate": "2021-03-04T07:00:00.000Z",
            "policyNumber": "unverified-with-contact"
            }
          }
        }
      }
    }
  ]
}
```

Locations on an unverified policy

To create or modify a policy location, use the following endpoint:

- POST `/claim/v1/unverified-policies/{policyId}/locations`
- PATCH `/claim/v1/unverified-policies/{policyId}/locations/{locationId}`

There is no information required to create a location on an unverified policy. ClaimCenter provides default values for all required fields.

The following example creates an unverified policy with a location.

```
POST /composite/v1/composite

{
  "requests": [
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies",
      "body": {
        "data": {
          "attributes": {
            "policyNumber": "unverified-with-location",
            "policyType": {
              "code": "PersonalAuto"
            }
          }
```

```
        }
      },
      "vars": [
        {
          "name": "policyId",
          "path": "$.data.attributes.id"
        }
      ]
    },
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies/${policyId}/locations",
      "body": {
        "data": {
          "attributes": {
          }
        }
      },
      "vars": [
        {
          "name": "locationId",
          "path": "$.data.attributes.id"
        }
      ]
    },
    {
      "method": "post",
      "uri": "/claim/v1/claims",
      "body": {
        "data": {
          "attributes": {
            "lossDate": "2021-03-04T07:00:00.000Z",
            "policyNumber": "unverified-with-location"
            }
          }
        }
      }
    }
  ]
}
```

Risk units on an unverified policy

A *risk unit* is a thing covered by the policy (other than the policyholder and any additional insureds). The type of risk units on a policy vary based on the type of policy. For example:

- On a personal auto policy or commercial auto policy, risk units are typically vehicles.
- On a homeowner's policy, risk units are typically dwellings, other structures on the property (fences, sheds), or items of value in the home (electronics, jewelry).

ClaimCenter policies make use of two types of risk units:

- Location-based risk units, for risk units that have a fixed location (such as a house)
- Vehicle risk units, for vehicles

To create a risk unit, use the following endpoints:

- POST `/claim/v1/unverified-policies/{policyId}/location-based-risk-units`
- POST `/claim/v1/unverified-policies/{policyId}/vehicle-risk-units`

To modify a risk unit, use the following endpoints:

- PATCH `/claim/v1/unverified-policies/{policyId}/location-based-risk-units/{locationBasedRiskUnitId}`
- PATCH `/claim/v1/unverified-policies/{policyId}/vehicle-risk-units/{vehicleRiskUnitId}`

The information required to create a risk unit can vary with the risk unit type. For example:

- For vehicle risk units, no information is required.
- For location-based risk units, you must provide a location.

If a field is not required and not specified, ClaimCenter provides a default value.

The following example creates an unverified policy with a vehicle risk unit.

```
POST /composite/v1/composite
```

```
{
  "requests": [
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies",
      "body": {
        "data": {
          "attributes": {
            "policyNumber": "unverified-with-vehicle-risk-unit",
            "policyType": {
              "code": "PersonalAuto"
            }
          }
        }
      },
      "vars": [
        {
          "name": "policyId",
          "path": "$.data.attributes.id"
        }
      ]
    },
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies/${policyId}/vehicle-risk-units",
      "body": {
        "data": {
          "attributes": {
          }
        }
      }
    },
    {
      "method": "post",
      "uri": "/claim/v1/claims",
      "body": {
        "data": {
          "attributes": {
            "lossDate": "2021-03-04T07:00:00.000Z",
            "policyNumber": "unverified-with-vehicle-risk-unit"
            }
          }
        }
      }
  ]
}
```

Coverages on unverified policies

There are two types of coverages on a policy: policy-level coverages and risk unit coverages.

- A *policy-level coverage* is a coverage that typically covers the policyholder or other additional insureds listed on the policy.
  - For example, personal auto policies typically come with a "Liability - Bodily Injury and Property Damage" coverage. This covers any damage to other people or other properties that is caused by the policyholder (or the additional insureds) while driving a vehicle. It does not matter which vehicle the policyholder was driving. The coverage applies to the policyholder.
- A *risk unit coverage* is a coverage that covers an associated risk unit.
  - For example, every vehicle listed on a personal auto policy typically comes with a "Collision" coverage. This covers damage done to the associated vehicle. Suppose there is a policy with two vehicles and only the first vehicle has collision coverage. If the second vehicle is involved in a collision, the policyholder will not be able to file a claim for damages done to the second vehicle.

Within the context of underwriting policies, a given type of coverage is either a policy-level coverage (and never gets attached to a risk unit) or a risk unit coverage (and always gets attached to a risk unit). However, ClaimCenter does not store information about whether a given type of coverage ought to be policy-level or risk unit level. ClaimCenter typically gets policy information from the Policy Administration System, and it assumes coverages are attached to the policy at the appropriate place.

When you create an unverified policy, it is possible to attach a coverage that is normally policy-level to a risk unit, or to attach a coverage that is normally risk unit level to the policy. This is allowed both in the ClaimCenter application and through the system APIs. However, you cannot make payments on claims with unverified policies. In order to verify a policy, you must retrieve updated information from the Policy Administration System. So if an unverified

policy has a coverage attached to the wrong location, an adjuster will need to address the error before payments on the claim can be made.

Creating an unverified policy with a policy coverage

To create or modify a policy coverage, use the following endpoints:

- POST `/claim/v1/unverified-policies/policyId/coverages`
- PATCH `/claim/v1/unverified-policies/policyId/coverages/{coverageId}`

The minimum amount of information for a policy coverage is the coverage type. This is a code from the `CoverageType` typelist. Coverage types are part of the ClaimCenter Line of Business Model, and only certain coverages can be attached to a policy based on its policy type. For more information, see the *Application Guide*.

The following example creates an unverified policy with a policy coverage.

```
POST /composite/v1/composite

{
  "requests": [
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies",
      "body": {
        "data": {
          "attributes": {
            "policyNumber": "unverified-with-policy-coverage",
            "policyType": {
              "code": "PersonalAuto"
            }
          }
        }
      },
      "vars": [
        {
          "name": "policyId",
          "path": "$.data.attributes.id"
        }
      ]
    },
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies/${policyId}/coverages",
      "body": {
        "data": {
          "attributes": {
              "coverageType" : {
                  "code": "PALiabilityCov"
              }
          }
        }
      }
    },
    {
      "method": "post",
      "uri": "/claim/v1/claims",
      "body": {
        "data": {
          "attributes": {
            "lossDate": "2021-03-04T07:00:00.000Z",
            "policyNumber": "unverified-with-policy-coverage"
            }
          }
        }
    }
  ]
}
```

Creating an unverified policy with a risk unit coverage

To create a risk unit coverage, use the following endpoints:

- POST `/claim/v1/unverified-policies/{policyId}/location-based-risk-units/ location-based-risk-units/{locationBasedRiskUnitId}/coverages`
- POST `/claim/v1/unverified-policies/{policyId}/vehicle-risk-units/{vehicleRiskUnitId}/coverages`

To modify a risk unit coverage, use the following endpoints:

- PATCH `/claim/v1/unverified-policies/{policyId}/location-based-risk-units/{locationBasedRiskUnitId}/{coverageId}`
- PATCH `/claim/v1/unverified-policies/{policyId}/vehicle-risk-units/{vehicleRiskUnitId}/{coverageId}`

The minimum amount of information for a risk unit coverage is the coverage type. This is a code from the `CoverageType` typelist. Coverage types are part of the ClaimCenter Line of Business Model, and only certain coverages can be attached to a risk unit based on the policy's policy type. For more information, see the *Application Guide*.

The following example creates an unverified policy with a vehicle risk unit and a coverage for that risk unit.

```
POST /composite/v1/composite

{
  "requests": [
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies",
      "body": {
        "data": {
          "attributes": {
            "policyNumber": "unverified-with-risk-unit-coverage",
            "policyType": {
              "code": "PersonalAuto"
            }
          }
        }
      },
      "vars": [
        {
          "name": "policyId",
          "path": "$.data.attributes.id"
        }
      ]
    },
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies/${policyId}/vehicle-risk-units",
      "body": {
        "data": {
          "attributes": {
          }
        }
      },
      "vars": [
        {
          "name": "riskUnitId",
          "path": "$.data.attributes.id"
        }
      ]
    },

    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies/${policyId}/vehicle-risk-units/${riskUnitId}/coverages",
      "body": {
        "data": {
          "attributes": {
              "coverageType" : {
                  "code": "PACollisionCov"
              }
          }
        }
      }
    },
    {
      "method": "post",
      "uri": "/claim/v1/claims",
      "body": {
        "data": {
          "attributes": {
            "lossDate": "2021-03-04T07:00:00.000Z",
            "policyNumber": "unverified-with-risk-unit-coverage"
            }
          }
        }
    }
  ]
}
```

PATCH an unverified policy

To modify information directly on the policy, use the following endpoint:

- PATCH `/unverified-policies/{policyId}`

You can theoretically PATCH an unverified policy in the same composite request that creates it. However, it is more common to PATCH an unverified policy in a subsequent call.

The following example specifies a service tier of gold for an existing unverified policy with id cc:59.

```
PATCH claim/v1/unverified-policies/cc:59

{
  "data" : {
    "attributes": {
        "serviceTier": {
              "code": "gold"
            }
    }
  }
}
```

Retrieving information about an unverified policy

You can use the following endpoints to retrieve information about an unverified policy:

- `GET claim/v1/unverified-policies/{policyId}`
- `GET claim/v1/unverified-policies/{policyId}/coverages`
- `GET claim/v1/unverified-policies/{policyId}/coverages/{coverageId}`
- `GET claim/v1/unverified-policies/{policyId}/location-based-risk-units`
- `GET claim/v1/unverified-policies/{policyId}/location-based-risk-units/{locationBasedRiskUnitId}`
- `GET claim/v1/unverified-policies/{policyId}/location-based-risk-units/{locationBasedRiskUnitId}/coverages`
- `GET claim/v1/unverified-policies/{policyId}/location-based-risk-units/{locationBasedRiskUnitId}/coverages/{coverageId}`
- `GET claim/v1/unverified-policies/{policyId}/locations`
- `GET claim/v1/unverified-policies/{policyId}/locations/{locationId}`
- `GET claim/v1/unverified-policies/{policyId}/vehicle-risk-units`
- `GET claim/v1/unverified-policies/{policyId}/vehicle-risk-units/{vehicleRiskUnitId}`
- `GET claim/v1/unverified-policies/{policyId}/vehicle-risk-units/{vehicleRiskUnitId}/coverages`
- `GET claim/v1/unverified-policies/{policyId}/vehicle-risk-units/{vehicleRiskUnitId}/coverages/{coverageId}`

Submitting a draft claim

You can use the POST `/claim/v1/claims/{claimId}/submit` endpoint to submit a draft claim. If the call executes successfully, then:

- The draft claim becomes an open claim.
- The claim is assigned an open claim number.
- Automated claim setup is executed on the claim. This includes executing business rules to:
  - Segment the claim
  - Assign the claim
  - Create and assign activities for the claim

Minimum information to submit a draft claim

From an internal standpoint, the minimum amount of information to submit a draft claim is the same as to create a draft claim:

- Policy number
- Loss date

However, ClaimCenter also includes a series of validation rules. Each rule can throw an error that is tied to a specific level of claim maturity. The two lowest levels are LoadSave and NewLossCompletion. The `/submit` endpoint will not succeed if the claim violates any rule at either of these levels.

For example, in the base configuration, there is a validation rule, "CLV04000 - ClaimContact Role Configuration", that throws an error at the NewLossCompletion level if the reporter is null. If this validation rule is not removed or modified, then a draft claim must have a specified reporter before it can be promoted to an open claim.

Draft claims that trigger errors

When you submit a draft claim, ClaimCenter generates an open claim number for the claim. If the submit action generates no errors, the open claim number is assigned to the claim in place of the draft claim number.

However, it is possible for the submit action to throw either a validation error (because it is missing information required for the LoadSave and NewLossCompletion levels) or an assignment error (because the assignment rules cannot successfully assign the claim to a group and user). If either of these occurs, the claim remains in a draft state and retains its draft claim number. The generated open claim number is discarded. The system APIs also return an error message stating that the claim could not be submitted. This error message references the claim by its draft number.

Minimum criteria for submitting a claim with an unverified policy

You can create a draft claim with only a policy number and loss date. However, in the base configuration, a claim must also have a reporter before it can be submitted.

The following composite request creates an unverified policy, a claim, and a ClaimContact who is then listed as the reporter. It then submits the claim. Note that this requires five sub-requests:

1. Create the unverified policy.
2. Create the claim.
3. Create the ClaimContact.
4. Modify the claim to assign the role of reporter to the ClaimContact.
5. Submit the claim.

In the base configuration, this is the minimum amount of information needed to create and submit a claim with an unverified policy.

```
POST /composite/v1/composite

{
  "requests": [
    {
      "method": "post",
      "uri": "/claim/v1/unverified-policies",
      "body": {
        "data": {
          "attributes": {
            "policyNumber": "unverified-minimum-submittable",
            "policyType": {
              "code": "PersonalAuto"
            }
          }
        }
      }
    },
    {
      "method": "post",
      "uri": "/claim/v1/claims",
      "body": {
        "data": {
          "attributes": {
            "lossDate": "2021-03-04T07:00:00.000Z",
            "policyNumber": "unverified-minimum-submittable"
```

```
            }
          }
        },
      "vars": [
        {
          "name": "claimId",
          "path": "$.data.attributes.id"
        }
      ]
    },
    {
      "method": "post",
      "uri": "/claim/v1/claims/${claimId}/contacts",
      "body": {
        "data": {
          "attributes": {
            "contactSubtype": "Person",
            "firstName": "Ray",
            "lastName": "Newton"
          }
        }
      },
      "vars": [
        {
          "name": "contactId",
          "path": "$.data.attributes.id"
        }
      ]
    },
    {
      "method": "patch",
      "uri": "/claim/v1/claims/${claimId}",
      "body": {
        "data": {
          "attributes": {
            "reporter": {
              "id": "${contactId}"
            }
          }
        }
      }
    },
    {
      "method": "post",
      "uri": "/claim/v1/claims/${claimId}/submit"
    }
  ]
}
```

Tutorial: Submitting a draft claim

This tutorial assumes you have completed the following prerequisite tutorials:

- “Tutorial: Set up your Postman environment” on page 21
- “Tutorial: Creating a policy using the Testsupport API” on page 125
- “Tutorial: POSTing a typical draft claim for personal auto” on page 128
  - That ID of the resulting claim is referred to below in this tutorial as *TutorialClaimID*.

In this tutorial, you will submit a draft claim for a personal auto policy.

1. Start ClaimCenter using the `ci-test` environment.
2. In Postman, start a new request by clicking the + to the right of the **Launchpad** tab.
3. Specify *Basic Auth* authorization using user `aapplegate` and password `gw`.
4. Enter the following call and click **Send**:
   - POST `http://localhost:8080/cc/rest/claim/v1/claims/{`*`TutorialClaimID`*`}/submit`

Checking your work

1. View the open claim in ClaimCenter.
   a. In the response payload, note the claim number of the new draft claim. (It is on or near line 20, and it likely starts with "000-00-".)
   b. Log on to ClaimCenter as `aapplegate`.
   c. Click the **Claim** tab, enter the claim number in the **Claim #** menu item, and press Enter.

ClaimCenter should navigates to the **Summary** screen for the claim. (If the claim is open, ClaimCenter takes you to the **Summary** screen, not the **New Claim Wizard**.)

Canceling a draft claim

You can use the POST `claim/v1/claims/{claimId}/cancel` endpoint to cancel a draft claim. If the call executes successfully, then the draft claim is discarded. All information about the draft claim is removed from the ClaimCenter database.

You can cancel only draft claims. Once a claim has been submitted, it can be closed. But it can no longer be canceled.

Sample payload addendum

This section contains sample payloads referenced in previous topics that are too long to include within those topics.

Sample policy payload

```
{
  "data": {
    "attributes": {
      "effectiveDate": "2020-01-01T07:00:00.000Z",
      "expirationDate": "2031-01-01T07:00:00.000Z",
      "policyNumber": "FNOL-POLICY",
      "verifiedPolicy": true,
      "policyType": {
        "code": "PersonalAuto"
      },
      "status": {
        "code": "inforce"
      },
      "policyContacts": [
        {
          "contact": {
            "refid": "rayNewton"
          },
          "roles": [
            {
              "code": "insured"
            }
          ]
        }
      ],
      "policyLocations": [
        {
          "address": {
            "addressLine1": "287 Kensington Rd. #1A",
            "city": "South Pasadena",
            "postalCode": "91145",
            "state": {
              "code": "CA"
            }
          }
        }
      ],
      "policyCoverages": [
        {
          "coverageType": {
            "code": "PALiabilityCov"
          },
          "incidentLimit": {
            "amount": "30000.00",
            "currency": "usd"
          },
          "exposureLimit": {
            "amount": "15000.00",
            "currency": "usd"
          }
        }
      ],
      "vehicleRiskUnits": [
        {
          "RUNumber": 1,
          "vehicle": {
            "licensePlate": "1HGJ465",
            "make": "Toyota",
            "model": "Prius",
            "policySystemId": "pcveh:0001-1",
            "state": {
              "code": "CA"
```

```
            },
            "vin": "1GV234TV347463345",
            "year": 2007
          },
          "coverages": [
            {
              "coverageType": {
                "code": "PACollisionCov"
              },
              "covTerms": [
                {
                  "covTermPattern": {
                    "code": "PACollDeductible"
                  },
                  "covTermSubtype": "FinancialCovTerm",
                  "financialAmount": {
                    "amount": "500.00",
                    "currency": "usd"
                  }
                }
              ],
              "incidentLimit": {
                "amount": "15000.00",
                "currency": "usd"
              }
            }
          ]
        }
      ]
    }
  },
  "included": {
    "Contact": [
      {
        "attributes": {
          "firstName": "Ray",
          "lastName": "Newton",
          "primaryAddress": {
            "addressLine1": "287 Kensington Rd. #1A",
            "city": "South Pasadena",
            "country": "US",
            "postalCode": "91145",
            "state": {
              "code": "CA"
            }
          },
          "subtype": {
            "code": "Person"
          },
          "policySystemId": "ab:0001-1"
        },
        "method": "post",
        "refid": "rayNewton",
        "uri": "/testsupport/v1/contacts"
      }
    ]
  }
}
```

Sample typical claim payload

```
{
  "data": {
      "attributes": {
        "lossDate": "2020-03-01T07:00:00.000Z",
        "policyNumber": "FNOL-POLICY",
        "lossCause": {
          "code": "vehcollision"
        },
        "mainContact": {
          "policySystemId": "ab:0001-1"
        },
        "reporter": {
            "policySystemId": "ab:0001-1"
        }
      }
  },
  "included": {
    "ClaimContact": [
      {
        "attributes": {
          "firstName": "Robert",
          "lastName": "Farley",
          "contactSubtype": "Person"
        },
        "method": "post",
        "refid": "robertFarley",
```

```
        "uri": "/claim/v1/claims/this/contacts"
      }
    ],
    "VehicleIncident": [
      {
        "attributes": {
          "collision": true,
          "damageDescription": "Minor collision",
          "driver": {
            "policySystemId": "ab:0001-1"
          },
          "lossParty": {
            "code": "insured"
          },
          "vehicle": {
            "policySystemId": "pcveh:0001-1"
          }
        },
      "method": "post",
      "uri": "/claim/v1/claims/this/vehicle-incidents"
      },
      {
        "attributes": {
          "collision": true,
          "damageDescription": "Minor collision",
          "driver": {
            "refid": "robertFarley"
          },
          "lossParty": {
            "code": "third_party"
          },
          "vehicle": {
            "licensePlate": "2PIX534",
            "make": "Honda",
            "model": "Civic",
            "state": {
              "code": "CA"
            },
            "vin": "3DT6YUQ3K9003LP19",
            "year": 2019
          }
        },
      "method": "post",
      "uri": "/claim/v1/claims/this/vehicle-incidents"
      }
    ]
  }
}
```

chapter 13

Working with claims

This topic covers the different ways that a caller application can get information on existing claims, and additional actions they can take on open claims.

For information on creating claims or working with draft claims, see “Executing FNOL” on page 117.

Querying for claims associated with you

Typically, the GET `/claim/v1/claims` endpoint does not return all claims in ClaimCenter. The endpoint is restricted by the caller's resource access. In the base configuration, this means the following:

- For *internal users*, the endpoint returns only claims that would be on that user's Access Control List (ACL)
- For external users who are *policyholders*, the endpoint returns only claims related to policies the user holds.
- For external users who are *vendors*, the endpoint returns only claims with a service request where the vendor is the service provider.
- For *services*, all claims are returned. (Services are not bound by resource access.)

For example, both Andy Applegate and Wendy Gompers are internal users defined in the sample data. Suppose that the sample data has been loaded and each adjuster executes the following:

GET `/claim/v1/claims?fields=id`

The count and data sections of the response payload for Andy Applegate consists of the 5 open claims on his ACL:

```
{
    "count": 5,
    "data": [
        {
            "attributes": {
                "id": "cc:34"
            }
        },
        {
            "attributes": {
                "id": "cc:33"
            }
        },
        {
            "attributes": {
                "id": "demo_sample:20"
            }
        },
        {
            "attributes": {
                "id": "demo_sample:2"
            }
        },
        {
            "attributes": {
```

```
                "id": "demo_sample:1"
            }
        }
    ],
...
```

The `count` and `data` sections of response payload for Wendy Gompers consists of the 6 open claims on her ACL:

```
{
    "count": 6,
    "data": [
        {
            "attributes": {
                "id": "trucking:7"
            }
        },
        {
            "attributes": {
                "id": "trucking:8"
            }
        },
        {
            "attributes": {
                "id": "demo_sample:30002"
            }
        },
        {
            "attributes": {
                "id": "demo_sample:30001"
            }
        },
        {
            "attributes": {
                "id": "gl:1"
            }
        },
        {
            "attributes": {
                "id": "trucking:6"
            }
        }
    ],
...
```

You can view this output for yourself in Postman by doing the following:

1. Open a request tab. From the **Authorization** tab, set the **TYPE** drop-down list to *Basic Auth*. For the **Username** and **Password**, specify `aapplegate` and `gw`.
2. Execute GET `/claim/v1/claims`.
3. Open a second request tab. From the **Authorization** tab, set the **TYPE** drop-down list to *Basic Auth*. For the **Username** and **Password**, specify `wgompers` and `gw`.
4. Execute GET `/claim/v1/claims`.
5. Compare the two response payloads.

For more information on resource access, see the *Cloud API Authentication Guide*.

Closed claims

By default, the GET `/claim/v1/claims` endpoint returns only open claims. You can query for open and closed claims by adding the following query parameter:

```
?filter=state:in:open,closed
```

Draft claims

Draft claims are claims that have been saved to the database, but the claim does not yet have enough information for it to be assigned to an adjuster or vendor. Therefore, for some callers, the GET `/claim/v1/claims` endpoint does not return draft claims.

Querying for a claim by claim ID

The GET `/claim/v1/claims/{claimId}` endpoint returns information on the given claim, assuming the caller's resource access permits the caller to view the claim.

For example, Andy Applegate owns the claim with ID cc:33. Betty Baker belongs to the same group as Andy Applegate, and therefore she has resource access permission to view claims assigned to Andy. Amy Baxter is a clerical user who works in a different group. She does not have resource access permission to view claim cc:33.

Suppose that each of these users triggers the following call:

GET `/claim/v1/claims/cc:33?fields=claimNumber`

The response for Andy Applegate is:

```
{
    "data": {
        "attributes": {
            "claimNumber": "235-53-425891"
        }
    }
}
```

The response for Betty Baker is:

```
{
    "data": {
        "attributes": {
            "claimNumber": "235-53-425891"
        }
    }
}
```

The response for Amy Baxter is:

```
{
    "status": 404,
    "errorCode": "gw.api.rest.exceptions.NotFoundException",
    "userMessage": "No resource was found at path /claim/v1/claims/cc:33"
}
```

Resource access is a component of the system API authentication and authorization framework. For more information on resource access, see the *Cloud API Authentication Guide*.

Querying for claims regardless of association

In some situations, a caller application may need to query for one or more claims that are not associated with the caller and would not appear in the results of GET `/claims`. This can be done with a claim search using the POST `/claim/v1/search/claims` endpoint. This endpoint returns a collection of `ClaimSearchView` resources. A *ClaimSearchView* is a resource that has summary information about a claim.

Note that a caller can have access to a claim's ClaimSerchView but not have access to the claim itself. In this case, the claim's ClaimSearchView would appear in the results of a POST `/claim/v1/search/claims` , and the caller would be able to access the summary information. But, any attempt to view the claim through GET `/claims/{claimId}` would fail.

Request payload for a claim search

The request object for a POST `/claim/v1/search/claims` must include a body. The body must specify the search parameters using the following syntax:

```
{
  "data": {
    "attributes": {
      "claimNumber": "stringValue",
      "firstName": "stringValue",
      "lastName": "stringValue",
      "nameSearchType": {
        "code": "ClaimSearchNameSearchTypeCode"
      },
      "policyNumber": "stringValue"
    }
  }
}
```

You must provide at least one field other than `nameSearchType`. You can provide more than one.

For example, the following payload will query for all claims associated with policy number 54-123456:

```
{
  "data": {
    "attributes": {
      "policyNumber": "54-123456"
    }
  }
}
```

The following payload will query for all claims where there is a claimant who has a first name of "Ray" and a last name of "Newton":

```
{
  "data": {
    "attributes": {
      "firstName": "Ray",
      "lastName": "Newton"
    }
  }
}
```

Searching by name

When the payload includes either the `firstName` or `lastName` field, the default behavior is to search for claims where there is a *claimant* with that first or last name.

You can also use the `nameSearchType` parameter to execute searches where the named person is either the *insured*, an *additional insured*, or has any role on the claim. To do this, provide one of the following codes for the `ClaimSearchNameSearchTypeCode`:

- `addinsured` (additional insured)
- `any` (any role on the claim, including the roles beyond additional insured, claimant, and insured)
- `claimant` (this is the default behavior)
- `insured`

For example, the following payload will query for all claims where the insured has a first name of "Ray" and a last name of "Newton":

```
{
  "data": {
    "attributes": {
      "firstName": "Ray",
      "lastName": "Newton",
      "nameSearchType": {
        "code": "insured"
      }
    }
  }
}
```

The following payload will query for all claims where there is a ClaimContact with a first name of "Ray" and a last name of "Newton", regardless of the ClaimContact's role on the claim:

```
{
  "data": {
    "attributes": {
      "firstName": "Ray",
      "lastName": "Newton",
      "nameSearchType": {
        "code": "any"
      }
    }
  }
}
```

Providing no search parameters

The system APIs do not require you to provide any query parameters, but ClaimCenter will not execute a claim search with no query parameters. If you attempt to execute a claim search without query parameters, either from the user interface or through a system API, ClaimCenter returns the following error message:

```
Please specify Claim #, Policy #, any Contact field, Assigned To Group, Assigned To User,
Created By, Cat #, VIN or License Plate
```

Note that this message is intended primarily for user interface claim searches, which is why it makes reference to fields not available to the `/claim/v1/search/claims` endpoint, such as **Assigned To Group**, **Assigned To User**, and **Created By**.

Response payload for a claim search

The `/claim/v1/claims` and `/claim/v1/claims/{claimId}` endpoints return a claim or a collection of claims. The `claim/v1/search/claims` endpoint returns a collection of ClaimSearchViews. Consequently, the three endpoints return payloads with slightly different structures.

The following table identifies the primary differences.

Information	Claim payload example	ClaimSearchView payload example
Claim owner	`"assignedUser": { "displayName": "Andy Applegate", "id": "demo_sample:1" }`	`"adjusterName": "Andy Applegate"`
Claim ID	`"id": "cc:33",`	`"claimId": "cc:33"`
Insured	`"insured": { "displayName": "Bill Kinman ", "id": "cc:33", "uri": "/claim/v1/claims/cc:33/ contacts/cc:101" }`	`"insuredName": "Bill Kinman"`
Claimants	`"included": { "ClaimContact": [ { "attributes": { ... "id": "cc:101", "roles": [ { "relatedTo": { "id": "cc:47", "type": "Exposure" }, "role": { "code": "claimant" } }, ...`	`"claimants": [ "Bill Kinman" ],`

You can use query parameters to refine the response payload to exclude default fields and include non-default fields. For more information, see “Refining response payloads” on page 47.

Retrieving policy information

During the initial POST of a draft claim, ClaimCenter copies information about the relevant policy from the Policy Administration System into ClaimCenter. This information is a snapshot of the policy as it existed on the claim's loss date.

The Policy Administration System is considered the System of Record for policy information. Consequently, for verified policies, you cannot edit policy information in ClaimCenter through the system APIs. (The user interface does

allow you to edit policy information, though this causes the policy to become unverified. For more information on unverified policies, refer to the *Application Guide*.)

The system APIs include several endpoints that let you view policy information.

Summary of the policy endpoints

The information returned by the following endpoints comes from the ClaimCenter snapshot of the policy. It does not come directly from the Policy Administration System.

The policy itself

The following endpoint returns information that is directly on the policy resource, such as effective date, expiration date, policy number and policy type:

- `/claims/{claimId}/policy`

Risk units

A *risk unit* is a thing covered by the policy (other than the policyholder and any additional insureds). The type of risk units on a policy vary based on the type of policy. For example:

- On a personal auto policy or commercial auto policy, risk units are typically vehicles.
- On a homeowner's policy, risk units are typically dwellings, other structures on the property (fences, sheds), or items of value in the home (electronics, jewelry).

The following endpoints return information about the risk units on the policy:

- `/claims/{claimId}/policy/location-based-risk-units`
- `/claims/{claimId}/policy/location-based-risk-units/{locationBasedRiskUnitId}`
- `/claims/{claimId}/policy/vehicle-risk-units`
- `/claims/{claimId}/policy/vehicle-risk-units/{vehicleRiskUnitId}`

Coverages

There are two types of coverages on a policy: policy-level coverages and risk unit coverages.

- A *policy-level coverage* is a coverage that typically covers the policyholder or other additional insureds listed on the policy.
  - For example, personal auto policies typically come with a "Liability - Bodily Injury and Property Damage" coverage. This covers any damage to other people or other properties that is caused by the policyholder (or the additional insureds) while driving a vehicle. It does not matter which vehicle the policyholder was driving. The coverage applies to the policyholder.
- A *risk unit coverage* is a coverage that covers an associated risk unit.
  - For example, every vehicle listed on a personal auto policy typically comes with a "Collision" coverage. This covers damage done to the associated vehicle. Suppose there is a policy with two vehicles and only the first vehicle has collision coverage. If the second vehicle is involved in a collision, the policyholder will not be able to file a claim for damages done to the second vehicle.

The following endpoints return information about the policy-level coverages on the policy:

- `/claims/{claimId}/policy/coverages`
- `/claims/{claimId}/policy/coverages/{coverageId}`

The following endpoints return information about the risk units on the policy. This includes the risk unit coverages attached to each risk unit:

- `/claims/{claimId}/policy/location-based-risk-units`
- `/claims/{claimId}/policy/location-based-risk-units/{locationBasedRiskUnitId}`
- `/claims/{claimId}/policy/vehicle-risk-units`
- `/claims/{claimId}/policy/vehicle-risk-units/{vehicleRiskUnitId}`

Locations

A *location* is a physical place listed on a policy. The ways in which locations are used vary based on the type of policy. For example:

- On a personal auto policy, a location can be used to identify where a vehicle is garaged.
- On a homeowner's policy, a location can be used to identify where the home is located.

The following endpoints return information about the locations on the policy:

- `/claims/{claimId}/policy/locations`
- `/claims/{claimId}/policy/locations/{locationId}`

Endorsements

An *endorsement* is a physical document detailing some aspect of the policy. Occasionally, an endorsement can become relevant to claims processing. Endorsements are also referred to as *forms*.

For example, suppose a home owner elects to get a homeowner's policy for a home that is in a flood zone. The insurer attaches an endorsement to the policy that excludes any damage caused by flooding. Later, the home owner files a claim for damage caused by a flood. When determining if payment will be made on the claim, the adjuster needs to see if the policy included a flood damage exclusion endorsement.

The following endpoints return information about the endorsements on the policy:

- `/claims/{claimId}/policy/endorsements`
- `/claims/{claimId}/policy/endorsements/{endorsementId}`

Assigning claims

When a claim completes the FNOL process (either through the user interface or through the `/submit` endpoint), it is assigned to a group and a user in that group. The assigned user has the primary responsible for managing the claim.

When you submit a claim through the system APIs, ClaimCenter automatically executes the claim assignment rules to initially assign the claim to a group and user. You can use the POST `/claims/{claimId}/assign` endpoint to reassign the claim as needed.

> **Note:** The functionality for assigning claims is a subset of the functionality for assigning activities. All assignment options that are applicable to both activities and claims have the same behavior.

Assignment options

A claim 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 ClaimCenter uses assignment rules to select a user in that group)
- By re-running the claim assignment rules
  - This can be appropriate if you have modified the claim since the last time assignment rules were run and the modification might affect who the claim would be assigned to.

The root resources for the `/claims/{claimId}/assign` endpoints is `ClaimAssignee`. This resource specifies assignment criteria. The schema has the following fields:

Field	Type	Description
`autoAssign`	Boolean	Whether to assign the claim using assignment rules
`groupId`	string	The ID of the group to assign the claim to
`userId`	string	The ID of the user to assign the claim to

The `ClaimAssignee` resource cannot be empty. It must specify a single logical assignment option (group and user, group only, or automatic assignment). For more information on how assignment rules execute assignment, see the *Rules Guide*.

Assignment example - Assigning to a specific group (and user)

The following assigns claim cc:34 to group demo_sample:31 (Auto1 - TeamA) and user demo_sample:2 (Sue Smith).

```
POST /claim/v1/claims/cc:34/assign

{
  "data": {
    "attributes" : {
      "groupId" : "demo_sample:31",
      "userId" : "demo_sample:2"
    }
  }
}
```

The following assigns claim cc:34 to group demo_sample:31 (Auto1 - TeamA). Because no user has been specified, ClaimCenter will execute assignment rules to assign the claim to a user in group demo-sample:31.

```
POST /claim/v1/claims/cc:34/assign

{
  "data": {
    "attributes" : {
      "groupId" : "demo_sample:31"
    }
  }
}
```

Note that there is currently no endpoint that returns groups or group IDs. To assign claims to a specific group, the caller application must determine the group ID using some method other than a groups system API.

Assignment example - Using automated assignment

The following assigns claim cc:34 using automated assignment rules.

```
POST /claim/v1/claims/cc:34/assign

{
  "data": {
    "attributes": {
        "autoAssign" : true
    }
  }
}
```

Validating claims

ClaimCenter validation levels

During a claim's lifecycle, a claim passes through one or more levels of maturity. Within ClaimCenter, these are called *validation levels*. The base configuration comes with the following levels:

- **Load and save** - The claim has enough information to be saved to the database.
- **New loss completion** - The claim has enough information to be assigned to an adjuster.
- **Valid for ISO** - The claim has enough information to be filed with ISO. (ISO is a national database used in the United States to verify that the same loss is not being filed with multiple insurers.)
- **Send to external (systems)** - The claim has enough information to send information about it to external systems within the insurer, such as a Policy Administration System that may be trying to assess policy renewal rates.
- **Ability to pay** - The claim has enough information such that payments can be written for it.

A claim's validation level is determined and enforced by a set of claim validation rules. Whenever a change is made to a claim, the validation rules determine if the claim can be advanced to a later stage of validation. The validation rules also prevent a claim from moving backwards to a lower level of validation. For more information on validation rules, see the *Rules Guide*.

Note: In the base configuration, the "load and save" level applies only to claims that are being imported through the ClaimCenter SOAP-based `ClaimAPI` API. Draft claims submitted through the system APIs do not need to

pass any level. In order for a draft claim to be promoted to an open claim, the draft claim must pass both the "load and save" level and the "new loss completion" level. For more information, see “Executing FNOL” on page 117.

Validating a claim through the system APIs

The Claim API includes a POST `/claim/{claimId}/validate` endpoint. This endpoint can be used to:

- Determine the claim's current validation level
- Perform validation on the claim for a specific validation level. (This returns information that identifies the conditions that must be true for the claim to advance to the specified level.)

Checking a claim's validation level can be useful in the following situations:

- You want to determine whether or not the claim has enough information to be assigned to an adjuster. You can use the `/validate` endpoint to determine if the claim is at or beyond the "new loss completion" level.
  - If the claim is below the "new loss completion" level, the payload identifies the conditions needed to reach "new loss completion".
- You want to execute a payment for a claim. You can use the `/validate` endpoint to determine if the claim is at the "ability to pay" level.
  - If the claim is below the "ability to pay" level, the payload identifies the conditions needed to reach "ability to pay".

Example of a claim at the "load save" level

Suppose you execute a POST `/claim/{claimId}/validate` for claim cc:706 and this is the response:


```
{
    "data": {
        "attributes": {
            "hasErrors": true,
            "validationIssues": [
                {
                    "field": "contacts",
                    "id": "cc:706",
                    "message": "The role Reporter is required on Claim 999-99-999705.",
                    "severity": {
                        "code": "error",
                        "name": "Error"
                    },
                    "type": "Claim",
                    "url": "/claim/v1/claims/cc:706",
                    "validationLevel": {
                        "code": "newloss",
                        "name": "New loss completion"
                    }
                },
                {
                    "field": "lossLocation",
                    "id": "cc:706",
                    "message": "The claim's loss location must not be null",
                    "severity": {
                        "code": "error",
                        "name": "Error"
                    },
                    "type": "Claim",
                    "url": "/claim/v1/claims/cc:706",
                    "validationLevel": {
                        "code": "payment",
                        "name": "Ability to pay"
                    }
                }           ],
            "validationLevelReached": {
                "code": "loadsave",
                "name": "Load and save"
            }
        }
    }
}
```


From this payload, you can determine the following:

- The claim is at the "load and save" level. (`validationLevelReached.code` is `loadsave`).

- To move to the "new loss completion" level, the reporter must be specified. (This comes from the first `validationIssues` message.)
- To move to the "ability to pay" level, the loss location must be non-null. (This comes from the second `validationIssues` message.)
  - Although it is not explicitly stated, the requirements for all previous levels must also be met. (In other words, to reach "ability to pay", the loss location must be non-null and the reporter must be specified.)

Example of a claim at the "ability to pay" level

Suppose you execute a POST `/claim/{claimId}/validate` for claim demo_sample:31 and this is the response:

```
{
    "data": {
        "attributes": {
            "hasErrors": false,
            "validationLevelReached": {
                "code": "payment",
                "name": "Ability to pay"
            }
        }
    }
}
```

From this payload, you can determine that the claim is at "ability to pay". The claim satisfies all validation rules. (The `hasErrors` value is false.)

chapter 14

Working with ClaimContacts

This topic provides a high-level overview of ClaimContacts, discussing both what they are and how to work with them through the system APIs.

For a more detailed discussion of the business functionality of ClaimContacts, refer to the *Application Guide*.

Overview of ClaimContacts in ClaimCenter

The following section provides an overview of ClaimContact behavior in ClaimCenter.

Note that, in ClaimCenter, ClaimContact information is stored across multiple entities, including `Contact`, `ClaimContact`, and `ClaimContactRole`. The system APIs capture this information in a single resource named `ClaimContact`. This documentation uses the term "ClaimContact" to refer to a ClaimContact resource in the system APIs, or its corresponding information in the ClaimCenter `Contact`, `ClaimContact`, and `ClaimContactRole` entities.

What is a ClaimContact?

A *ClaimContact* is a person or organization who has a relationship with a claim. This includes people and organizations who:

- Are covered by the relevant policy
- Suffered a covered loss
- Provided information relevant to the claim
- Provided a service to address the loss

For example, suppose that Ray Newton has a personal auto policy. He informs the insurer that, while driving his Toyota, he hit Robert Farley's Honda and damaged both cars. Wilma Weeks witnessed the collision. Robert Farley's Honda was repaired at Joe's Auto Body Shop. This claim has the following ClaimContacts:

- Ray Newton, who is covered by the personal auto policy and who suffered a loss.
- Robert Farley, who also suffered the loss.
- Wilma Weeks, who provided information relevant to the claim.
- Joe's Body Shop, who provided service to address the loss.

What is a ClaimContact related to?

Claims typically have child objects. This can include:

- A **policy**, which contains a copy of information from the policy that is relevant to the claim.
- One or more **incidents**, which represent anything that was damaged, stolen, or otherwise representative of the loss (such as a vehicle, a property, or an injured person).

- One or more **exposures**, which track a potential payment for one claimant from one coverage.
- One or more **service requests**, which are requests to outside vendors to provide service that addresses the loss.

Every ClaimContact is related to the claim itself. A ClaimContact can also be related to one or more specific child objects.

For example, the claim described above might have the following child objects:

- The personal auto policy
  - Ray Newton is related to this.
- A vehicle incident for Ray's damaged Toyota.
  - Ray Newton is related to this.
- An exposure to pay Ray Newton from the policy's collision damage coverage.
  - Ray Newton is related to this.
- A vehicle incident for Robert's damaged Honda.
  - Robert Farley is related to this.
- An exposure to pay Robert Farley from the policy's third-party property damage coverage.
  - Robert Farley is related to this.
- A service request to repair Robert's Honda.
  - Joe's Body's Shop is related to this.

What is the nature of the relationship?

Every ClaimContact must have one or more roles with each object the ClaimContact is related to. A *ClaimContact role* defines the nature of a relationship between a ClaimContact and the claim.

For example, the claim described above might have these ClaimContacts with the following roles:

- The claim itself
  - Ray Newton is the *insured*, the *reporter*, and a *claimant*.
  - Robert Farley is a *claimant*.
  - Wilma Weeks is a *witness*.
- The vehicle incident for Ray's Toyota.
  - Ray Newton is the *driver*.
- The vehicle incident for Robert's Honda.
  - Robert Farley is the *driver*.
- The exposure to pay Ray Newton from the policy's collision coverage.
  - Ray Newton is the *claimant*.
- The exposure to pay Robert Farley from the policy's third-party property damage coverage.
  - Robert Farley is the *claimant*.
- The service request to repair Robert's Honda.
  - Joe's Body's Shop is the *service vendor*.

Overview of ClaimContacts in the system APIs

The following section provides an overview of ClaimContact behavior as it exists in the system APIs.

ClaimContact roles

Every ClaimContact has a `roles` array. This is a read-only list of all the roles the ClaimContact has.

Every member of the `roles` array includes the following properties:

- `relatedTo` - the type and ID of the object that the ClaimContact is related to
- `role` - the role the ClaimContact has on that object

- `active` - a Boolean identifying whether the ClaimContact actively holds the role on the claim.

The `active` field is used to identify ClaimContacts who previously held a role on the claim but are no longer actively involved in the claim. For example, suppose an injured person is treated by one doctor, but then the case is reassigned to a second doctor. Both doctors could be ClaimContacts on the claim, but `active` would be set to true only for the second doctor.

You can modify the roles a ClaimContact has, but this is not done by modifying the `roles` array. The way in which it is done depends on whether the role is reserved or not.

Reserved roles

A *reserved role* is a role that cannot be set on a ClaimContact explicitly. Instead, the role must be set implicitly through a field, array, or action on another object.

For example, *reporter* is a reserved role. You cannot add this role directly to a ClaimContact. However, you can set a Claim's `reporter` field to a given ClaimContact. This implicitly adds the *reporter* role to that ClaimContact. This also removes the *reporter* role from any other ClaimContact that previous had it.

The reserved roles are defined in the `ReservedContactRoles.yaml` file in the `integration/contactroles/v1` directory. In general, the reserved roles are either:

- Roles for which there can be at most one ClaimContact with the role. (For example, *reporter* is reserved. A claim can have at most one reporter.)
- Roles that are set through an array on a non-ClaimContact object. (For example, *witness* is reserved. A claim can have several witnesses. These witnesses are defined on the `Claim` resource's `witnesses` array.)

For more information on assigning a reserved role to a ClaimContact, see “Setting reserved roles” on page 156.

Non-reserved roles

A *non-reserved role* is a role that can be set on a ClaimContact explicitly. Every role that is not listed in the `ReservedContactRoles.yaml` file is a non-reserved role. For example, *alternate contact* is a non-reserved role. A claim can have any number of alternate contacts, and this type of ClaimContact is not managed by an array on Claim.

For more information on assigning a non-reserved role to a ClaimContact, see “Setting non-reserved roles” on page 157.

Identifiers

When specifying a ClaimContact in a payload, there are several different identifiers you can use.

- `id` - The ClaimContact's system API ID. This is equal to the ClaimContact's Public ID in ClaimCenter.
- `policySystemId` - An identifier in the Policy Administration System that uniquely identifies the contact.
- `refid` - When the ClaimContact is being created in a given payload, other parts of the payload can reference it using an arbitrary "reference id".

For more information on the different options for identifying a ClaimContact, see “Identifying the ClaimContact” on page 159.

Contrasting ClaimContacts and "contacts"

The name of the resource that captures contact information is ClaimContact. This documentation refers to contacts related to claims as ClaimContacts.

Be aware that there are places where the system APIs use the term "contacts" to refer to ClaimContacts:

- For endpoints that have ClaimContact as the root resource, the endpoint path refers to the resources as a "contact". For example:
  - GET `/claim/v1/claims/{claimId}/contacts`
  - PATCH `/claim/v1/claims/{claimId}/contacts/{contactId}`
- When using the `include` query parameter to include related ClaimContacts, the resources are referred to as "contacts". For example:

- GET `/claim/v1/claims?include=contacts`

ClaimContact endpoints

You can use the following endpoints to interact with ClaimContacts directly:

Operation	Endpoint	Description
GET	`/claims/{claimId}/contacts`	Retrieve the ClaimContacts for a given claim
POST	`/claims/{claimId}/contacts`	Create a new ClaimContact on the given claim
GET	`/claims/{claimId}/contacts/{contactId}`	Retrieve information about the given ClaimContact
PATCH	`/claims/{claimId}/contacts/{contactId}`	Update information on the given ClaimContact
DELETE	`/claims/{claimId}/contacts/{contactId}`	Delete the given ClaimContact
GET	`/claims/{claimId}/contact-role-owners`	Retrieve a list of objects on the given claim that can have ClaimContacts associated with them

For reserved roles, you can also modify a ClaimContact indirectly by modifying the object that controls the role. For example, when you execute a PATCH `/claims/{claimId}` and set or modify the Claim's `reporter` field to a given ClaimContact, this assigns the *reporter* role to that ClaimContact.

The `/claims/{claimId}/contact-role-owners` endpoint returns all objects on the claim that can have ClaimContacts associated with them. This includes:

- The claim itself
- The policy
- Any existing incidents
- Any existing exposures
- Any existing service requests
- Any existing negotiations or matters
  - A *negotiation* is a history of the offers and counter-offers related to one disputed aspect of the loss.
  - A *matter* is a collection of information pertaining to a lawsuit or potential lawsuit.

Be aware that the `/claims/{claimId}/contact-role-owners` endpoint returns the objects that are able to have associated ClaimContacts. These objects may or may not have ClaimContacts already associated with them. If there are ClaimContacts associated with them, the ClaimContacts are not included in the response

Modifying ClaimContact roles

The ClaimContact resource has two role-related array properties:

- `roles` - A read-only array of all roles held by the ClaimContact
- `editableRoles` - An editable array of non-reserved roles held by the ClaimContact

Both properties use the `ContactRole` schema.

You can modify the roles a ClaimContact has, but this is never done by modifying the `roles` array. Instead, you either modify a field or array on a related object, or you modify the `editableRoles` array. Which approach to use is determined by whether the role is reserved or not.

Setting reserved roles

A *reserved role* is a role that cannot be set on a ClaimContact explicitly. Instead, the role must be set by:

- Setting a field on another object
- Modifying an array on another object
- Executing an action on another object

The reserved roles are defined in the `ReservedContactRoles.yaml` file in the `integration/contactroles/v1` directory.

To assign a reserved role to a ClaimContact, you must identify the field, array, or action that implicitly sets the role.

Reserved roles that are set from a field

For example, the *reporter* role is set from the Claim's `reporter` field. To add the *reporter* role to a ClaimContact, modify the Claim's `reporter` field so that it references the ClaimContact.

Suppose that there is a claim with ID cc:610 , and there is a ClaimContact with ID cc:1306. The following is an example of adding the *reporter* role to that ClaimContact:

```
PATCH http://localhost:8080/cc/rest/claim/v1/claims/cc:610

{
  "data": {
    "attributes": {
      "reporter": {
        "id": "cc:1306"
      }
    }
  }
}
```

Reserved roles that are set from an array

As another example, the *witness* role is set from the Claim's `witnesses` array. To add the *witness* role to a ClaimContact, add the ClaimContact the `witnesses` array.

Suppose that there is a claim with ID cc:610 , and there is a ClaimContact with ID cc:1306. The claim has no witnesses. The following is an example of adding the *witness* role to ClaimContact cc:1306.

```
PATCH http://localhost:8080/cc/rest/claim/v1/claims/cc:610

{
  "data": {
    "attributes": {
      "witnesses": [
        {
          "contact": {
            "id": "cc:1306"
          }
        }
      ]
    }
  }
}
```

Keep in mind that, within the system APIs, PATCHing an array does not add new members to the existing members. It replaces the existing members with the new members. If you want to add members to an array, you must first determine the existing members, and then specify an array with those members and the ones you wish to add. For more information, see “PATCHes” on page 73.

Reserved roles that are set through actions

In some situations, a reserved role is set when an action is executed on a resource other than the ClaimContact itself. For example, when a service request is created, the ServiceRequestInstruction can specify a ClaimContact as the CustomerContact. This ClaimContact is given the reserved role *servicerequestparticipant*.

Setting non-reserved roles

A *non-reserved role* is a role that can be set on a ClaimContact explicitly. Every role that is not listed in the `ReservedContactRoles.yaml` file is a non-reserved role.

To assign a non-reserved role to a ClaimContact, you must modify the ClaimContact's `editableRoles` array.

JSON syntax for the editableRoles array

When POSTing or PATCHing a ClaimContact, every member of the `editableRoles` array must include three pieces of information:

- The role's code
- The type of object on which the ClaimContact has this role
- The ID of the object on which the ClaimContact has this role

The syntax used to specify this is:

```
"editableroles": [
  {
    "role": {
      "code": "<roleCode>"
    },
    "relatedTo": {
      "type": "<parentObjectType>",
      "id": "<parentObjectId>"
    }
  },
  ... <additionalRoles>
```

For example, the following PATCHes Claim cc:610 so that ClaimContact cc:777 has the *alternate contact* role (whose code is `altcontact`) on the claim itself.

```
PATCH http://localhost:8080/cc/rest/claim/v1/claims/cc:610/contacts/cc:777

{
  "data": {
    "attributes": {
      "editableRoles": [
        {
          "role": {
            "code": "altcontact"
          },
          "relatedTo": {
            "type": "Claim",
            "id": "cc:610"
          }
        }
      ]
    }
  }
}
```

Similarly, this example shows how to PATCH Claim cc:610 so that ClaimContact cc:208 has the *owner* role (whose code is `incidentowner`) on the vehicle incident whose ID is cc:102. (In other words, ClaimContact cc:208 is the owner of the vehicle specified in vehicle incident cc:102.)

```
PATCH http://localhost:8080/cc/rest/claim/v1/claims/cc:610/contacts/cc:208

{
  "data": {
    "attributes": {
      "editableRoles": [
        {
          "role": {
            "code": "incidentowner"
          },
          "relatedTo": {
            "type": "vehicleIncident",
            "id": "cc:102"
          }
        }
      ]
    }
  }
}
```

PATCHing editableRoles scenarios

Keep in mind that, within the system APIs, PATCHing an array does not add new members to the existing members. It replaces the existing members with the new members. If you want to add members to an array, you must first determine the existing members, and then specify an array with those members and the ones you wish to add. For more information, see “PATCHes” on page 73.

When PATCHing `editableRoles`, the following table details the possible request payloads and they way the system APIs will respond.

If the request payload contains...	...then...
No `editableRoles` array	The non-reserved roles on the ClaimContact remain unchanged.
An `editableRoles` array with one or more non-reserved roles	The existing non-resolved roles are replaced by the non-reserved roles specified in the payload.
An empty `editableRoles` array	All existing non-reserved roles are removed. (However, if this would result in the ClaimContact no longer having any roles, the system API returns an error.)
An `editableRoles` array with one or more reserved roles	The system API returns an error.
A `roles` array	The system API returns an error.

Identifying the ClaimContact

There are several ways you can add ClaimContact information to a claim. You can:

- Create a new ClaimContact and specify its role
- Specify a role for a contact that is on the policy in the Policy Administration System
- Specify a role for a ClaimContact that is already on the claim

Each of these approaches uses a different property to identify the ClaimContact.

Creating a new ClaimContact and specifying its role

You can create a new ClaimContact and specify its role in the same payload using request inclusion. When using this approach, you identify the ClaimContact by `refid`. For more information on this approach, see “Request inclusion” on page 80.

The following example is the payload for a PATCH to an existing claim with id cc:402. This PATCH creates a new ClaimContact and sets the claim's reporter to that ClaimContact. Note that the value used for `refid`, "newContact", is arbitrary. Any value could be used so long as the same value is used for the reporter's `refid` in the `data` section and the `refid` in the `included` ClaimContact section.

```
PATCH /claim/v1/claim/cc:402

{
  "data": {
    "attributes": {
      "reporter": {
        "refid": "newContact"
      }
    }
  },
  "included": {
    "ClaimContact": [
      {
        "refid": "newContact",
        "attributes": {
            "contactSubtype": "Person",
            "firstName": "Carol",
            "lastName": "Daniels"
        },
        "method": "post",
        "uri": "/claim/v1/claims/cc:402/contacts"
      }
    ]
  }
}
```

Specifying a role for a ClaimContact that is already on the claim

When a ClaimContact is created in ClaimCenter, it is assigned an `id`. This value is the ClaimContact's Public ID in ClaimCenter.

ClaimContact roles can be specified using the `id` field. This is useful when the caller application is constructing the payload for a PATCH to an existing claim, the ClaimContact already exists on the claim, and the caller application knows the value of the ClaimContact's `id`.

The following example is the payload for a PATCH to an existing claim. This PATCH sets the Claim's `reporter` to the ClaimContact whose `id` is cc:202.

```
{
  "data": {
    "attributes": {
      "reporter": {
        "id": "cc:202"
      }
    }
  }
}
```

Specifying a role for a contact that is on the policy

When a draft claim is initially POSTed to ClaimCenter, information on the policy is copied from the Policy Administration System to ClaimCenter. This typically includes contacts listed on the policy.

Policy system IDs

One of the policy contact attributes that is copied to ClaimCenter is the contact's policy system ID. *Policy system ID* is a value that uniquely identifies a given type of object on the policy. For example, on a given policy, every contact's policy system ID must be used by only one contact.

The value that is used as the policy system ID is determined by the integration code that copies the policy information over to ClaimCenter. This code is written during implementation and will be different for each insurer and each Policy Administration System. A policy system ID is not required to be unique across the entire Policy Administration System. But, it is required to be unique across all instances of the given type of object on a given policy.

Specifying roles by policy system ID

In the system APIs, the name of the policy system ID field is `policySystemId`. Specifying roles through `policySystemId` is useful in the following circumstances:

- The caller application is constructing the payload for the initial POST `/claims` and wants to include an existing ClaimContact and its role in the payload. At this point, the policy information (including policy contacts) have not yet been copied to ClaimCenter. Therefore, the relevant contact does not yet have a ClaimCenter ID.
- The caller application is constructing the payload for a PATCH to an existing draft or open claim. The application knows the policy system ID from a previous call to the Policy Administration System, and it does not want to execute a separate GET to retrieve the ClaimCenter ID. Therefore, it identifies the ClaimContact by policy system ID.

  Note: All resource field names are case-sensitive. Unlike `refid` and `id`, which use lower-case i's, `policySystemId` uses an upper-case I.

The following example is the payload for a PATCH to an existing claim. This PATCH sets the Claim's `reporter` to the ClaimContact whose policy system ID is ab:0001-1.

```
{
  "data": {
    "attributes": {
      "reporter": {
        "policySystemId": "ab:0001-1"
      }
    }
  }
}
```

ClaimContact role constraints

Role constraints

In ClaimCenter, a *role constraint* is an logical expression that prevent users from assigning roles to ClaimContacts in a manner that does not make business sense.

There are two types of role constraints:

- **Entity role constraint** - This identifies which type of objects can make use of the role, and how many ClaimContacts can be associated with that object using that role (exactly one, at least one, at most one, or unlimited).
  - For example, this constraint could stipulate that the role of *driver* can be held by ClaimContacts associated with a vehicle incident, and that there can be at most one driver on a given vehicle incident.
  - This type of constraint can be thought of as both a "which type of object" constraint and a "how many" constraint.
- **Contact role type constraint** - This identifies the subtype for which a given role is allowed.
  - For example, this constraint could stipulate that the role of *primary doctor* can be held by ClaimContacts with an associated contact whose subtype is Doctor, but not ClaimContacts with an associated contact whose subtype is Attorney.
  - This type of constraint can be thought of as a "which subtype" constraint.

In ClaimCenter, ClaimContact role constraints are configured in `entityroleconstraints-config.xml`. For more information, refer to the *Configuration Guide*.

Role constraint endpoints

You can use the following endpoints to retrieve information about role constraints.

Operation	Endpoint	Description
GET	`/role-constraints`	Retrieve a list of all contact role constraints for the given instance of ClaimCenter
GET	`/role-constraints/{contactRoleId}`	Retrieve information for the given contact role. Note that `contactRoleId` is the contact role's code, such as `reporter`.

These are metadata endpoints. They return information about the configuration of the given instance of ClaimCenter, not about any of its business resources.

Role constraint example: Doctor

This is a portion of the payload when GET `/role-constraints/doctor` is executed on the base configuration:

```
{
    "data": {
            "schemaConstraints": [
                {
                    "constraints": [
                        {
                            "constraintType": "ZeroToMore"
                        }
                    ],
                    "schema": "Claim"
                },
                {
                    "constraints": [
                        {
                            "constraintType": "ZeroToMore"
                        }
                    ],
                    "schema": "Exposure"
                }
            ],
            "subtype": "Doctor"
        },
```

From this payload, you can determine the following about `doctor`:

- It can be used as a role for a ClaimContact that is associated with a claim.
  - There can be any number of doctors on an claim, including 0.
- It can be used as a role for a ClaimContact that is associated with an exposure.
  - There can be any number of doctors on an exposure, including 0.
- The role of *doctor* can only be used on ClaimContacts whose associated contact has a subtype of `Doctor` (or a child subtype of `Doctor`).

Role constraint example: Reporter

This is a portion of the payload when GET `/role-constraints/reporter` is executed on the base configuration:

```
{
    "data": {
        "attributes": {
            "schemaConstraints": [
                {
                    "constraints": [
                        {
                            "constraintType": "Exclusive"
                        },
                        {
                            "constraintType": "Required"
                        }
                    ],
                    "schema": "Claim"
                },
                {
                    "constraints": [
                        {
                            "constraintType": "ZeroToMore"
                        }
                    ],
                    "schema": "Exposure"
                }
            ]
```

From this payload, you can determine the following about *reporter*:

- It can be used as a role for a ClaimContact that is associated with a claim.
  - The role is "exclusive". (There can be at most one ClaimContact on a Claim with this role.)
  - The role is "required". (There must be at least one ClaimContact on a Claim with this role.)
  - Taken together, these two constraints mean there must be exactly one reporter on a Claim.
- It can be used as a role for a ClaimContact that is associated with an Exposure.
  - There can be any number of reporters on an exposure, including 0.
- There is no subtype restriction. Therefore, the role of *reporter* can be used with any ClaimContact, regardless of the subtype of its associated contact.

chapter 15

Working with incidents

This topic provides a high-level overview of incidents, discussing both what they are and how to work with them through the system APIs.

For a more detailed discussion of the business functionality of incidents, refer to the *Application Guide*. For a more detailed discussion of the configuration of incidents, refer to the *Configuration Guide*.

Overview of incidents in ClaimCenter

The following section provides an overview of incident behavior in ClaimCenter.

What is an incident?

An *incident* is a collection of information about an item that was lost or damaged, such as:

- A vehicle
- A property (such as a house or a fence)
- A person suffering one or more injuries

For example, a vehicle incident can store the following information:

- Where was the point of collision?
- Who was the driver?
- What is the severity of the damage?
- Were the airbags deployed?
- Is the vehicle so damaged that it is considered a "total loss"?

Incident subtypes

Incidents are subtyped. The following is a portion of the incident hierarchy.

- **Injury Incident** - An injury suffered by a claimant
- **Property Incident** - A property (such as a house, fence, vehicle, or expenses incurred from lose of use)
  - **Fixed Property Incident** - A fixed property, such as a house, shed, or fence
    - **Dwelling Incident** - A fixed property use for dwelling, such as a house
  - **Living Expenses Incident** - Expenses incurred from the lose of use of a dwelling
  - **Mobile Property Incident** - A mobile property, such as baggage or a vehicle
    - **Vehicle Incident** - A vehicle

Incidents and policy types

Every claim is attached to a policy with a specific policy type, such as `PersonalAuto` or `HOPHomeowners`. Every incident type is indirectly associated with one or more of these policy types. Incidents of a given type can be created only on claims with a matching policy type.

For example, vehicle incidents are associated with four policy types:

- `BusinessAuto`
- `Businessowners`
- `PersonalAuto`
- `PersonalTravel`

You can create vehicle incidents on a claim whose policy type is one of these types. You cannot create vehicle incidents on a claim whose policy type is not one of these types.

The association between incident type and policy type occurs in the Line of Business typelists. For more information on the Line of Business typelists, see the *Configuration Guide*.

Incidents and ClaimContacts

Incidents can have ClaimContacts associated with them. When a ClaimContact is associated with an incident, the ClaimContact also has a role defining the relationship.

For example, with a vehicle incident, a ClaimContact could be:

- An *owner*
- A *driver*
- A *passenger*

For more information on ClaimContacts, see "Working with ClaimContacts" on page 153.

Incidents and exposures

Every exposure is associated with an incident. You cannot create an exposure without an incident. For more information on exposures, see "Working with exposures" on page 171.

Overview of incidents in the system APIs

The system APIs support the following incident resources:

- Dwelling incident
- Fixed property incident
- Injury incident
- Living expense incident
- Vehicle incident

For each incident type, there are typically five endpoints as described in the following table:

Operation	Endpoint	For the given claim...
GET	`/claims/{claimId}/`*`incidentType`*	Query for all incidents of *incidentType*
POST	`/claims/{claimId}/`*`incidentType`*	Create a new incident whose type is *incidentType*
GET	`/claims/{claimId}/`*`incidentType`*`/{incidentId}`	Query for the given incident
PATCH	`/claims/{claimId}/`*`incidentType`*`/{incidentId}`	Update the given incident
DELETE	`/claims/{claimId}/`*`incidentType`*`/{incidentId}`	Delete the given incident

For example, the following endpoints interact with vehicle incidents:

- GET `/claims/{claimId}/vehicle-incidents`

- POST `/claims/{claimId}/vehicle-incidents`
- GET `/claims/{claimId}/vehicle-incidents/{incidentId}`
- PATCH `/claims/{claimId}/vehicle-incidents/{incidentId}`
- DELETE `/claims/{claimId}/vehicle-incidents/{incidentId}`

Primary child objects

Most types of incidents include an inlined "primary" child object that stores information that is inherent to the damaged thing from before it was damaged. For example:

- Dwelling incidents and fixed property incidents have a `location` object.
  - This stores information inherent to the location, such as `address`.
- Injury incidents have an `injuredPerson` object.
  - This stores information inherent to the injured person, such as `firstName` and `lastName`.
- Vehicle incidents have a `vehicle` object.
  - This stores information inherent to the vehicle, such as `make`, `model`, and `licenseplate`.

Information about the damage (such as the damage description or severity) are stored on the incident, but not as part of this child object. For example, vehicle incidents have an `airbagsdeployed` field. This field is directly on the vehicle incident itself, not on the `vehicle` child object.

Incidents and risk units

There are two resources that can have "primary" child objects:

- *Incidents*, which are things that are lost or damaged (whether or not they were covered on the policy).
- *Risk units*, which are things covered on the policy associated with the claim (whether or not they have been lost or damaged).

A "primary" child object could be associated with only a risk unit, only an incident, or both a risk unit and an incident. Consider the following examples with vehicles and personal auto policies:

- A vehicle owned by the policyholder that was not damaged.
  - This appears on the ClaimCenter copy of the policy as part of a vehicle risk unit.
  - But because it was not damaged, there is no vehicle incident for this vehicle.
- A vehicle owned by a third party that was damaged.
  - This does not appear on the ClaimCenter copy of the policy. (It may be covered on the third party's policy, but it is not covered on the policyholder's policy.) For this vehicle, there is no vehicle risk unit.
  - But because it was damaged, there is a vehicle incident for this vehicle.
- A vehicle owned by the policyholder that was damaged.
  - This appears on the ClaimCenter copy of the policy as part of a vehicle risk unit.
  - This also appears as part of a vehicle incident because it was damaged.

Incompatible incident types

You cannot create an incident whose type is incompatible with the policy type. For example, you cannot create a dwelling incident on a claim associated with a personal auto policy. If you attempt to do so, the system APIs respond with an error message similar to the following:

```
{
        "status": 404,
        "errorCode": "gw.api.rest.exceptions.NotFoundException",
        "userMessage": "No resource was found at path /claim/v1/claims/cc:34/dwelling-incidents"
}
```

For a given policy type, some insurers may have a business requirement that involves creating incidents that are incompatible with that policy type in the base configuration. For example, an insurer may have a business requirement to create fixed property incidents for inland marine policies, even though, in the base configuration, fixed property incidents are incompatible with inland marine policies. To implement this business requirement, the insurer must

configure the ClaimCenter LOB typelists to make the incident type compatible with the policy type. For more information, see the *Configuration Guide*.

Creating incidents

In the base configuration, there are no required fields for creating any type of incident. All fields are optional.

The following sections provide additional information and examples of the various types of incidents you can create through the system APIs.

Dwelling incidents

A *dwelling incident* is an object that captures loss information about a place where people live.

In the base configuration, dwelling incidents can be used with policies of the following type:

- HOPHomeowners

A dwelling incident typically includes a primary child object called `location`, with fields that describe inherent qualities of the dwelling's location, such as `address`. The dwelling incident also contains additional information specific to the loss, such as `damagedAreaSize` and `severity`. You do not have to specify `location` when you create a dwelling incident. If you do want to specify `location`, you can either:

- Specify an existing location on the policy by providing its `policySystemId`.
- Specify an existing location on the claim by providing its ClaimCenter `id`.
- Create a new location by providing its attributes inline.

Unlike some other child objects, a `location` cannot be created as a referenced resource in the `included` section and then specified by `refid`. A new location must be created as an inlined resources.

Example of creating a typical dwelling incident

In this example, the dwelling incident's `location` is included, and it is specified by `policySystemId`.

```
POST /claims/{claimId}/dwelling-incidents

{
    "data": {
        "attributes": {
            "description": "water from heavy rains leaked through the roof damaging walls and floor.",
            "location" : {
              "policySystemId" : "pcdwl:0001-1"
              } ,
         "yearsInHome" : 7
        }
    }
}
```

Fixed property incidents

A *fixed property incident* is an object that captures loss information about a fixed piece of property (such as a building) or a permanent structure (such as a fence or a fountain).

In the base configuration, fixed property incidents can be used for a large range of policy types. This includes:

- Business auto
- Businessowners
- Commercial package
- Commercial property
- Personal auto

This list is not exhaustive. For a complete list of policy types that are compatible with fixed property incidents, refer to the `Incident` typelist in Studio.

A fixed property incident typically includes a primary child object called `location`, with fields that describe inherent qualities of the property's location, such as `address`. The fixed property incident also contains additional information specific to the loss, such as `lossparty` and `severity`. You do not have to specify `location` when you create a fixed property incident. If you do want to specify `location`, you can either:

- Specify an existing location on the policy by providing its `policySystemId`.
- Specify an existing location on the claim by providing its ClaimCenter `id`.
- Create a new location by providing its attributes inline.

Unlike some other child objects, a `location` cannot be created as a referenced resource in the `included` section and then specified by `refid`. A new location must be created as an inlined resources.

Example of creating a typical fixed property incident

In this example, the dwelling incident's `location` is included, and it is created as an inlined resource.

```
POST /claims/{claimId}/fixed-property-incidents

{
    "data": {
        "attributes": {
            "location": {
                "address": {
                    "addressLine1": "1313 Monroe Lane",
                    "city": "Pomona",
                    "country": "US",
                    "state": {
                        "code": "CA"
                    }
                },
                "primaryLocation": false
            },
            "severity" : {
              "code" : "major-prop"
            }
        }
    }
}
```

Injury incidents

An *injury incident* is an object that captures loss information about a single injury that a claimant suffered.

In the base configuration, injury incidents can be used for a large range of policy types. This includes:

- Business auto
- Businessowners
- Commercial package
- General liability
- Personal auto

This list is not exhaustive. For a complete list of policy types that are compatible with fixed property incidents, refer to the `Incident` typelist in Studio.

An injury incident typically includes a primary child object called `injuredPerson`, with fields that describe inherent qualities of the person, such as `firstName` and `lastName`. The injury incident also contains additional information specific to the injury, such as `ambulenceused`, `primaryDoctor`, and `treatmentType`. You do not have to specify `injuredPerson` when creating an injury incident. If you do want to specify `injuredPerson`, you can either:

- Specify an existing ClaimContact on the policy by providing its `policySystemId`.
- Specify an existing ClaimContact on the claim by providing its ClaimCenter `id`.
- Create a new ClaimContact in the `included` section and reference that ClaimContact by `refid`.

Example of creating a typical injury incident

In this example, the injury incident's `injuredPerson` is provided, and it is specified by ClaimCenter `id`.

```
POST /claims/{claimId}/injury-incidents
```

```
{
    "data": {
        "attributes": {
            "bodyParts": [
                {
                    "primaryBodyPart": {
                        "code": "head"
                    }
                }
            ],
            "description": "Potential vision loss",
            "detailedInjuryType": {
                "code": "58"
            },
            "generalInjuryType": {
                "code": "specific"
            },
            "injuredPerson": {
                "id": "cc:102"
            },
            "lossParty": {
                "code": "third_party"
            },
            "lostWages": true,
            "severity": {
                "code": "major-injury"
            },
            "treatmentType": {
                "code": "hospital"
            }
        }
    }
}
```

Living expenses incidents

A *living expenses incident* is an object that captures loss information about expenses incurred as a result of the loss of use of a property. (For example, staying in a hotel while a damaged home is repaired.)

In the base configuration, living expenses incidents can be used with policies of the following type:

- HOPHomeowners

Unlike other types of incidents, a living expense incident does not make use of a primary child object.

Example of creating a typical living expense incident

In this example, a living expense incident is created. There is no primary child object to reference or create.

```
POST /claims/{claimId}/living-expenses-incidents

{
    "data": {
        "attributes": {
            "description": "7-day hotel stay during flood damage repair",
        "lossParty" : {
          "code" : "insured"
        },
        "startDate" : "2020-08-31T07:00:00.000Z"
        }
    }
}
```

Vehicle incidents

A *vehicle incident* is an object that captures loss information about a vehicle.

In the base configuration, vehicle incidents can be used with policies of the following type:

- Business auto
- Businessowners
- Personal auto
- Personal travel

A vehicle incident typically includes a primary child object called `vehicle`, with fields that describe inherent qualities of the vehicle, such as `make`, `model`, and `licenseplate`. The vehicle incident also contains additional information

specific to the loss, such as `airbagsdeployed`, `collisionpoint`, and `driver`. You do not have to specify `vehicle` when creating a vehicle incident. If you do want to specify vehicle, you can either:

- Specify an existing vehicle on the policy by providing its `policySystemId`.
- Specify an existing vehicle risk unit on the claim by providing its ClaimCenter `id`.
- Create a new vehicle by providing its attributes inline.

Unlike some other child objects, a `vehicle` cannot be created as a referenced resource in the `included` section and then specified by `refid`. A new vehicle must be created as an inlined resources.

Example of creating a typical vehicle incident

In this example, the vehicle incident's `vehicle` is provided, and it is created as an inlined resource.

```
POST /claims/{claimId}/vehicle-incidents

{
    "data": {
        "attributes": {
            "collisionPoint": {
                "code": "front"
            },
            "damageDescription": "Damage to bumper and front panels",
            "driver": {
                "id": "cc:102"
            },
            "severity": {
                "code": "moderate-auto"
            },
            "vehicle": {
                "licensePlate": "7FDG745",
                "make": "Mercury",
                "model": "Sable",
                "state": {
                    "code": "CA",
                    "name": "California"
                },
                "vin": "6GYF54637HD645370",
                "year": 1993
            }
        }
    }
}
```

Summary of incident types

The following chart summarizes the types of incidents, whether the schema includes a primary child object, and how that child object can be specified.

Incident	Primary child object	Specify by policySystemId?	Specify by ClaimCenter id?	Create in included section and specify by refid?	Create inline?
Dwelling	`location`	yes	yes	no	yes
Fixed Property	`location`	yes	yes	no	yes
Injury	`injuredPerson`	yes	yes	yes	no
Living Expense	(none)	(not applicable)	(not applicable)	(not applicable)	(not applicable)
Vehicle	`vehicle`	yes	yes	no	yes

chapter 16

Working with exposures

This topic provides a high-level overview of exposures, discussing both what they are and how to work with them through the system APIs.

For a more detailed discussion of the business functionality of exposures, see the *Application Guide*. For a more detailed discussion of the configuration of exposures, see the *Configuration Guide*.

Overview of exposures in ClaimCenter

The following section provides an overview of exposure behavior in ClaimCenter.

What is an exposure?

An *exposure* is an object associated with a claim which is used to track a potential payment or a set of related potential payments. Every exposure is linked to one coverage (where the money is "coming from") and one claimant (where the money is "going to").

For example, suppose that Ray Newton has a personal auto policy. He informs the insurer that, while driving his Toyota, he hit Robert Farley's Honda and damaged both cars. Robert Farley also suffered a neck injury. The associated claim would have three exposures to track these potential payments:

	Claimant		Coverage	
A potential payment to...	Ray Newton	...from the policy's...	collision coverage	...to pay for repairs to Ray's car.
A potential payment to...	Robert Farley	...from the policy's...	third-party property damage coverage	...to pay for repairs to Robert's car.
A potential payment to...	Robert Farley	...from the policy's...	third-party bodily damage coverage	...to pay for Robert's medical bills to treat his neck injury.

Some exposures result in a single payment. This is likely to be true for the first and second exposure in the previous example. Typically, repairs to a vehicle can be covered in a single payment. Other exposures manage a set of related payments. This could be true for the third exposure in the previous example. Medical treatment might occur over an extended period of time, and multiple payments may be needed, one for each treatment.

Exposures and coverages

Every exposure is directly linked to a coverage type. A *coverage type* is a type of loss specified on a policy. For example, for personal auto policies, `PACollisionCov` and `PALiabilityCov` are two coverage types. `PACollisionCov`

covers damage to a vehicle owned by the insured. `PALiabilityCov` covers damages to vehicles owned by a third party where the damage was caused by the insured.

Every exposure is indirectly linked to an exposure type. An *exposure type* is a set of information to gather for an exposure. For example, `VehicleDamage` is an exposure type. It consists of information to gather about a damaged vehicle, such as where on the vehicle is the damage, who was the driver, and were the airbags deployed.

Two exposures can be linked to the same exposure type, even if the coverages are different. For example, `PACollisionCov` and `PALiabilityCov` are different coverages, but they can both involve damaged vehicles. The same set of information needs to be gathered about a damaged vehicle, regardless of which coverage is involved. Therefore, the two coverages are mapped to a single exposure type - the `VehicleDamage` exposure type. This exposure type is used to determine the information to gather during the claims process.

Some coverages link to multiple exposure types. Therefore, ClaimCenter does not link coverage types directly to exposure types. Instead, ClaimCenter links them through coverage subtypes. A *coverage subtype* is a value that links a coverage type to an exposure type. For example:

- `PACollisionCov` is a coverage subtype that links the `PACollisionCov` coverage to the `VehicleDamage` exposure type. (In this case, the coverage type and coverage subtype have the same name.)
- `PALiabilityCov_vd` is a coverage subtype that links the `PALiabilityCov` coverage to the `VehicleDamage` exposure type.

When you create an exposure, you must specify both its coverage and coverage subtype.

Exposures and reserve lines

When an exposure is created, ClaimCenter also creates a reserve line for the exposure. A *reserve line* is an amount of money set aside for expected payments related to a given exposure. Insurers are often legally required to create reserve lines to ensure that they maintain financial solvency.

Reserve lines can be created:

- Automatically by business rules
- Manually by adjusters

When a payment is made from an exposure, the money comes from this reserve line.

Exposures and validation levels

Just as is the case with claims, during an exposure's lifecycle, an exposure passes through one or more levels of maturity. Within ClaimCenter, these are called *validation levels*. The base configuration comes with the following levels, which are common to both claims and exposures:

- **Load and save** - The claim/exposure has enough information to be saved to the database.
- **New loss completion** - The claim/exposure has enough information to be assigned to an adjuster.
- **Valid for ISO** - The claim/exposure has enough information to be filed with ISO. (ISO is a national database used in the United States to verify that the same loss is not being filed with multiple insurers.)
- **Send to external (systems)** - The claim/exposure has enough information to send information about it to external systems within the insurer, such as a Policy Administration System that may be trying to assess policy renewal rates.
- **Ability to pay** - The claim/exposure has enough information such that payments can be written for it.

  Note: In the base configuration, the "load and save" level applies only to claims and exposures that are being imported through the ClaimCenter SOAP-based `ClaimAPI` API. Draft exposures submitted through the system APIs do not need to pass any level. In order for a draft claim to be promoted to an open claim, the draft claim and all of its exposures must pass both the "load and save" level and the "new loss completion" level. For more information, see “Executing FNOL” on page 117.

A exposure's validation level is determined and enforced by a set of exposure validation rules. Whenever a change is made to an exposure, the validation rules determine if the exposure can be advanced to a later stage of validation. The validation rules also prevent an exposure from moving backwards to a lower level of validation. For more information on validation rules, see the *Rules Guide*.

A claim and its exposures are not necessarily always at the same validation level. For example, suppose there is a claim with two exposures. It is possible for the claim to be at "send to external" while one of the exposures is at "new loss completion" and other is at "ability to pay".

In order to make a payment, both the claim and the exposure from which the payment is coming must be at "ability to pay". If a claim has multiple exposures, and the claim and one of the exposures are at "ability to pay", you can make payments from that one exposure, even though the other exposures are not yet at "ability to pay".

Exposures and ClaimContacts

Every exposure has at least one ClaimContact - the claimant. Exposures can have additional ClaimContacts associated with them.

For more information on ClaimContacts, see “Working with ClaimContacts” on page 153.

Exposures and incidents

Every exposure is associated with an incident. An *incident* is a collection of information that typically represents an item that was lost or damaged, such as:

- A vehicle
- A property (such as a house or a fence)
- A person suffering one or more injuries

You cannot create an exposure without an incident. For more information on incidents, see “Working with incidents” on page 163.

Creating exposures

The POST `/claims/{claimId}/exposures` endpoint can be used to create new exposures.

Minimum creation criteria

In order for an exposure to be assignable, the exposure must have the following:

- A coverage and coverage subtype
- A claimant
- An incident

The following JSON skeleton summarizes these components as they appear in a POST `/exposures` request payload.

```
{
    "data": {
        "attributes": {
            "primaryCoverage": {
                "code": "..."
            },
            "coverageSubtype": {
                "code": "..."
            },
            "claimant": {
                "id" / "refid" / "policySystemId" : "..."
            },
            "...Incident": {
                "id" / "refid" : "..."
            }
        }
    },
    "included": {
        "...Incident": ...
        "ClaimContact": ...
    }
}
```

Note the following:

- The coverage and coverage subtype are identified by typecodes from the `CoverageType` and `CoverageSubtype` typelists.

- The claimant can be referenced by any of the following:
  - `id`, if the claimant already exists on the claim
  - `refid`, if the claimant is being created in the same payload as the exposure
  - `policySystemId`, if the claimant is listed on the policy
- The exposure must reference an incident. The incident type varies based on the coverage.
- The incident can be referenced by any of the following:
  - `id`, if the incident already exists on the claim
  - `refid`, if the incident is being created in the same payload as the exposure

Building an exposure payload

To build an exposure payload, you must:

1. Identify the coverage type
2. Identify the coverage subtype
3. Create or identify the claimant
4. Create or identify the incident

Note that each item in the previous list does not necessarily map to a single block of code. When it is time to create the exposure, the caller application may already have the required information. Also, the caller application may be able to query ClaimCenter for multiple pieces of information in a single call.

Example creation of an exposure payload

The following sections provide an example of creating the payload for a new exposure. This exposure will be for claim 235-53-373906 in the sample data, which is assigned to Betty Baker. The ID for this claim is demo_sample:8037. The claim's policy has two vehicles: a Honda Civic and a Ford Explorer. The new exposure will be for the Honda Civic using the collision coverage. The claimant is Allen Robertson, an additional insured on the policy.

All of the calls assume the instance of ClaimCenter is on the local machine.

Step 1: Identify the coverage type

If the caller application does not know the coverage type, it can use the GET `/claims/{claimId}/policy` endpoint to determine the coverages attached to the Honda Civic.

Request to determine the coverage type

```
GET  http://localhost:8080/cc/rest/claim/v1/claims/demo_sample:8037/policy/vehicle-risk-units?fields=*all
```

Response payload (snippet)

```
"RUNumber": 1,
  "coverages": [
    {
      ...
      "coverageType": {
        "code": "PACollisionCov",
        "name": "Collision"
        }
       ...
    }
  ],
  "id": "cc:9",
  "vehicle": {
    ...
    "id": "demo_sample:4",
    "make": "Honda",
    "model": "Civic"
    ...
```

Exposure request payload (first part)

Based on the previous query, the first part of the POST `/exposures` request payload looks like this:

```
{
    "data": {
        "attributes": {
            "primaryCoverage": {
                "code": "PACollisionCov"
            },
            ...
```

Step 2: Identify the coverage subtype

The set of ClaimCenter coverage types and coverage subtypes change infrequently. To reduce the number of calls, you may want to store the possible coverage types and coverage subtypes locally with the caller application.

Either during development or at the time of exposure creation, the caller application can determine the coverage subtypes for a given coverage by executing the Common API's GET `/typelists` endpoint. To limit the response to only the coverage subtypes for a given coverage type, the call can filter the `CoverageSubtype` typelist using the exposure's `CoverageType` (such as `PACollisionCov`).

For more information on the Common API's GET `/typelists` endpoint, see “The /typelists endpoints” on page 31.

Request to determine the coverage subtype

```
GET  http://localhost:8080/cc/rest/common/v1/typelists/CoverageSubtype
        ?typekeyFilter=category:cn:CoverageType.PACollisionCov
```

Response payload (snippet)

```
"description": "Subtype of coverage, filtered by CoverageType",
"name": "CoverageSubtype",
"typeKeys": [
  {
    "code": "PACollisionCov",
    "description": "Collision",
    "name": "Collision",
    "priority": -1
  }
]
```

Exposure request payload (first two parts)

Based on the previous query, the first and second part of the POST `/exposures` request payload looks like this:

```
{
    "data": {
        "attributes": {
            "primaryCoverage": {
                "code": "PACollisionCov"
            },
            "coverageSubtype": {
                "code": "PACollisionCov"
            },
            ...
```

Step 3: Create or identify the claimant

If the claimant exists on the policy, the payload can identify the claimant by its `policySystemId`. If necessary, the caller application can query the Policy Administration System for the `policySystemId`.

If the claimant does not already exist, the caller application can create a new ClaimContact in the POST `/exposures` request payload and then reference that ClaimContact using a `refid`. This technique is referred to as request inclusion. For more information, see “Request inclusion” on page 80.

If the claimant exists in ClaimCenter, the payload can identify the claimant by its `id`. If necessary, the caller application can query ClaimCenter for the `id`.

Note: If the claimant already exists in ClaimCenter, always reference the existing ClaimContact and use the `id` field. Do not create an additional ClaimContact through the use of the `refid` field. Creating the same logical ClaimContact twice results in duplicate data. This can complicate the processing of the claim.

In this example, the claimant has already been copied over to ClaimCenter. Therefore, the payload will identify the claimant by ClaimCenter `id`.

Request to determine the claimant ID

```
GET  http://localhost:8080/cc/rest/claim/v1/claims/demo_sample:8037/contacts
```

Response payload (snippet)

```
{
  "attributes": {
    ...
    "displayName": "Allen Robertson",
    "id": "cc:32",
    ...
    },
```

Exposure request payload (first three parts)

Based on the previous query, the first three parts of the POST `/exposures` request payload looks like this:

```
{
    "data": {
        "attributes": {
            "primaryCoverage": {
                "code": "PACollisionCov"
            },
            "coverageSubtype": {
                "code": "PACollisionCov"
            },
            "claimant": {
                "id": "cc:32"
            },
            ...
```

Step 4: Create or identify the incident

An *incident* is a collection of information that typically represents an item that was lost or damaged. Incidents may reference objects on a policy. (For example, a vehicle incident can reference a vehicle on the policy.) But, incidents never appear on policies. Incidents exist solely in ClaimCenter.

If the incident does not already exist, the caller application can create a new incident in the POST `/exposures` request payload and then reference that incident using a `refid`. This technique is referred to as request inclusion. For more information, see “Request inclusion” on page 80. Depending on the incident type, the new incident can reference a child object (a location, injured person, or vehicle). This child object could be on the policy, in ClaimCenter, or also created in the POST `/exposures` request payload.

If the incident already exists in ClaimCenter, the caller application can reference it by its `id`.

Note: If the incident already exists in ClaimCenter, always reference the existing incident and use the `id` field. Do not create an additional incident through the use of the `refid` field. Creating the same logical incident twice results in duplicate data. This can complicate the processing of the claim.

In this example, the incident does not exist and must be created. But, it will reference a vehicle that is already on the policy. The ID for this vehicle was already retrieved in the first step when the coverage type was identified. The ID is demo_sample:4. There is no need for an additional request to retrieve additional vehicle information.

Exposure request payload (complete)

The complete POST `/exposures` request payload looks like this:

```
{
    "data": {
```

```
        "attributes": {
            "primaryCoverage": {
                "code": "PACollisionCov"
            },
            "coverageSubtype": {
                "code": "PACollisionCov"
            },
            "claimant": {
                "id": "cc:32"
            },
            "vehicleIncident": {
                "refid": "newVehicleIncident"
            }
        }
    },
    "included": {
        "VehicleIncident": [
            {
              "attributes": {
                "vehicle": {
                  "id": "demo_sample:4"
                }
              },
            "refid": "newVehicleIncident",
            "method": "post",
            "uri": "/claim/v1/claims/demo_sample:8037/vehicle-incidents"
            }
        ]
    }
}
```

Querying for and modifying exposures

You can query for exposures using:

- GET `/claims/{claimId}/exposures`
- GET `/claims/{claimId}/exposures/{exposureId}`

You can modify an exposure using:

- PATCH `/claims/{claimId}/exposures/{exposureId}`

Assigning exposures

Every exposure is assigned to a group and a user in that group. The assigned user has the primary responsible for managing the exposure.

Most exposures are assigned to the same user and group that owns the claim. However, exposures occasionally require special expertise that require assignment to a different user and group than that of the claim. For example, suppose there is a personal auto claim that includes three exposures: two exposures for damaged vehicles and one exposure for medical payments related to a fatal injury. The claim and the vehicle exposures are routine and can all be assigned to the same group and user. But the injury exposure is likely to involve a significant payment or litigation. This exposure is assigned to a group and user that specialize in fatalities.

When you create an exposure through the system APIs, ClaimCenter automatically executes the exposure assignment rules to initially assign the exposure to a group and user. You can use the POST `/claims/{claimId}/exposures/{exposureId}/assign` endpoint to reassign the exposure as needed.

> **Note:** The functionality for assigning exposures is a subset of the functionality for assigning activities. All assignment options that are applicable to both activities and exposures have the same behavior.

Assignment options

An exposure 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 ClaimCenter uses assignment rules to select a user in that group)
- To the claim owner
- By re-running the exposure assignment rules

- This can be appropriate if you have modified the exposure since the last time assignment rules were run and the modification might affect who the exposure would be assigned to.

The root resources for the `/exposures/{exposureId}/assign` endpoints is `ExposureAssignee`. This resource specifies assignment criteria. The schema has the following fields:

Field	Type	Description
`autoAssign`	Boolean	Whether to assign the exposure using assignment rules
`claimOwner`	Boolean	Whether to assign the exposure to the claim owner
`groupId`	string	The ID of the group to assign the exposure to
`userId`	string	The ID of the user to assign the exposure to

The `ExposureAssignee` resource cannot be empty. It must specify a single logical assignment option (group and user, group only, claim owner, or automatic assignment).

For more information on how assignment rules execute assignment, see the *Rules Guide*.

Assignment example - Assigning to a specific group (and user)

The following assigns exposure cc:48 (on claim cc:34) to group demo_sample:31 (Auto1 - TeamA) and user demo_sample:2 (Sue Smith).

```
POST /claim/v1/claims/cc:34/exposures/cc:48/assign

{
  "data": {
    "attributes" : {
      "groupId" : "demo_sample:31",
      "userId" : "demo_sample:2"
    }
  }
}
```

The following assigns exposure cc:48 (on claim cc:34) to group demo_sample:31 (Auto1 - TeamA). Because no user has been specified, ClaimCenter will execute assignment rules to assign the exposure to a user in group demo-sample:31.

```
POST /claim/v1/claims/cc:34/exposures/cc:48/assign

{
  "data": {
    "attributes" : {
      "groupId" : "demo_sample:31"
    }
  }
}
```

Note that there is currently no endpoint that returns groups or group IDs. To assign exposures to a specific group, the caller application must determine the group ID using some method other than a groups system API.

Assignment example - Assigning to the claim owner

The following assigns exposure cc:48 (from claim cc:34) to the group and user that owns the parent claim.

```
POST /claim/v1/claims/cc:34/exposures/cc:48/assign

{
  "data": {
    "attributes" : {
        "claimOwner" : true
    }
  }
}
```

Assignment example - Using automated assignment

The following assigns exposure cc:48 (from claim cc:34) using automated assignment rules.

```
POST /claim/v1/claims/cc:34/exposures/cc:48/assign

{
  "data": {
    "attributes": {
        "autoAssign" : true
    }
  }
}
```

Additional exposure endpoints

The system APIs provide additional endpoints to interact with exposures.

Deleting draft exposures

During the FNOL process, the claim passes through two states: draft and open.

- A *draft claim* is a claim that has been saved to the ClaimCenter database, but there is not yet enough information for the claim to enter the adjudication process. Draft claims are not assigned to any user.
- An *open claim* is a claim that has been saved to the ClaimCenter database with enough information to enter the adjudication process. Once a claim becomes open, it is assigned to an adjuster.

A *draft exposure* is an exposure on a draft claim. While a claim is in a draft state, you can delete any exposures created on the claim using the DELETE `/claims/{claimId}/exposures/{exposureId}` endpoint.

Executing a POST `/claims/{claimId}/submit` on a draft claim promotes the claim and all of its exposures to open status. Once a claim has been submitted, you can no longer delete its exposures.

Validating exposures

Similar to claim validation, the POST `/claim/{claimId}/exposures/{exposureId}/validate` endpoint returns the validation level for the given exposure. It can also be used to determine what conditions must be met for the exposure to advance to a given validation level.

Checking an exposure's validation level can be useful in the following situations:

- You want to determine whether or not the exposure has enough information to be assigned to an adjuster. You can use the `/validate` endpoint to determine if the exposure is at or beyond the "new loss completion" level. If the exposure is below the "new loss completion" level, the payload identifies the conditions needed to reach "new loss completion".
- You want to execute a payment for an exposure. You can use the `/validate` endpoint to determine if the exposure is at the "ability to pay" level. If the exposure is below the "ability to pay" level, the payload identifies the conditions needed to reach "ability to pay".

For more information on validation through system APIs, see “Validating claims” on page 150.

Closing exposures

During an exposure's life cycle, the exposure's status typically moves from draft to open to closed. An exposure is closed to indicate that no further payments are expected to be made from the exposure.

In the base configuration, ClaimCenter automatically closes an exposure when a "final payment" is made from the exposure's reserve line. (A "final payment" is a payment whose payment type is *final*, as opposed to a payment whose type is *partial*.)

You can also close exposures through the system APIs at any time. Broadly speaking, once an exposure is closed, payments can no longer be made from that exposure. However, there are exceptions to this rule. For more information, see the *Application Guide*.

To close an exposure, use the POST `/claim/{claimId}/exposures/{exposureId}/close` endpoint.

chapter 17

Working with service requests

A claim can have one or more service requests. This topic provides a high-level overview of service requests, discussing both what they are and how to work with them through the system APIs.

For a more detailed discussion of the business functionality of service requests, refer to the *Application Guide*. For a more detailed discussion of the configuration of service requests, refer to the *Configuration Guide*.

Overview of service requests in ClaimCenter

The following section provides an overview of service request behavior in ClaimCenter.

What is a service request?

A *service* is an action performed by a vendor to address a loss associated with a claim. For example:

- For a damaged car, services could include towing, auto body repair, and replacement car rental.
- For a damaged house, services could include plumbing and roof repair.
- For an injury, services could include conducting an examination, taking an x-ray, or performing surgery.

A *service request* is a collection of services managed by ClaimCenter for a given claim. Services are grouped into service requests because a single vendor often provides multiple services. When this occurs, it is easier to have the group of service requests associated with a single instruction, a single set of invoices, and a single payment for the related services. The service request provides this grouping.

Components of a service request

A service request includes the following information:

- The vendor (also referred to as the "specialist")
- A service instruction, which specifies:
  - The customer
  - The location where the service is being performed
  - The set of services being performed
- The relevant exposure and incident

Service request kinds

Every service request has a service request kind. A *service request kind* is a business flow that describes the steps to be used to quote, process, and invoice the services. ClaimCenter uses the following service request kinds:

- **Quote Only** - This kind of service request requires only a quote from the vendor.
  - This kind is appropriate when an insurer wishes to compare quotes from multiple vendors before deciding who to assign the work to. (This kind of service request can be promoted to Quote and Service.)
  - In the user interface, this kind of service request is labeled "Quote".
- **Quote and Service** - This kind of service request involves a quote, which is followed by the vendor providing the service and sending invoices for the service.
  - This kind is appropriate when you want the vendor to provide a quote, but you expect to use the assigned vendor regardless of the quote.
  - In the user interface, this kind of service request is labeled "Quote and Perform Service".
- **Service Only** - This kind of service request involves the vendor providing the service (without preparing a quote) and sending invoices for the service.
  - This kind is appropriate when you want the vendor to provide a service and you do not need a quote. This is often used for "flat fee" services, such as car rentals, whose prices do not vary from claim to claim.
  - In the user interface, this kind of service request is labeled "Perform Service".
- **Unmanaged** - This kind of service request is appropriate when you want the vendor to provide a service and you wish to have minimal processing in ClaimCenter. There are no associated quotes. The service request can be invoiced and paid immediately.
  - This kind of service request is for services that are to be performed as quickly as possible, such the repair to a cracked windshield for an "Auto - First and Final" claim.
  - In the user interface, this kind of service request is labeled "Service".

Service type compatibility

Every service is not necessarily compatible with all service request kinds. For example, in the base configuration:

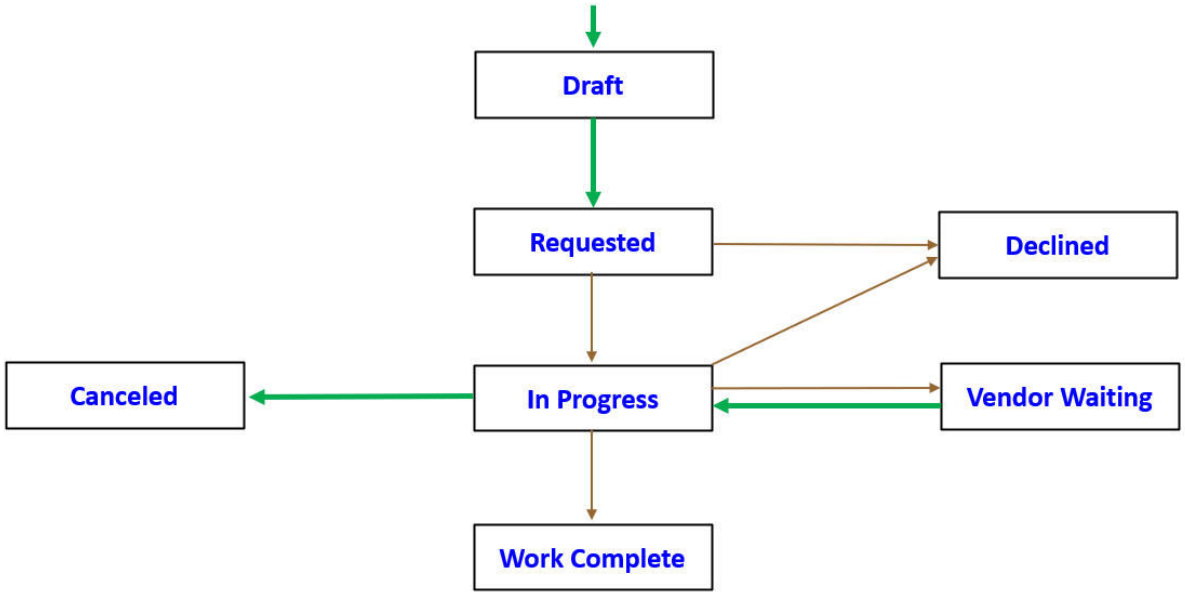
- An "auto appraisal" service can be attached only to a Quote Only service request.
- An "auto towing" service can be attached only to a Quote and Service, Service Only, or Unmanaged service request.

Service type compatibility is configured in the `vendorservicedetails.xml` file. This file is accessible through Studio.

The service request lifecycle

The lifecycle of a service request involves several stages. The current stage is listed in the service request's **Progress** field. In some cases, a service request advances to the next stage because of activity completed by ClaimCenter or a ClaimCenter user. In other cases, a service request advances to the next stage because of activity completed by the vendor.

The following diagram identifies the stages in the lifecycle. Each rectangle is a stage. Thick green arrows lead to stages that are typically reached because of ClaimCenter activity. Thin brown arrows lead to stages that are typically reached because of vendor activity.



A service request that is fully executed goes through the following stages:

1. **Draft**
   - The service request has been created in ClaimCenter but not yet submitted to the vendor.
2. **Requested**
   - The service request has been submitted to the vendor, but not yet accepted.
3. **In Progress**
   - The vendor has accepted the service request and started the work.
   - For a Quote Only service request, the "work" consists of generating a quote.
   - For a Quote and Service service request, the "work" consists of generating a quote and then, once the quote is approved, providing the services.
   - For all other service request kinds, the "work" consists of directly providing the services.
4. **Work Complete**
   - The work (the quote or the set of services) is complete.
   - At this point, the vendor may or may not have submitted invoices for the work.
   - At this point, the invoices may or may not have been paid.

There are additional stages that a service request could reach:

- **Declined**
  - A service request can reach this stage if the vendor decides to not accept a service request. For example, this could happen if the service request is for a rental car and the vendor has no available cars.
  - A service request can reach this stage if the vendor accepts a service request, but then later states they cannot complete it. For example, this could happen if the vendor experiences an unexpected reduction in available mechanics.
- **Canceled**
  - A service request can reach this stage when the insurer decides the service request is no longer needed. For example, the insured could decide to buy a new car instead of fixing the damaged car.
- **Vendor Waiting**
  - A service request can reach this stage when the vendor cannot take action on the service without further input from ClaimCenter. For example, the vendor could have submitted a quote that requires approval from an adjuster.

Invoices for service request

For all service requests whose kind is Quote and Service, Service Only, or Unmanaged, once the work is complete, the vendor typically submits one or more invoices. These invoices are attached to the service request. They are paid using money from one of the claim's reserve lines. Depending on the nature of the service request, they may also require adjuster approval.

Straight-through invoice processing

Straight-through invoice processing is a configurable ClaimCenter behavior in which invoices that meet certain criteria are automatically approved and paid. Straight-through invoice processing is frequently used with Unmanaged service requests, as these service requests are designed to involve minimal processing.

Overview of service requests in the system APIs

Service request APIs and vendor portals

In previous releases of ClaimCenter, service requests were primarily managed by two systems: ClaimCenter and a vendor portal. The *vendor portal* is an application used by a vendor to manage information about service requests from ClaimCenter. In this paradigm:

ClaimCenter is responsible for actions such as:

- Creating the service request
- Submitting the service request to the vendor
- Paying the vendor.

The vendor is responsible for actions such as:

- Accepting the service request
- Quoting the service request
- Submitting invoices for the service request

Cloud API provides a wider range of options for processing service requests. The service request APIs can be used by a vendor portal. But they can also be used by:

- An alternate front-end application for adjusters who specialize in service requests
- A service that submits or pays for vendor invoices in bulk
- A vendor management system that manages service requests for multiple vendors

Thus, the service request functionality exposed by Cloud API is not limited to only the functionality that would be used by vendor portals. Rather, it exposes the service request functionality needed to manage the entire service request process.

Lifecycle management

Cloud API provides a number of endpoints to manage the lifecycle of a service request. This includes both endpoints for actions taken by the insurer (such as submitting a service request) and endpoints for actions taken by the vendor (such as accepting a service request).

As of this release, there are endpoints to advance a service request to most stages in the lifecycle. However, there are currently no endpoints to move a service request to the "Vendor Waiting" status.

Required service request data model

ClaimCenter includes two service requests data models: the "legacy model" and the Core Service Request data model. Each instance of ClaimCenter can use only one of these models. In the base configuration, the Core Service Request data model is enabled by default. In general, Guidewire recommends insurers use the Core Service Request data model.

Note: In order to use the service request system APIs in Cloud API, the Core Service Requests data model must be enabled. Guidewire recommends that insurers who are going into production on this version of ClaimCenter use the Core Service Requests data model. Some insurers may be upgrading from a previous release that offered only the legacy model. If an upgrading customer wishes to use the service request APIs, the insurer must modify their configuration to use the Core Service Requests data model. For more information, refer to the *Upgrade Guide*.

Service request numbers

In addition to a public ID, every service request is assigned a "service request number". By default, this number is included in the response payload for most service request actions (in the `serviceRequestNumber` field). Unlike public IDs, service request numbers are shown in the user interface. During testing, you can use the service request number to match a service request as seen in a system API response with the corresponding service request in the user interface.

Support for each service request kind

If an insurer wants to go into production with this release and requires the ability to create quotes or pay invoices through an integration point, then the insurer must write their own integration points. For more information on service request functionality that may be available in future release, check with your Guidewire account manager or your project manager.

Quote Only and Quote and Service service requests

The following table lists the stages that a Quote Only or Quote and Service service request can advance to through the system APIs. It identifies which system API action advances the service request to the next stage, and the value of the service's **Next Action** column in the ClaimCenter **Services** list.

System API endpoint	Moves Progress to...	Services list's Next Action is...
POST `/service-requests`	Draft	"Submit request"
POST `/{serviceRequestId}/submit`	Requested	"Agree to provide quote"
POST `/{serviceRequestId}/accept`	In Progress	"Add quote"

As of this release, there are no endpoints to create quotes or pay invoices. However, users can create quotes, pay invoices, and take other actions that advance the service request to completion, through the user interface.

Service Only service requests

The following table lists the stages that a Service Only service request can advance to through the system APIs. It identifies which system API action advances the service request to the next stage, and the value of the service's **Next Action** column in the ClaimCenter **Services** list.

System API endpoint	Moves Progress to...	Services list's Next Action is...
POST `/service-requests`	Draft	"Submit request"
POST `/{serviceRequestId}/submit`	Requested	"Agree to perform service"
POST `/{serviceRequestId}/accept`	In Progress	"Finish the work"
POST `/{serviceRequestId}/complete-work`	Work Complete	"Add invoice"
POST `/{serviceRequestId}/invoices`	Work Complete	"Pay invoice"

As of this release, there are no endpoints to pay invoices. However, invoices can be paid through the user interface.

Unmanaged service requests

The following table lists the stages that an Unmanaged service request can advance to through the system APIs. It identifies which system API action advances the service request to the next stage, and the value of the service's **Next Action** column in the ClaimCenter **Services** list.

System API endpoint	Moves Progress to...	Services list's Next Action is...
POST `/service-requests`	Work Complete	"Add invoice"
POST `/{serviceRequestId}/invoices`	Work Complete	"Pay invoice"

As of this release, there are no endpoints to pay invoices. Unmanaged service requests are expected to make use of straight-through invoice processing to automatically approve and pay invoices. However, if required, invoices can be paid through the user interface.

Querying for service requests

The following Claim API endpoints can be used to request information about service requests:

Endpoint	Response
GET `/service-requests`	All service requests By default, the payload in the response includes the ID of each service request and each claim the service request belongs to.
GET `/claims/{claimId}/service-requests`	All service requests for the specified claim
GET `/claims/{claimId}/service-requests/{serviceRequestId}`	The specified service request. Note that in order to get information about a specific service request, you must access the service request through its parent claim.

Creating service requests

To create a service request, use the following endpoint:

- POST `/claims/{claimId}/service-requests`

Once a service request has been created, its **Progress** field is set to Draft.

Minimum creation criteria

At a minimum, a service request must specify:

- The **service request kind**, such as Service Only or Unmanaged (in the `kind` field)
- The **vendor** (in the `specialist` field)
- A **service instruction** (in the `instruction` field), which at a minimum must contain:
  - The **customer** (in the `customer` field)
  - The location **where the service is being performed** (in the `serviceAddress` field)
  - The set of services being performed (in the `services` array)
- A **requested quote completion date**, if the service request is Quote Only or Quote and Service
- A **requested service completion date**, if the service request is Service Only

Additional details on each required field

A service request must specify the **service request kind**. This is specified in the `kind` field, and it must be set to a typecode from the `ServiceRequestKind` typelist, such as:

- `quoteonly`
- `quoteandservice`
- `serviceonly`
- `unmanaged`

A service request must specify the **vendor**. This is specified in the `specialist` field.

- You can specify an existing ClaimContact by listing the `id` field and setting it to the ClaimContact ID.
- You can create a new ClaimContact by listing the `refid` field and specifying a new ClaimContact in the `included` section.

A service request must include a **service instruction**. This is specified in the `instruction` field. At a minimum, a service instruction must have a customer, a location where the service is being performed, and a set of services.

The **customer** is specified in the `customer` field. This must be a reference to an existing or new ClaimContact. You can specify the ClaimContact:

- By `id` (if it already exists in ClaimCenter)
- By `policySystemId` (if it exists in the Policy Administration System)
- By `refid` (if it does not yet exist and is being created in the POST's `included` section.)

The location **where the service is being performed** is specified in the `serviceAddress` field. You can specify the address:

- By `id` (if it already exists in ClaimCenter)
- Inline (if it does not already exist in ClaimCenter)

The **set of services** being performed is specified in the `services` array. Each entry in this array specifies the service's code. The codes come from the `vendorservicetree.xml` file, which you can access through Studio. Each service must be a leaf-level service in the service tree. Also, each service must be compatible with the service request kind. Service compatibility is defined in the `vendorservicedetails.xml` file, which you can also access through Studio.

If the service request's kind is `quoteonly` or `quoteandservice`, you must also specify a **requested quote completion date** in the `requestedQuoteCompletionDate`.

If the service request's kind is `serviceonly`, you must also specify a **requested service completion date** in the `requestedServiceCompletionDate`.

Sample Service Only service request

The following payload shows an example of a minimal Service Only service request for claim 235-53-365889 in the sample data (whose ID is cc:33). The service request will be performed by Joe's Auto Body Shop (ClaimContact cc:16) at 1313 Mockingbird Lane in Arcadia, California, for Robert Farley (ClaimContact cc:13). There is one service to be performed: Salvage (autoothersalvage). The service is requested to be completed by March 3, 2021.

```
POST http://localhost:8080/cc/rest/claim/v1/claims/demo_sample:20/service-requests

{
  "data": {
    "attributes": {
       "kind": {
           "code": "serviceonly"
       },
       "specialist": {
           "id": "cc:16"
       },
       "instruction": {
         "customer": {
           "id": "cc:13"
         },
         "serviceAddress": {
                         "addressLine1": "1313 Mockingbird Lane",
                         "city": "Arcadia",
                         "country": "US",
                         "postalCode": "91006",
                         "state": {
                             "code": "CA",
                             "name": "California"
                           }
         },
         "services": [
           {
             "code": "autoothersalvage"
           }
         ]
      },
      "requestedServiceCompletionDate": "2021-03-19"
    }
  }
}
```

Sample Unmanaged service request

The following payload shows an example of a minimal Unmanaged service request for claim 235-53-365889 in the sample data (whose ID is cc:33). The service request will be performed by Joe's Auto Body Shop (ClaimContact cc:16) at 1313 Mockingbird Lane in Arcadia, California, for Robert Farley (ClaimContact cc:13). There is one service to be performed: Towing (autoothertowing).

```
POST http://localhost:8080/cc/rest/claim/v1/claims/demo_sample:20/service-requests

{
  "data": {
    "attributes": {
       "kind": {
           "code": "unmanaged"
       },
       "specialist": {
           "id": "cc:16"
       },
       "instruction": {
         "customer": {
           "id": "cc:13"
         },
         "serviceAddress": {
                        "addressLine1": "1313 Mockingbird Lane",
                        "city": "Arcadia",
                        "country": "US",
                        "postalCode": "91006",
                        "state": {
                            "code": "CA",
                            "name": "California"
                          }
         },
         "services": [
           {
             "code": "autoothertowing"
           }
         ]
      }
    }
  }
}
```

Modifying existing service requests

Use the following endpoints to modify a service request without advancing it through its lifecycle.

PATCHing service requests

To PATCH a service request, use:

- PATCH `/claims/{claimId}/service-requests/{serviceRequestId}`

You cannot PATCH any field in the base configuration, as all of them can be set during creation only. But, if your instance includes extension fields on `ServiceRequest` or a related entity, you could use this endpoint to update those fields.

Specifying the reason for change

In ClaimCenter, the `ServiceRequest` entity has a `History` array which contains a set of `ServiceRequestChange` instances. The `ServiceRequestChange` entity has a `Description` field, which is used to capture the reason for the change.

Whenever a service request is modified through a PATCH, the `Description` field is set using the following display key:

```
Rest.Claim.V1.ServiceRequest.PropertiesChanged = Service request values changed\: {0}
```

The `{0}` placeholder is populated with a list of the schema properties that have been changed. You can configure the value of the `Description` field by modifying this display key.

Assigning service requests to users

Every service request is assigned to a group and a user in that group. This user has the primary responsible for managing the service request.

When you create a service request through the system APIs, ClaimCenter automatically executes the service request assignment rules to initially assign the service request to a group and user. You can use the POST `/claims/{claimId}/service-requests/{serviceRequestId}/assign` endpoint to reassign the service request as needed.

> **Note:** The functionality for assigning service requests is a subset of the functionality for assigning activities. All assignment options that are applicable to both activities and service requests have the same behavior.

Assignment options

A service request 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 ClaimCenter uses assignment rules to select a user in that group)
- To the claim owner
- By re-running the service request assignment rules
  - This can be appropriate if you have modified the service request since the last time assignment rules were run and the modification might affect who the service request would be assigned to.

The root resource for the `/{serviceRequestId}/assign` endpoint is `ServiceRequestAssignee`. This resource specifies assignment criteria. The `ServiceRequestAssignee` schema has the following fields:

Field	Type	Description
`autoAssign`	Boolean	Whether to assign the service request using assignment rules
`claimOwner`	Boolean	Whether to assign the service request to the claim owner
`groupId`	string	The ID of the group to assign the service request to
`userId`	string	The ID of the user to assign the service request to

The `Assignee` resource cannot be empty. It must specify a single assignment option (group and user, group only, claim owner, or automatic assignment).

For more information on how assignment rules execute assignment, see the *Rules Guide*.

Assignment example - Assigning to a specific group (and user)

The following assigns service request cc:102 (from claim demo_sample:20) to group demo_sample:31 (Auto1 - TeamA) and user demo_sample:2 (Sue Smith).

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:102/assign

{
  "data": {
    "attributes" : {
      "groupId" : "demo_sample:31",
      "userId" : "demo_sample:2"
    }
  }
}
```

The following assigns service request cc:102 (from claim demo_sample:20) to group demo_sample:31 (Auto1 - TeamA). Because no user has been specified, ClaimCenter will execute assignment rules to assign the service request to a user in group demo-sample:31.

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:102/assign

{
  "data": {
    "attributes" : {
      "groupId": "demo_sample:31"
    }
```

```
  }
}
```

Note that there is currently no endpoint that returns groups or group IDs. To assign service requests to a specific group, the caller application must determine the group ID using some method other than a groups system API.

Assignment example - Assigning to the claim owner

The following assigns service request cc:102 (from claim demo_sample:20) to the group and user that owns the parent claim (demo_sample:20).

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:102/assign

{
  "data": {
    "attributes" : {
        "claimOwner" : true
    }
  }
}
```

Assignment example - Using automated assignment

The following assigns service request cc:102 (from claim demo_sample:20) using automated assignment rules.

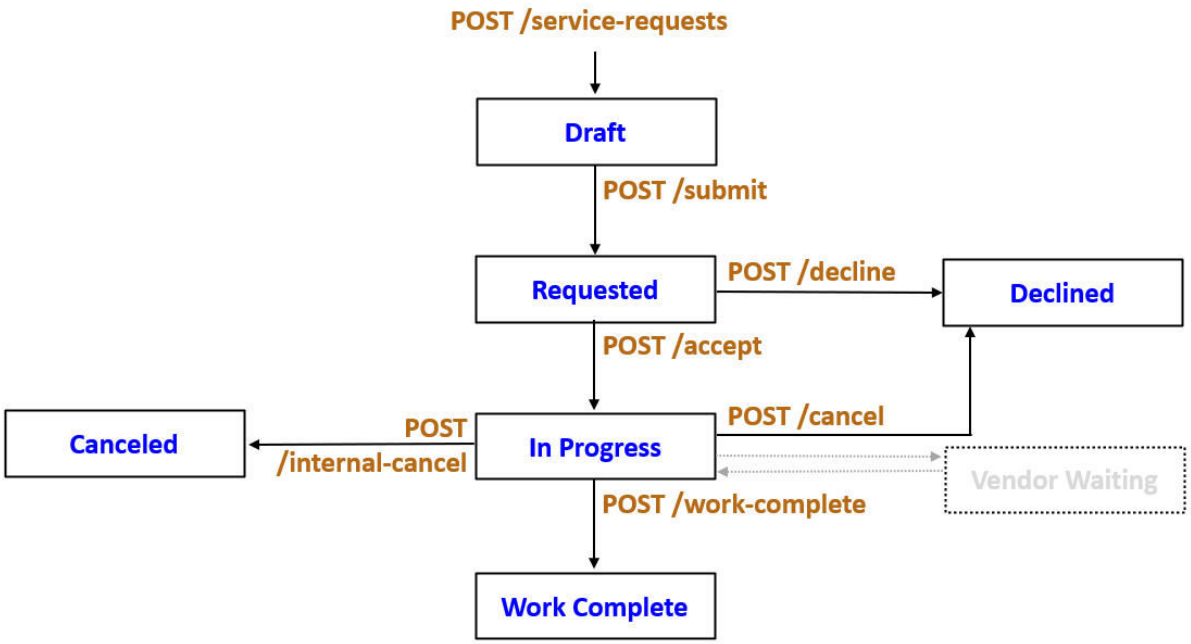
```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:102/assign

{
  "data": {
    "attributes": {
        "autoAssign" : true
    }
  }
}
```

Advancing a service request in its lifecycle

Summary of the service request lifecycle

The following diagram identifies the lifecycle of a service request and the endpoints used to advance the service request to each stage.



Note: As of this release, there are no system API endpoints to advance a service request to the Vendor Waiting stage.

Be aware that you can create and submit a service request in a composite request. But you cannot advance a service request to any other stage in its life cycle (such as in progress, declined, or canceled).

The ServiceRequestOperationContext resource

The endpoints that advance a service request to the next stage in the lifecycle use a `ServiceRequestOperationContext` resource. This resource contains fields that map to the `ServiceRequest` entity in ClaimCenter. In most cases, when using a service request lifecycle endpoint, there is a single field you must specify, such as the `reason` field, which must be specified when declining or canceling a service request. Required fields are specified in the following sections.

If an insurer has extended the behavior of the service request workflow in ClaimCenter, the insurer may also need to extend the `ServiceRequestOperationContext` resource so that values can be provided by system APIs as needed. For more information on extended system API resources, see “Extending system API resources” on page 217.

Submitting, accepting, and declining service requests

Submitting service requests

Service requests that are Quote Only, Quote and Service, or Service Only must be submitted to a vendor. (Unmanaged service requests are automatically marked as submitted to the specified vendor.)

To indicate that a service request has been submitted to the vendor, use:

- POST `/claims/{claimId}/service-requests/{serviceRequestId}/submit`

When submitting a service request, there is no additional required information. The response can have no body.

The following submits service request cc:9:

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:9/submit

<no request body>
```

When a Draft service request has been submitted, its **Progress** field is set to Requested.

Accepting service requests

Once a service requests that is Quote Only, Quote and Service, or Service Only is submitted to a vendor, it can be accepted by the vendor. This means the vendor has agreed to take on the service request. (Unmanaged service requests are automatically accepted by the specified vendor.)

To indicate a vendor has accepted a service request, use:

- POST `/claims/{claimId}/service-requests/{serviceRequestId}/accept`

When accepting a submitted service request, you must specify an `expectedCompletionDate`.

The following accepts service request cc:9:

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:9/accept

{
  "data": {
    "attributes": {
      "expectedCompletionDate" : "2021-03-22"
    }
  }
}
```

When a Requested service request has been accepted, its **Progress** field is set to In Progress.

Declining service requests

After a service requests that is Quote Only, Quote and Service, or Service Only has been submitted to a vendor, it can be declined by the vendor. This means the vendor is not going to take on the service request.

To indicate a vendor has declined a service request, use:

- POST `/claims/{claimId}/service-requests/{serviceRequestId}/decline`

When declining a submitted service request, you must specify a reason for the decline.

The following declines service request cc:9:

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:9/decline

{
  "data": {
    "attributes": {
        "reason" : "All mechanics are booked through the end of the month."
    }
  }
}
```

When a Requested service request has been declined, its **Progress** field is set to Declined.

Completing and canceling service requests

Completing service requests

Once a service requests has been accepted by the vendor, you can specify that the work is completed.

To indicate that work on a service request is complete, use:

- POST `/claims/{claimId}/service-requests/{serviceRequestId}/complete-work`

When indicating work is completed on a service request, there is no additional required information. The response can have no body.

The following indicates the work is complete for service request cc:9:

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:9/work-complete

<no request body>
```

When an In Progress service request has been completed, its **Progress** field is set to Work Complete.

Canceling service requests (at the vendor's request)

Even after a service request has been accepted, it can be canceled by the vendor.

To indicate that work on a service request is canceled at the vendor's request, use:

- POST `/claims/{claimId}/service-requests/{serviceRequestId}/cancel`

When canceling an accepted service request, you must specify a reason for the cancellation.

The following cancels service request cc:9 at the vendor's request:

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:9/cancel

{
  "data": {
    "attributes": {
      "reason" : "Vendor realized they cannot service this type of auto."
    }
  }
}
```

When an In Progress service request has been canceled by the vendor, its **Progress** field is set to Declined. This is the same state a service request is set to if the service request is initial declined rather than accepted.

Canceling service requests (at the insurer's request)

Even after a service request has been accepted, it can be canceled by the insurer.

To indicate that work on a service request is canceled at the insurer's request, use:

- POST `/claims/{claimId}/service-requests/{serviceRequestId}/internal-cancel`

When canceling an accepted service request, you must specify a reason for the cancellation.

The following cancels service request cc:9 at the insurer's request:

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:9/internal-cancel

{
  "data": {
    "attributes": {
      "reason" : "Claimant has decided not to request service."
    }
  }
}
```

When an In Progress service request has been canceled by the insurer, its **Progress** field is set to Canceled.

Service request invoices

Once a service request that is Quote and Perform Service, Service Only, or Unmanaged reaches the Work Complete stage, invoices can be created for the service request.

An invoice consists of:

- A description
- An optional reference number
- One or more line items. Each line item consists of:
  - A monetary amount (an amount and currency)
  - A category
  - An optional description

Note that there are mechanisms for suspending further work on both accepted service requests and service request invoices. But the verb used for each mechanism is different.

- To stop further work on an accepted service request, you *cancel* it (or *internal-cancel* it).
- To stop further work on an invoice, you *withdraw* it.

Note: You cannot create or update the child objects of a service request (such as service request invoices) in a composite request.

Querying for invoices

The following Claim API endpoints can be used to request information about service request invoices:

Endpoint	Response
GET `/claims/{claimId}/service-requests/{serviceRequestId}/invoices`	All invoices for the specified service request
GET `/claims/{claimId}/servicerequests/{serviceRequestId}/invoices/{invoiceId}`	The specified invoice. Note that in order to get information about a specific invoice, you must access the invoice from its parent claim and service request.

Creating invoices for service requests

To create an invoice for a service request, use:

- POST `/claims/{claimId}/service-requests/{serviceRequestId}/invoices`

Minimum creation criteria

An invoice must have the following information:

- A description
- A set of one of more line items. Each line item must specify:
  - A monetary amount (with amount and currency values)
  - A category (a code from the `ServiceRequestStatementLineItemCategory` typelist)

The following creates an invoice for service request cc:9. The invoice contains two line items: a $150 USD charge for labor, and a $50 USD charge for parts. It also includes an optional reference number for the invoice, and optional descriptions for each line item.

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:9/invoices

{
  "data": {
    "attributes": {
            "description": "Invoice submitted using system APIs",
            "referenceNumber": "771-DX5667",
            "lineItems": [
              {
                "amount": {
                  "amount": "150.00",
                  "currency": "usd"
                  },
                "category": {
                  "code": "labor"
                },
                "description": "Invoiced labor"
              },
              {
                "amount": {
                  "amount": "50.00",
                  "currency": "usd"
                  },
                "category": {
                  "code": "parts"
                },
                "description": "Invoiced parts"
              }
            ]
    }
  }
}
```

Withdrawing service request invoices

To withdraw an invoice, use:

- POST `/claims/{claimId}/service-requests/{serviceRequestId}/invoices/{invoice-id}/withdraw`

When withdrawing an invoice, you must specify a reason. This can be set to any string value.

When you withdraw an invoice, the invoice is flagged as withdrawn in the user interface.

The following withdraws invoice cc:06 for service request cc:9.

```
POST /claim/v1/claims/demo_sample:20/service-requests/cc:9/invoices/cc:6/withdraw

{
  "data": {
    "attributes": {
            "reason": "Invoice submitted in error"
    }
  }
}
```

chapter 18

Working with activities

An *activity* is an action related to the processing of a claim 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 responsible for closing the activity.

Activities are typically created by users or by automatic ClaimCenter 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 ClaimCenter, 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 claims, which do not exist in all InsuranceSuite applications. This means that other activity endpoints, such as the endpoint for creating an activity for a claim, are declared in the Claim 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 ClaimCenter, activities can be attached only to claims (directly or indirectly).

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
`/claim/v1/claims/{claimId}/activities`	All activities associated with the given claim

Creating activities

Activities must be created from an existing claim using the following endpoint:

- POST `claim/v1/claims/{claimId}/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 `/claim/v1/claims/{claimId}/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 claim cc:102. The activity defaults to all values in the `contact_insured` activity pattern.

```
POST /claim/v1/claims/cc:102/activities

{
  "data": {
    "attributes": {
      "activityPattern": "contact_insured"
    }
  }
}
```

The following is an example of creating a `legal_review` activity for claim cc:102. In this case, two activity pattern values are overridden: the activity is `mandatory` (it cannot be skipped) and the `priority` is urgent.

```
POST /claim/v1/claims/cc:102/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, ClaimCenter 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 ClaimCenter uses assignment rules to select a user in that group)
- To a specific group and queue
- To the claim owner
- 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
`claimOwner`	Boolean	Whether to assign the activity to the claim owner
`groupId`	string	The ID of the group to assign the activity to
`queueId`	string	The ID of the queue 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, ClaimCenter 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 the claim owner

The following payload assigns activity xc:1 to the group and user that owns the claim that the activity is associated with.

```
POST /common/v1/activities/xc:1/assign

{
  "data": {
    "attributes" : {
        "claimOwner" : true
    }
  }
}
```

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 *Rules Guide*.

Retrieving recommended assignees

When ClaimCenter users are assigning activities manually, the user interface includes a drop-down list of "recommended assignees". Typically, this list includes:

- The option to use assignment rules
- The option to assign the activity to the user who owns the activity's claim
- 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
`/claim/v1/claims/{claimId}/activity-assignees`	The list of suggested assignees for activities on this claim

The following is a portion of an example response from the Common API's `/assignee` endpoint.

```
GET /common/v1/activities/cc:301/assignees

{
    "count": 16,
    "data": [
        {
            "attributes": {
                "autoAssign": true,
                "name": "Use automated assignment"
            }
        },
        {
            "attributes": {
                "claimOwner": true,
                "name": "Claim/Exposure Owner"
            }
        },
        {
            "attributes": {
                "groupId": "demo_sample:31",
                "name": "Sue Smith (Auto1 - TeamA)",
                "userId": "demo_sample:2"
            }
        },
        {
            "attributes": {
                "groupId": "demo_sample:31",
                "name": "Andy Applegate (Auto1 - TeamA)",
                "userId": "demo_sample:1"
            }
        },
      ...
        {
            "attributes": {
                "name": "FNOL - Acme Insurance",
                "queueId": "default_data:1"
            }
        }
    ],
    ...
```

Closing activities

A general 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. (Approval activities, which are discussed later in this topic, are closed in a different way.)

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"
      }
    ]
  }
}
```

Approving an approval activity

Approval activities are associated with actions that require approval from a user with sufficient authority, such as a manager. Approval activities are closed either by approving or rejecting the activity. This either allows or prevents the associated action.

Only approval activities can be closed by being approved or rejected. General activities must be closed by being completed or skipped.

Approval activities often involve financial activities, such as sending money to an insured or a third party. As an added layer of protection, caller applications may want to use checksums with calls to the `/approve` endpoint to ensure that no changes were made to the activity between the time it was retrieved and the time it is to be approved. For more information on checksums, see “Lost updates and checksums” on page 99.

When approving an activity, the options for the request payload are:

- An empty payload
- A payload with an approval rationale. (This is a string value that describes why the activity was approved or rejected.)
- A payload with an `included` note.
- A payload with an approval rationale and an `included` note.

The following payload approves activity xc:2.

```
POST /common/v1/activities/xc:2/approve

<no request payload>
```

The following payload approves activity xc:2 with an approval rationale.

```
POST /common/v1/activities/xc:2/approve

{
  "data":
    {
      "attributes": {
        "approvalRationale": "Higher reserve approved because claimant is gold-tier customer."
      }
    }
}
```

The following payload approves activity xc:2 with an approval rationale and a note.

```
POST /common/v1/activities/xc:2/approve

{
  "data":
    {
      "attributes": {
        "approvalRationale": " Higher reserve approved because claimant is gold-tier customer"
      }
    },
  "included": {
    "Note": [
      {
        "attributes": {
          "body": "This activity was approved through a system API call."
        },
        "method": "post",
        "uri": "/common/v1/activities/xc:2/notes"
      }
    ]
  }
}
```

There are currently no Cloud API endpoints that reject approval activities.

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 209.

chapter 19

Working with documents

In ClaimCenter, a document is a file (such as a PDF, Word document, or digital photograph) which contains information relevant to a claim. Typically, a document is an electronic file, though some insurers may also maintain documents as physical files. Examples of documents could include reports filed by police offices, assessments of damage from home inspectors, or correspondences with the insured.

For a complete description of the functionality of documents in ClaimCenter, 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 claims, which do not exist in all InsuranceSuite applications. This means that other document endpoints, such as the endpoint for creating a document for a claim, are declared in the Claim 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, ClaimCenter documents are always attached to claims.

Document metadata and content

The ClaimCenter 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. ClaimCenter 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 ClaimCenter 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 ClaimCenter 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
`/claim/v1/claims/{claimId}/documents`	Metadata for documents on the given claim
`/claim/v1/claims/{claimId}/documents/{documentId}`	Metadata for the given document

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`
- `/claim/v1/claims/{claimId}/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.

- `/claim/v1/claims/{claimId}/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 ClaimCenter using a `/documents` endpoint. In this approach:
   - ClaimCenter 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 ClaimCenter. 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 claim cc:102 through ClaimCenter.

```
POST /claim/v1/claims/cc:102/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 a claim on an instance of ClaimCenter on your machine, enter: `http://localhost:8080/cc/rest/claim/v1/claims/{claimId}/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}`
- `/claim/v1/claims/{claimId}/documents/{documentId}`

Every document has a status. Once a document's status is set to "final", it can no longer be edited, either from the ClaimCenter user interface or through the system APIs.

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}`
- `/claim/v1/claims/{claimId}/documents/{documentId}`

chapter 20

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 ClaimCenter. They can also be created by caller applications using system APIs.

Through Cloud API, a note can be attached to a claim. It can also optionally be attached to an exposure, a ClaimContact, or a service request on that claim. Notes can also be attached to activities.

For a complete description of the functionality of notes in ClaimCenter, 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 endpoint for querying for a note with a given ID, are declared in the Common API. Notes can also belong to claims, which 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 claim, are declared in the Claim 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)
`/claim/v1/claims/{claimId}/notes`	All notes associated with the given claim
`/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 ClaimCenter. This includes notes that are attached to a parent object other than a claim.

Creating claim notes

Notes must be created from an existing claim or activity using one of the following endpoints:

- POST `claim/v1/claims/{claimId}/notes`
- POST `common/v1/activities/{activityId}/notes`

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
`relatedTo`	Inline object	For notes attached to claims, this is either the parent claim, or the child object (the ClaimContact, exposure, or service request), if any, that the note is related to	The parent claim
`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 minimal note for claim cc:102.

```
POST /claim/v1/claims/cc:102/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 claim cc:102.

```
POST /claim/v1/claims/cc:102/notes

{
  "data": {
    "attributes": {
        "body": "The insured's last name, Cahill, is pronounced 'KAH-hill', not 'KAY-hill'." ,
        "confidential": false,
        "securityType": {
          "code": "public"
        },
        "subject": "Pronunciation note",
        "topic": {
          "code": "general"
        }
    }
  }
}
```

Notes attached to child objects

By default, every note is attached only to the parent claim. You can attach a note to one of the claim's child objects using the `relatedTo` field. This field has the following syntax:

```
"relatedTo": {
  "id": "<childObjectId>",
  "type": "<childObjectType>"
```

For example, the following creates a note on claim cc:102 for the exposure with id cc:48:

```
POST /claim/v1/claims/cc:102/notes

{
  "data": {
    "attributes": {
        "body": "The claimant's last name, Cahill, is pronounced 'KAH-hill', not 'KAY-hill'.",
        "relatedTo": {
          "id": "cc:48",
          "type": "Exposure"
        }
    }
  }
}
```

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 21

Working with users

With the Admin API, authorized callers can create, update, and retrieve system user data through the `/admin/v1/users` endpoints.

Querying for users

Authorized callers can query for a user through the `/admin/v1/users/{userId}` endpoint. Calls to this endpoint return a User resource:

```
{
    "data": {
        "attributes": {
            "active": true,
            "displayName": "Andy Applegate",
            "employeeNumber": "1000001",
            "externalUser": false,
            "firstName": "Andy",
            "id": "demo_sample:1",
            "lastName": "Applegate",
            "username": "aapplegate",
            "vacationStatus": {
                "code": "atwork",
                "name": "At work"
            },
            "workPhone": {
                "displayName": "213-555-8164",
                "number": "2135558164"
            }
        },
        "checksum": "0",
        "links": {
            "self": {
                "href": "/admin/v1/users/demo_sample:1",
                "methods": [
                    "get",
                    "patch"
                ]
            }
        }
    }
}
```

In order to expose the GET `/admin/v1/users/{userId}` endpoint, the GET `/admin/v1/users` collection must also be exposed. However, the latter endpoint is not filterable or sortable, and can possibly return many pages. In light of this limitation, querying the users collection for a specific user can require an unknown number of calls. Thus Guidewire recommends that callers do not use the `/admin/v1/users` collection to find a specific user.

Creating users

Authorized callers can create users. To create a user, callers can submit a POST request to the `/admin/v1/users` endpoint.

At minimum, the request body must contain a value for the `username` field:

```
{
  "data": {
    "attributes": {
      "firstName": "Alfred",
      "lastName": "Martin",
      "username": "amartin"
    }
  }
}
```

The call returns a User resource for the new user:

```
{
    "data": {
        "attributes": {
            "active": true,
            "displayName": "Alfred Martin",
            "externalUser": false,
            "firstName": "Alfred",
            "id": "cc:203",
            "lastName": "Martin",
            "username": "amartin",
            "vacationStatus": {
                "code": "atwork",
                "name": "At work"
            }
        },
        . . .
    }
}
```

Updating users

Authorized callers can update users. To update a user, callers can submit a PATCH request to the `/admin/v1/users/{userId}` endpoint.

part 3

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)

chapter 22

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 ClaimCenter 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 ClaimCenter entity and its fields. An *updater* enables resource data to be written to ClaimCenter, 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 225.

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 ClaimCenter entities.
- `updaters` contains updaters that make resource data writeable to ClaimCenter.

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 ClaimCenter. 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 ClaimCenter. 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 *Integration Guide*. 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 ClaimCenter 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 ClaimCenter 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: ClaimCenter scalar types with associated JSON primitive types

ClaimCenter 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`
`mediumtext`	`string`
`money`	`number`
`percentage`	`number`

ClaimCenter scalar type	JSON primitive type
shorttext	string
text	string
varchar	string

The following example depicts base resource properties that support ClaimCenter 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 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 ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 ClaimCenter 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

ClaimCenter 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 `ClaimContact` 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 `ClaimContact.Contact.DisplayName`. The schema definition for this property is as follows:

```
"ClaimContact": {
  "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 ClaimCenter entity. This step is necessary in order to expose your resource extensions to ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 ClaimCenter `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 ClaimCenter
- `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 a claim contact is `ClaimContact.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 ClaimCenter `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 ClaimCenter.

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 ClaimCenter 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 ClaimCenter 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 ClaimCenter `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 ClaimCenter
- `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 a claim contact is `ClaimContact.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 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 ClaimCenter 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 ClaimCenter, 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.cc.common.v1.common_content-1.0",
    "ext.framework.v1.framework_ext-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 218.

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.cc.common.v1.common_content-1.0",
    "ext.framework.v1.framework_ext-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 ClaimCenter `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.cc.common.v1.common_content-1.0",
    "ext.framework.v1.framework_ext-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 ClaimCenter `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 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 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:8080/cc/rest/common/v1/activites/cc:201`
4. Review the body of the response. It appears as follows:


```
{
    "data": {
        "attributes": {
            "activityClass_Ext": {
                "code": "task",
                "name": "Task"
            },
            "activityPattern": "vendor_did_not_accept_work",
            "activityType": {
                "code": "general",
                "name": "General"
            },
            "assignedByUser": {
                "displayName": "System User",
                "id": "systemTables:2"
            },
            "assignedGroup": {
                "displayName": "Auto1 - TeamA",
                "id": "demo_sample:31"
            },
            "assignedUser": {
                "displayName": "Andy Applegate",
                "id": "demo_sample:1"
            },
            "assignmentStatus": {
                "code": "assigned",
                "name": "Assigned"
            },
            "createTime_Ext": "2020-06-04T22:10:00.091Z",
            "createUser_Ext": {
                "displayName": "System User",
                "id": "systemTables:2"
            },
            "description": "Follow up with vendor - work not accepted in timely manner",
            "dueDate": "2020-06-05T22:10:00.035Z",
            "escalated": false,
            "externallyOwned": false,
            "id": "cc:201",
            "importance": {
                "code": "high",
                "name": "High"
            },
            "mandatory": false,
            "priority": {
                "code": "high",
                "name": "High"
            },
            "recurring": false,
            "status": {
                "code": "open",
                "name": "Open"
            },
            "subject": "Follow up with vendor - work not accepted in timely manner"
```

```
        },
        . . .
        }
    }
}
```

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:8080/cc/rest/common/v1/activites/cc:201`
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": "vendor_did_not_accept_work",
            "activityType": {
                "code": "general",
                "name": "General"
            },
            "assignedByUser": {
                "displayName": "System User",
                "id": "systemTables:2"
            },
            "assignedGroup": {
                "displayName": "Auto1 - TeamA",
                "id": "demo_sample:31"
            },
            "assignedUser": {
                "displayName": "Andy Applegate",
                "id": "demo_sample:1"
            },
            "assignmentStatus": {
                "code": "assigned",
                "name": "Assigned"
            },
            "createTime_Ext": "2020-06-04T22:10:00.091Z",
            "createUser_Ext": {
                "displayName": "System User",
                "id": "systemTables:2"
            },
            "description": "Follow up with vendor - work not accepted in timely manner",
            "dueDate": "2020-06-05T22:10:00.035Z",
            "escalated": false,
            "externallyOwned": false,
            "id": "cc:201",
            "importance": {
                "code": "high",
                "name": "High"
            },
            "isAutogenerated_Ext": true,
            "mandatory": false,
            "priority": {
                "code": "high",
                "name": "High"
            },
            "recurring": false,
            "shortSubject_Ext": "shortsub",
            "status": {
```

```
                "code": "open",
                "name": "Open"
            },
            "subject": "Follow up with vendor - work not accepted in timely manner"
        },
        . . .
    }
}
```

Providing feedback

The system APIs expose a subset of ClaimCenter 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 23

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 217.

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

For example, a claim has ClaimContacts. Depending on business purposes, it might have been necessary to obtain tax identification information for a ClaimContact. Later, a system API caller could request the ClaimContact 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.

```
"ClaimContact": {
  "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 ClaimContact mapper as follows:

```
"ClaimContact": {
  "schemaDefinition": "ClaimContact",
  "root": "entity.ClaimContact",
  "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 217.

For example, a claim has ClaimContacts. Depending on business purposes, it might have been necessary to obtain tax identification information for a ClaimContact. Later, a system API caller could request the ClaimContact 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 ClaimContact resource contains a `taxId` property:

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

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

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

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

With the `taxId.path` property set to `ClaimContact.Contact.maskTaxId(ClaimContact.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 217. 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.
