Guidewire PolicyCenter™

REST API Framework

Release 10.0.3

GUIDEWIRE
Adapt and succeed™

Product Name: Guidewire PolicyCenter
Product Release: 10.0.3
Document Name: REST API Framework
Document Revision: 17-March-2021

Contents

Guidewire Documentation

About PolicyCenter documentation

The following table lists the documents in PolicyCenter documentation:

Document	Purpose
InsuranceSuite Guide	If you are new to Guidewire InsuranceSuite applications, read the *InsuranceSuite Guide* for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications.
Application Guide	If you are new to PolicyCenter or want to understand a feature, read the *Application Guide*. This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with PolicyCenter.
Upgrade Guide	Describes the overall upgrade process, and describes how to upgrade your configuration and database. The intended readers are system administrators and implementation engineers who must merge base application changes into existing application extensions and integrations. Visit the Guidewire Community to access the *Upgrade Guide*, which is available for download, separately from the main documentation set, with the Guidewire InsuranceSuite Upgrade Tools.
Installation Guide	Describes how to install PolicyCenter. The intended readers are everyone who installs the application for development or for production.
System Administration Guide	Describes how to manage a PolicyCenter system. The intended readers are system administrators responsible for managing security, backups, logging, importing user data, or application monitoring.
Configuration Guide	The primary reference for configuring initial implementation, data model extensions, and user interface (PCF) files for PolicyCenter. The intended readers are all IT staff and configuration engineers.
PCF Format Reference	Describes PolicyCenter PCF widgets and attributes. The intended readers are configuration engineers. See the *Configuration Guide*
Data Dictionary	Describes the PolicyCenter data model, including configuration extensions. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers.
Security Dictionary	Describes all security permissions, roles, and the relationships among them. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers.
Globalization Guide	Describes how to configure PolicyCenter for a global environment. Covers globalization topics such as global regions, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who localize PolicyCenter.
Rules Guide	Describes business rule methodology and the rule sets in Guidewire Studio for PolicyCenter. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu.
Guidewire Contact Management Guide	Describes how to configure Guidewire InsuranceSuite applications to integrate with ContactManager and how to manage client and vendor contacts in a single system of record. The intended readers are PolicyCenter implementation engineers and ContactManager administrators.
Best Practices Guide	A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers.
Integration Guide	Describes the integration architecture, concepts, and procedures for integrating PolicyCenter with external systems and extending application behavior with custom programming code. The intended readers are system architects and the integration programmers who write web services code or plugin code in Gosu or Java.
Java API Reference	Javadoc-style reference of PolicyCenter Java plugin interfaces, entity fields, and other utility classes. The intended readers are system architects and integration programmers.

Document	Purpose
Gosu Reference Guide	Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration.
Gosu API Reference	Javadoc-style reference of PolicyCenter Gosu classes and properties. The reference can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers, system architects, and integration programmers.
Testing Guide	Describes the tools and functionality provided by InsuranceSuite for testing application behavior during an initial implementation or an upgrade. The guide covers functionality related to Behavior Testing Framework, GUnit, and Gosu functionality designed specifically for application testing. There are two sets of intended readers: business analysts who will assist in writing tests that describe the desired application behavior; and technical developers who will write implementation code that executes the tests.
Glossary	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.
Product Model Guide	Describes the PolicyCenter product model. The intended readers are business analysts and implementation engineers who use PolicyCenter or Product Designer. To customize the product model, see the *Product Designer Guide*.
Product Designer Guide	Describes how to use Product Designer to configure lines of business. The intended readers are business analysts and implementation engineers who customize the product model and design new lines of business.
REST API Framework	Describes the Guidewire InsuranceSuite framework that provides the means to define, implement, and publish REST API contracts. It also describes how the Guidewire REST framework interacts with JSON and Swagger objects. The intended readers are system architects and integration programmers who write web services code or plugin code in Gosu or Java.

Conventions in this document

Text style	Meaning	Examples
italic	Indicates a term that is being defined, added emphasis, and book titles. In monospace text, italics indicate a variable to be replaced.	A *destination* sends messages to an external system. Navigate to the PolicyCenter installation directory by running the following command: `cd` *`installDir`*
bold	Highlights important sections of code in examples.	`for (i=0, i<someArray.length(), i++) {` **`newArray[i] = someArray[i].getName()`** `}`
narrow bold	The name of a user interface element, such as a button name, a menu item name, or a tab name.	Click **Submit**.
`monospace`	Code examples, computer output, class and method names, URLs, parameter names, string literals, and other objects that might appear in programming code.	The `getName` method of the `IDoStuff` API returns the name of the object.
`monospace italic`	Variable placeholder text within code examples, command examples, file paths, and URLs.	Run the `startServer` *`server_name`* command. Navigate to `http://`*`server_name`*`/index.html`.

Support

For assistance, visit the Guidewire Community.

Guidewire customers

`https://community.guidewire.com`

Guidewire partners

`https://partner.guidewire.com`

chapter 1

Guidewire REST APIs

Guidewire released the basic Guidewire REST API framework in InsuranceSuite 10.0.0. The Guidewire REST API framework, in combination with Guidewire Integration Views, provides the InsuranceSuite platform with support for the following:

- Swagger and OpenAPI 2.0 API schemas
- JSON data schema payloads
- Mappings from the InsuranceSuite data model schema to JSON

Working with custom REST APIs

If you intend a large scale implementation of REST APIs (or conversion of existing APIs to RESTful APIs), Guidewire recommends that you contact your account representative for more information. For more immediate REST API needs, Guidewire provides this document as a guide towards a self-managed implementation of APIs using the InsuranceSuite 10.0 REST API framework.

REST API building blocks

The Guidewire REST API framework uses the following types of files to define the API contract:

- Swagger 2.0 schema files to define the structure of a given API
- JSON schema files to define the schema for API inputs and outputs

Together, the defined Swagger and JSON schema file determine the following parts of the REST API.

API resources

Swagger files define the set of endpoint resources exposed by the API. Each resource is called a "path" or "path item" in Swagger, and can have variables interposed for things such as resource IDs. For example, the following string can define a resource for the addresses associated with a specific user:

```
/users/{contactId}/addresses
```

In this example, `/users` and `/addresses` are Swagger path items.

HTTP verbs

Swagger files define the set of HTTP verbs it is possible to use to operate on a particular resource. Swagger calls these verbs the API "operations." For example:

- A `GET` operation on `/users/{userId}/addresses` returns the addresses for the specified user.
- A `POST` operation on `/users/{userId}/addresses` adds a new address to the specified user.
- A `PATCH` operation on `/users/{userId}/addresses/{addressId}` updates existing information on the specified user/address.

- A `DELETE` operation on `/users/{userId}/addresses/{addressId}` deletes the specified user/address.

Payload schema

JSON files define the schema for the payload (the body of the request) that the API passes during request, if any. In the previous example, a `POST` operation to resource `/users/{userId}/addresses` can specify a JSON schema definition that defines the format for the body of the `POST` request.

Response schema

JSON files define the schema for the response returned by an operation, if any, as well as any possible HTTP codes that the operation returns. For example, a `GET` request to resource `/users/{userId}/addresses` can specify a JSON schema for the list of addresses and that the default success code is 200.

Path and custom headers

Swagger files define the set of path parameters and custom headers that are applicable to each operation. In the previous example, `{userId}` and `{addressId}` are path parameters. At runtime, the REST API framework matches the request URL against the defined path and extracts out those path parameters for use by the application code.

Validation and serialization/deserialization

Swagger files define the validation and serialization/deserialization process for API parameters, and API inputs and outputs. For example, it is possible to declare a parameter as required, or with a specified minimum and maximum value, or as an integer rather than a string. The REST API framework can then use that information to automatically validate the inputs and return an appropriate HTTP response if the input is invalid, and can deserialize the data into appropriate POJOs for use by the application code

Each operation defined within the Swagger schema binds to a specific handler class and method. The REST API framework then uses that class method to process the request whenever a request is made against the appropriate URL and HTTP method.

Working with REST files in Guidewire Studio

Only place REST configuration files in the following locations in Guidewire Studio:

configuration→config→Integration→...

configuration→gsrc→*namespace*→...

These Studio folders contain the following types of files:

Studio location	File types
config→Integration	Swagger and JSON schema files, mapping files, and similar items.
gsrc→*namespace*→...	Gosu classes that perform the actual API work, API handler classes, for example. Place your files in their own namespace so as to keep these files separate from Guidewire files.

Use a unique name space

Guidewire recommends that you place any REST-related files that you create in their own name space. Using your company name in the file path segregates and isolates any files that you add to the PolicyCenter installation.

Thus, for files that you add, the `apis` directory structure underneath the Studio **Integrations** node looks similar to the following:

```
/apis
  /mycompany
```

Place the Gosu resources or authorization files that you create in a subdirectory underneath **gsrc** that uses your company name in the file path, for example:

configuration→gsrc→mycompany→...

REST configuration files

Place the YAML Swagger and JSON configuration files that you create in subdirectories of the **Integration** directory in the Guidewire Studio **Project** window:

configuration→config→Integration→...

The **Integration** folder contains the following directory structure.

`/apis`

The **Integration→apis** folder contains the following types of files.

File type	Description
`*name-version*.swagger.yaml`	Describes the Swagger schema to use with the REST API, including: • The endpoints the REST API supports • The methods each path supports • The schema responses to expect
`published-apis.yaml`	Controls which Swagger schemas the REST servlet exposes. Every API that file `published-apis.yaml` lists is addressable through the REST servlet.

`/filters`

The **Integration→filters** folder contains the following types of files.

File type	Description
`*name-version*.gql`	Provides GraphQL-style filters; the filter serves as a white list of properties to include with object fields to actually materialize and serialize.

`/mappings`

The **Integration→mappings** folder contains the following types of files.

File type	Description
`*name-version*.mapping.json`	Describe how to transform PolicyCenter data into a JSON object that conforms to a specific JSON schema document.

`/schemas`

The **Integration→schemas** folder contains the following file types.

File type	Description
`*name-version*.schema.json`	Describes the structure of a JSON object with which an API client interacts.
`schema_reserved_words.txt`	Defines a set of reserved words that are impermissible to use as JSON schema property names.

How to version configuration files

The Swagger (`*.yaml`) and JSON (`*.json`) schema files that define a REST API exist in the following Guidewire Studio folders:

Integration→apis for Swagger schema files

Integration→mappings for JSON mapping files

Integration→schemas for JSON schema files

Every YAML and JSON file name consists of two parts, each part separated by a '-' character:

- Actual file name
- Version number

Use the following syntax in naming YAML (Swagger) and JSON files:

```
name-version.swagger.yaml
name-version.schema.json
```

The use of version numbers provides for the following capabilities:

- It provides for a common namespace for a set of configuration files.
- It provides for the ability to create different functionality or behavior for different versions of the API.
- It provides for the ability to publish different versions of an API in different environments.

Guidewire recommends that you use this same file naming scheme for any Gosu handler class and resource files that you create as well.

Fully qualified file names

The file name and the subfolder path all become part of the fully qualified name of the file. The folder path to the actual file becomes part of the fully qualified file name similar to a package name in Java, for example:

```
gw.system.server.profiler.base-1.0.swagger.yaml
```

REST configuration files that change

If the application server is operating in development mode, the REST API framework monitors the file system for changes to configuration files. The REST framework checks every API request to determine if the request requires a reload of the REST servlet configuration. If the framework detects a configuration change, the framework reloads all of the following items:

- Swagger schemas
- JSON schemas
- Integration mappings
- GraphQL filters
- File `published-apis.yaml`

The framework also reloads these configuration files after any refresh of the Gosu type system, which a hot swap of Gosu classes generally triggers. By combining dynamic reload of schema files with using the DCEVM to hot swap classes, it is possible to do much of your API development without having to restart the application server. As the REST framework instantiates the API handler classes on every API request as well, the only states that persist across API requests are the static variables that you define in the handler classes.

Notes

1. The file system watcher looks only for changes to files in directories that are present at server start up. The watcher does not watch for changes to files in directories added to the file system after the server starts.
2. The file system watcher groups together the file changes that it finds once every five seconds. Thus, it is possible for the watcher to not be aware of a recent change to a configuration file. If you change a schema file and the next API request does not seem to reflect the change, retry the request in a few seconds.

Rolling (configuration) upgrade

Guidewire supports rolling (configuration) upgrades in a production system.

Gosu classes

It is generally safe to add new Gosu classes to Guidewire PolicyCenter during a rolling upgrade. Therefore, it is generally safe to add new Gosu handler classes, and other REST Gosu configuration classes, during a rolling upgrade.

Swagger and JSON configuration files

It is safe to add, delete, or update the following file types in the Studio **Integration** folder during a rolling (configuration) upgrade.

Folder	File type
apis	• `*.swagger.yaml` • `versions.json` • `published-apis.yaml`
filters	• `*.gql`
mappings	• `*.mapping.json`
schemas	• `*.schema.json` • `codegen-schema.txt` • `versions.json`

Published APIs

It is safe to make the following changes to file `published-apis.yaml` during a rolling upgrade:

- Add or remove API entries from the file
- Change the value of a property on a published API entry in the file

REST API requests

REST (REpresentatonal State Transfer) is the idea of using HTTP protocol to view and manipulate resources. The REST API receives HTTP requests from clients and then queries the database for what it needs. After the REST API receives back what it needs from the database, the API sends back a JSON response to the requesting client.

The REST framework uses 'Unique Resource Locators' (URLs) to reference exposed entities. It also uses HTTP response codes to provide information on the state of the interaction.

HTTP operations

The Guidewire REST framework provides the following set of allowable HTTP operations on PolicyCenter resource, which are:

`GET`	Retrieves information about a resource
`PATCH`	Updates or modifies information about a resource
`POST`	Creates a resource
`PUT`	Updates or fully replaces a resource
`DELETE`	Deletes a resource

In addition to the listed standard HTTP operations, the HTTP protocol defines the following operations as well.

Method	Description
`OPTIONS`	Retrieves information about a given URL. The REST framework automatically provides a default implementation of the `OPTIONS` method that it does not authenticate. The default method returns a 200 status code and a single Allow header that contains the list of HTTP methods defined for the specified path. For example, if you define `GET` and `PATCH` for a given path, making an `OPTIONS` request to that path returns an Allow header value of "`GET`, `HEAD`, `PATCH`, `OPTIONS`".

Method	Description
`HEAD`	Returns the headers that would come back from a standard `GET` HTTP request. The REST API framework automatically implements the `HEAD` method for any path that defines a `GET` operation. The REST framework authenticates the `HEAD` method only if it authenticates the associated `GET` method as well. The default implementation of this method by the framework simply calls the `GET` HTTP method and then returns the response with an empty response body.

HTTP status codes

The REST request and response cycle generates HTTP status codes. The status codes generally fall into the following categories.

Status code	Category	Meaning
1xx	Information	Communicates transfer protocol-level information
2xx	Success	Indicates that the server accepted the client request successfully
3xx	Redirection	Indicates that the client must take some additional action in order to complete its request
4xx	Client error	Indicates that an error condition occurred on the client side of the HTTP request and response
5xx	Server error	Indicates that an error condition occurred on the server side of the HTTP request and response

The Guidewire REST API framework returns a standard set of HTTP codes for successful operations.

Operation	Response
`GET`, `PATCH`	200
`POST`	201
`DELETE`	204

REST API responses

One of the final steps in handling a REST request is the serialization of the HTTP response. During this step, it is necessary to serialize both the response body and the response headers:

- The REST API framework writes the response body to the output stream of the `HttpServletResponse` servlet container.
- The REST API framework turns the response headers into `String` objects that the servlet container is responsible for writing to the response.

The REST API Framework supports the serialization of a number of different Java object types that the API handler method can return directly, using one of the following ways:

- Set the object as the value of the body of the `Response` object.
- Add the object as the value of a header on the `Response` object.

Database transactions

It is possible for a REST API caller to supply a unique transaction ID while making a request. This ID ultimately maps to an insert operation into the `TransactionID` table in the database. If this transaction ID does not already exist, the REST request completes successfully. However, if a transaction with that ID already exists in the `TransactionID` table, the database commit for the API fails.

You can use this functionality to avoid making duplicates requests, in cases in which the request actually issues a database commit. This functionality works globally for any API handler that commits a bundle to the database.

Transaction ID usage

To use this functionality, the caller needs to specify the transaction ID in the `GW-DBTransaction-ID` custom header. The value of the header can be any alphanumeric string generated by the client, up to 128 characters. The header must be a globally unique request ID.

Transaction ID limitations

The transaction ID functionality has the following limitations:

- It only works with APIs whose handlers actually commit to the database.
- It only works if the API commits to the database a single time only (except for certain very rare exceptions)
- It only works if the database commit is one of the following:
  - The database commit is the only interesting side effect of the API call.
  - The database commit happens before any other side effect happens (such as notifications to external systems).
- It does not work for duplicate requests. Duplicate requests do not return identical responses back to the client. Instead, PolicyCenter marks the request as failed. The client code needs to decide how (or even if) to handle that situation

Reserving a database connection

To reserve a database connection for the duration of the request handling by the REST framework, set the following Swagger operation to `true`:

```
x-gw-reserve-db-connection
```

Localizing REST requests and responses

There are four possible ways in which you can localize a REST request. In order of preference, they are:

- Set values for custom headers `GW-Language` and `GW-Locale`
- Use the locale preference set on browser set by user
- Set a value for header `Accept-Language`
- Use the browser default language and locale

Guidewire recommends that you use custom headers `GW-Language` and `GW-Locale` to localize a REST request if you also need to localize the request response. This is most likely the case if there is a REST request directly from a browser or otherwise on behalf of an actual person.

Be aware that:

- The authenticated user in PolicyCenter can be a generic integration user that does not have a meaningful language or locale preference.
- Often, it is not possible for a user to change the `Accept-Language` header value if the REST request originates from a browser.

Rules for locale processing

PolicyCenter uses the following rules in determining and processing the locale on a REST request.

Rule #1 - If set, GW-Language and GW-Locale determine the language and locale for processing the request from that point on

The `GW-Language` and `GW-Locale` headers need to map to a virtual language and locale code. This means that if the header language or locale code is more specific than a language or locale configured on the server, the server falls back to the more generic option. For example, if you specify `en_GB`, which does not exist on the server, then REST processing on the server uses `en` instead.

If your header specification does not match any virtual language or locale, REST processing uses the default server language or locale instead. If this does happen, PolicyCenter does not report that the header language or locale did not match a configured server language or locale.

Rule #2 - If present, header Accept-Language sets the language for processing the request prior to authenticating the user

REST processing only uses the value of `Accept-Language` to set the language prior to authenticating the user. It sets the language only for the request, not the locale. The `Accept-Language` header functions in a similar fashion to the `GW-Language` header in that it can be a virtual language as well.

Rule #3 - If GW-Language and GW-Locale are not set, use the authenticated user's preferences for language and locale

The authenticated user's preferences override the `Accept-Language` header, but not the `GW-Language` or `GW-Locale` headers. If there is no authenticated user, or the authenticated user has no configured language preference, REST processing uses the `Accept-Language` header to set the language for remainder of REST processing.

REST-related configuration parameters

Guidewire provides a number of REST- and JSON-related configuration parameters in the basic REST API framework. You find these configuration parameters in file `config.xml`. These parameters have the following meanings.

Parameter	Default	Description
`OverrideJSONMappingWarnings`	`default`	Determines how the REST framework handles JSON mapping warnings.
`OverrideJSONSchemaWarnings`	`default`	Determines how the REST framework handles JSON schema warnings.
`OverrideRESTSchemaWarnings`	`default`	Determines how the REST API framework handles Swagger schema warnings:
`PreloadRestServletConfig`	`false`	Determines whether the REST API framework preloads the REST servlet configuration during servlet initialization.

See also

- See the PolicyCenter *Configuration Guide* for more information on these configuration parameters.

chapter 2

Accessing REST API information

Guidewire designs its REST APIs to be self-documenting at runtime. To further this end, Guidewire provides the following useful tools:

- Embedded Swagger UI tool that provides information on each published REST API and its supported HTTP operations
- Default API endpoint `/apis` that provides the list of available REST APIs

Additional Guidewire tools

In addition to the already listed tools for gathering information about the REST APIs, Guidewire also provides the following tools that provide further API documentation and testing:

Tool	More information
A `gwb` build tool for externalizing API schemas	"Exporting the API schema" on page 20
A Guidewire Profiler for REST APIs	"Profiling REST APIs" on page 20

Third-party tools

Guidewire recommends that you use a third-party tool such as Postman to view and test the REST APIs that you create. You can download Postman from the following web site:

`https://www.getpostman.com/`

The /apis endpoint

In the base configuration, Guidewire provides a special Swagger schema that the REST framework uses to document the set of APIs published on that server. By default, the REST API schema is available as a `GET` operation to the `/apis` endpoint. For example, if running the application server locally, the URL to access the API schema is the following:

localhost:8180/pc/rest/apis

The `/apis` endpoint is itself a standard Swagger API. Accessing the `/apis` endpoint provides a full list of the available REST APIs along with additional useful information.

The /apis schema definition

Guidewire defines the `/apis` endpoint in schema file `api_list_-1.0.json` located in the **Project** window in Guidewire Studio in the following location:

Integration→schemas→gw→pl→framework

In the base configuration, Guidewire lists the `api_list-1.0` schema in file `published-apis.yaml`, which makes the schema file accessible by default.

In file `published-apis.yaml`, you can do the following:

- Make the `/apis` endpoint inaccessible by removing the endpoint from the file.
- Publish the `/apis` endpoint to certain environments only by setting environment variables for the endpoint

The /swagger.json endpoint

Each individual API can also serve up its own documentation using the `/swagger.json` API that the REST framework adds automatically to the API. A `GET` request to `/`*`apiBasePath`*`/swagger.json` returns the externalized, canonical Swagger JSON for that API, suitable for use with standard tools that understand Swagger 2.0 schemas.

For example, to access the Swagger schema for the list of published APIs, use the following URL:

```
http://localhost:8180/pc/rest/apis/swagger.json
```

Accessing this URL returns an HTML-formatted version of the Swagger schema for that API endpoint.

Automatic generation of /swagger.json

The Guidewire REST framework uses the API template mechanism to add the `/swagger.json` path to the REST APIs. In the base configuration, file `published-apis.yaml` defines the default template for the REST APIs:

```
gw.pl.framework.dev_template-1.0.swagger.yaml
```

File `gw.pl.framework.dev_template-1.0` includes Swagger schema `gw.pl.framework.api_docs-1.0.swagger.yaml`, which automatically adds the following REST endpoint:

```
GET /swagger.json
```

It is possible to remove the `/swagger.json` endpoint from the list of published APIs by using a different default template.

Overview of Swagger UI

In the base configuration, each InsuranceSuite application contains a static distribution of the Swagger UI tool. You can use this tool to browse API documentation live from the same server that is publishing the APIs. The Swagger UI tool is available from the following location:

```
servletBasePath/resources/swagger-ui/
```

On the local server, the location Swagger UI becomes the following URL:

```
http://localhost:8180/pc/resources/swagger-ui
```

In this URL, the `servletBasePath` variable becomes:

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

Swagger UI performance

Swagger UI can be quite slow. As a consequence, Guidewire does not recommend that you use Swagger UI to test large APIs. For the most part, use Swagger UI to view schema information about an API only. Guidewire recommends that you use a third-party tool such as Postman to design, develop, and test REST APIs instead.

The Swagger UI screen

The Swagger UI screen contains the following elements.

The Explore entry field

At the top of the Swagger UI entry screen is a box in which you can enter the URL of a specific REST API. Clicking the **Explore** button returns the list of endpoints and HTTP operations for the that specific API. To determine the API URL, review the list of published APIs generated by the `/apis` endpoint.

The base URL link

Directly underneath the API name (in large font), the screen defines the base path for this particular API, as well as a clickable link to the Swagger schema for the API. You can also see the same information in a more human-readable form lower down on the Swagger UI screen.

The Authorize dialog

If authentication is set up correctly, Swagger UI displays an **Authorize** button near the top of the screen in which you can enter your user credentials. This enables the Swagger UI to authorize your API requests. If basic authentication is not set up, you can still browse the documentation using Swagger UI. However, you cannot try live requests, as Swagger UI cannot authenticate those requests correctly.

HTTP operations

The lower portion of the Swagger UI screen contains a list of the valid HTTP operations for the listed API, organized by API endpoint. Clicking one of the API operation buttons expands a pane that provides further information about this particular operation, including the following:

- A description of the operation
- Any parameters that the operation uses
- Information about the operation parameters such as a description, whether it is required, and the parameter type
- Response code associated with the request

You can also try out the action of a particular API operation by clicking **Try it out**. If you need to enter specific data for a parameter, use the fields that open to do so.

More information

For information on Swagger UI, refer to the following web site:

https://swagger.io/tools/swagger-ui/

Known Swagger Issue - null pointer exception

Null pointer exception in development environments

Issue

Under specific circumstances, attempting to perform an action in Swagger UI that requires PolicyCenter authentication can cause a null pointer exception. This happens if you use the same browser to log into both the PolicyCenter application and the Swagger UI.

In this case, the browser shares the session between the two applications. As a consequence, the PolicyCenter log in does not run through the REST code that is necessary for a REST API to log into the application.

Work-around

Do one of the following:

- Log out of Guidewire PolicyCenter.
- Use a different browser type for PolicyCenter than the one you are using for Swagger UI.
- Use a different tool such as Postman instead of Swagger UI.

Access Swagger UI

Before you begin

You must have a running application server before you can access Swagger UI.

Procedure

1. In Guidewire Studio, or from a command prompt, start the PolicyCenter application server.

2. After the application server starts, open a browser window and enter a URL using the following syntax:

   ```
   server:port/xx/rest/apis
   ```

   In the URL, replace *server*, *port*, and *xx* (two-letter application context) with actual values, for example:

   ```
   localhost:8180/pc/rest/apis
   ```

   Swagger UI opens and you can view information on the REST APIs.

Access information about an API endpoint

The Swagger UI provides self-documenting information about each published REST API.

Procedure

1. Ensure that the PolicyCenter application server is running.
2. Navigate to some variation of the following URL, using the correct *host*:*port* as necessary:

   ```
   http://localhost:8180/pc/resources/swagger-ui/
   ```

3. Click **GET / Returns a list of available APIs**.
4. In the pane that opens, click **Try it out**, then click **Execute**.
5. The `GET` operation returns a list of published APIs in the **Response body** area.
6. Copy the **docs** URL for the API of interest to the clipboard.
7. Paste the URL string into the box at the top of the Swagger UI screen and click **Explore**.

   The Swagger UI updates to contain information about that specific API.

Exporting the API schema

Guidewire provides the following `gwb` build tool to externalize and generate the Swagger schemas:

```
gwb genExternalSchemas
```

This command writes the REST API externalized JSON schema files to the following location in the Studio **Project** window:

configuration→build→external-schemas

After running this command, you see the schema file (`swagger.json`) and its associated XSD file (`schema.xsd`), for example:

- `gw.pl.framework.api.docs-1.0.swagger.json`
- `gw.pl.framework.api.docs-1.0.swagger.xsd`

Profiling REST APIs

Guidewire integrates the REST API framework with Guidewire Profiler, available for all InsuranceSuite applications. You must enable REST profiling for each specific endpoint that you want to profile. After you enable the profiler for a given endpoint, all requests to that endpoint generate profiler records that you can view in Guidewire Profiler. In many cases, Guidewire Profiler can be helpful in investigating the specific details of a given API for which you want more data.

To start REST profiling for a specific endpoint:

1. In PolicyCenter Server Tools, navigate to the following location:

   Guidewire Profiler→Configuration

2. Click **Enable Profiling for Rest Operations**.

To view information on a profiled REST operation:

1. In PolicyCenter Server Tools, navigate to the following location:

   Guidewire Profiler→Profiler Analysis→Rest Operation

2. Select the server for which you want to view data.

See also

- For more information on Guidewire Profiler, see the PolicyCenter *System Administration Guide*.

Logging REST APIs

Guidewire provides the means to use intentional logging with REST APIs in the default implementation class for the optional `IRestDispatchPlugin` plugin. This plugin is suitable for very high-level profiling of REST APIs, as it captures information about the API, path template, and elapsed time for each request. The log messages that the plugin generates are often useful for obtaining an aggregate view of what requests the REST framework receives and how long these requests are taking to complete.

REST endpoints and intentional logging

Guidewire provides several endoints that you can use with intentional logging.

Setting `ILConfig` options

REST endpoint `/intentionallogging/config` supports the following request types:

- `GET`
- `PUT`

API `PUT` requests accept JSON payloads that have the following form.

```
"ILConfig": {
  "type": "object",
  "properties": {
    "enabled": {
      "type": "boolean"
    }
   },
  "required": [
    "enabled"
   ]
}
```

Setting `ILElementConfig` options

REST endpoint `/intentionallogging/elements` supports the following request types:

- `GET`
- `PATCH`

API `PATCH` requests accept JSON payloads that have the following form.

```
"ILElementConfigs": {
  "type": "object",
  "properties": {
    "elementConfigurations": {
      "type": "array", "items": { "$ref": "#/definitions/ILElementConfig" }
    }
  },
  "required": [
    "elementConfigurations"
 ]
},
"ILElementConfig": {
  "type": "object",
  "properties": {
    "elementType": {
      "type": "string",
      "x-gw-type": "typekey.ILElementType",
      "x-gw-export-enumeration": true
    },
    "identifier": { "type": "string" },
      "enabled": { "type": "boolean" }
    },
    "required": [ "elementType", "identifier", "enabled" ]
}
```

chapter 3

Designing a REST API

In designing a REST API, your first task is to determine the following:

- The structure of the API resources
- The HTTP verbs to make available
- The structure of the API inputs and outputs

API definition tasks

The actual definition of the API consists of the following two tasks:

1. Create a `schema.json` file to define the structure of any JSON inputs and outputs.
2. Create a `swagger.yaml` file that defines the API paths (endpoints) and operations.

Guidewire recommendations

In general, Guidewire recommends that you use the following conventions in designing a REST API:

1. Try to structure actions as operations on resources rather than as RPC calls. This means that instead of calling an `addContact` method, you perform a `POST` operation to a `/contacts` resource. However, not all API actions fit nicely within the standard HTTP verbs.
2. Do not try to mutate (change) a resource using a `GET` operation.
3. Use the following HTTP verbs to perform actions on a resource:
   - `GET` to retrieve an existing resource or resources
   - `PATCH` to partially update a resource, updating specific fields, for example
   - `DELETE` to remove objects
   - `POST` to create new resources under an existing container
4. Use plurals for resource paths in which there are multiple sub-resources. For example, if there are multiple contacts, use `/`**`contacts`**`/{contactId}` rather than `/`**`contact`**`/{contactId}`.

Working with JSON files

Use a JSON schema to define your REST API inputs and outputs. A JSON schema consists of a set of definitions, with each definition corresponding to a logical type, which maps to a possible JSON object. These definitions themselves have properties, which define the set of properties allowed on that object. Properties can be either simple scalar values, references to other definitions, or arrays of scalars or objects.

In practice, Guidewire supports a subset of the JSON schema, draft 4 specification. The subset of the JSON schema that Guidewire defines is almost identical to the subset of the JSON schema that one can use in Swagger declarations. However, there are a few items that the Swagger specification defines that Guidewire does not support. In InsuranceSuite, the REST API framework and the Integration Views framework share the JSON schema files. Both of these frameworks makes use of the same files, with the same rules, and both frameworks support the same subset of the JSON schema.

Naming JSON schema files

As with Swagger schemas, each JSON schema file defines its own namespace and version. JSON schema files have the following characteristics:

- Contain JSON code
- Have an ending suffix of `.schema.json`

Guidewire stores JSON schemas in the following location in Guidewire Studio:

configuration→config→integration→schemas

Guidewire recommends that you create a sub-folder under the **schemas** folder using your company name to store your JSON files. For example, suppose that you create a JSON schema file named `contact-1.0.schema.json`. You then store this file in the following location in Studio:

configuration→config→integration→schemas→mycompany

The fully qualified name of the JSON schema file becomes `mycompany.contact-1.0`.

Guidewire JSON specification

See the PolicyCenter *Integration Guide* for details of the Guidewire JSON specification.

JSON scheme documentation

Refer to the following web site to view the full JSON schema documentation:

https://tools.ietf.org/html/draft-fge-json-schema-validation-00

Overview of the JSON schema format

The following code defines a basic JSON file.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "definitions": {
    "Contact" : {
      "type" : "object",
      "properties" : {
        "FirstName":
          "type": "string"
        },
        "Addresses" : {
          "type" : "array",
          "items" : {
            "type" : "object",
            "$ref" : "#/definitions/Address"
          }
        }
      }
    },
    "Address" : {
      "type" : "object",
      "properties" : {
        "AddressLine1" : {
          "type" : "string"
        },
        "State" : {
          "type" : "string",
          "x-gw-type" : "typekey.State"
        }
      }
    }
  }
}
```

In this example, the various JSON property have the following meanings.

definitions

The `definitions` section is itself an object that contains keys for each type that it defines. Each definition has a `properties` property that contains keys for each allowed property.

properties

Each property defines its own type. The type must be one of the following:

- `array`
- `boolean`
- `integer`
- `number`
- `object`
- `string`

If the type is `array`, there must be an `items` property that defines the types of items. Items can be any of the above types except for array, as Guidewire does not permit schemas with nested arrays.

If the type is `object` (for both properties or items), the `$ref` property defines the schema of the referenced object. A reference to a definition within the same file starts with `#/definitions/`.

If the type is a scalar type, it is possible to define an additional `format` and `x-gw-type` property.

Using the example JSON schema, a Contact definition looks similar to the following JSON code.

```
{
  "FirstName" : "Homer",
  "Addresses" : [
    {
      "AddressLine1" : "100 Evergreen Terrace",
      "State" : "MI"
    }
  ]
}
```

Mapping PolicyCenter objects to JSON objects

In defining your REST API, you need to develop a JSON schema that describes how to convert a given PolicyCenter entity object into a JSON object that a REST response returns. The PolicyCenter object can be nearly any data model entity. In creating your JSON mapping file, use the extension `mapping.json` and prepend the appropriate namespace for your REST configuration files to the file name.

Place any JSON mapping files that you create in the following location in Guidewire Studio:

configuration→config→Integration→mappings

Guidewire recommends that you create a sub-folder under the **Integration→mappings** folder using your company name to store your JSON files. For example, suppose that you create a JSON mapping file named `contact-1.0.mapping.json`. You then store this file in the following location in Studio:

configuration→config→Integration→mappings→mycompany

Important

The JSON mapping schema must directly reflect the JSON API schema. This means that for every property you define on a JSON object in a `*.schema.json` file, there must an equivalent property for the same JSON object in the associated `*.mapping.json` file.

The mapping schema

The mapping schema file looks similar to the following JSON definition.

```
{
  "schemaName": "...",

  "mappers": {
    "Object": {
      "schemaDefinition": "..."

      "root": "..."

      "properties": {
        "propertyOne": {
          "path": "..."
        },
        "propertyTwo": "..." {
          "path": "..."
        }
      }

    }

  }

}
```

The example code inserts spaces for readability. The schema requires the use of the '`schemaName`' and '`mappers`' element. The use of the '`combine`' element is optional. You must supply real values for variables in italic font.

Working with JSON objects

The runtime JSON API that you use to consume or produce JSON data is based on a PolicyCenter `JsonObject` object. The `JsonObject` object is basically a `Map<String, Object>`, the values for which can be one of the following:

- Scalar POJO objects
- Other `JsonObject` objects
- Lists of scalars or `JsonObject` objects

These scalar values are the actual deserializer value, meaning `BigDecimal` objects or `TypeKey` objects, rather than just JavaScript primitive types. Thus, after the REST framework receives a request, the framework deserialization and validation of input happens in a single step. If the input does not validate against the schema or it is impossible to deserialize the input, the framework returns a 400 "bad input" error and does not invoke the API handler class.

It is possible to manipulate JSON input by doing the following:

- Use standard map operations such as `get`, put, and `containsKey`
- Use a generated schema wrapper class

The following code examples illustrate these concepts.

Example JSON manipulation with JsonObject

The following example code illustrates how to create a Gosu function to manipulate a JSON object.

```
// Function with body of type JsonObject
function startBatchProcess(body: JsonObject): Long {
  var processId: ProcessID
  var processName = body.get("processType") as String
  var args = body.get("args") as List<String>
  ...
}

// Function returns a JsonObject
function getWorkQueue(processType: String): JsonObject {
  try {
    var wQueueStatus = getDelegate().getWQueueStatus(processType)
    var jsonObject = new JsonObject()
    if (wQueueStatus != null) {
        jsonObject.put("processType", wQueueStatus.getQueueName())
        jsonObject.put("numActiveExecutors", wQueueStatus.getNumActiveExecutors())
        jsonObject.put("numOfActiveWorkItems", wQueueStatus.getNumActiveWorkItems())
      }
      return jsonObject
  } catch (illegalArgumentException: IllegalArgumentException) {
    throw new NotFoundException("Invalid process type " + processType)
  }
```

Notice the following:

- The `startBatchProcess` method takes a single `JsonObject` object as input.
- The `startBatchProcess` method performs standard map `get` operations on the passed-in JSON object.
- The `getWorkQueue` method returns a `JsonObject` object.
- The `getWorkQueue` method performs standard map `put` operations to create the JSON object returned by the method.

Example JSON manipulation with JSONWrapper class

The following example code illustrates a Gosu function that takes a generated `JSonWrapper` class as a parameter.

```
uses jsonschema.gw.pl.system.maintenance_tools.v10_0.Workqueue

...
//Function with body of type ProcessStartParams class, which is one of the generated JsonWrapper classes
function startBatchProcess(body: ProcessStartParams): Long {
  var processId: ProcessID
  var processName = body.getprocessType()
  var args = body.getargs()
  ...
}

//Function returning JsonWrapper object
function getWorkQueue(processType: String): Workqueue {
  try {
    var wQueueStatus = getDelegate().getWQueueStatus(processType)
    var workqueue = new Workqueue ()
    if (wQueueStatus != null) {
        workqueue.setprocessType(wQueueStatus.getQueueName())
        workqueue.setnumActiveExecutors(wQueueStatus.getNumActiveExecutors())
        workqueue.setnumOfActiveWorkItems(wQueueStatus.getNumActiveWorkItems())
      }
      return workqueue
  } catch (illegalArgumentException: IllegalArgumentException) {
    throw new NotFoundException("Invalid process type " + processType)
  }
}
```

Notice the following:

- The `startBatchProcess` method takes a single `JsonWrapper` object as input.
- The `startBatchProcess` method uses standard methods on `ProcessStartParams` to perform business logic.

JSON schema wrapper types

It is possible to generate schema wrapper types based on the JSON schema definitions. As you need to generate the wrapper classes explicitly (and not at build time like other code generators), you need to check the generated classes into source control. The generated classes do not represent DTO objects (Data Transfer Objects), but instead represent wrappers around a `JsonObject` object.

To make use of the generated wrapper classes, you can do one of the following:

- Create a new wrapper instance that contains a new, empty `JsonObject` object
- Wrap an existing `JsonObject` instance

For more information on the JSON schema wrapper classes, see the Guidewire JSON specification in the PolicyCenter *Integration Guide*.

Generate schema wrapper classes

JSON schema wrapper classes contain statically typed getter and setter methods on the specified object.

Procedure

1. In Guidewire studio, create a `codegen-schemas.txt` file in the following directory, if one does not already exist:

configuration→config→integration→schemas

2. In `codegen-schemas.txt`, list the fully qualified names of the schemas for which you want to generate wrapper classes, each schema on its own separate line.

   For example, enter the fully qualified name of a schema in the following form:

   ```
   gw.pl.system.cluster_tools-1.0
   ```

   Use a dot separator (.), not a slash (/) and leave off the `.schema.json` file ending.
3. On the Studio toolbar, select **Run→Edit Configurations**.

   This action opens a **Run/Debug Configurations** dialog.
4. In the dialog toolbar, click +, then select **Application** from the drop-down.

   This action adds a new unnamed, node under **Application**.
5. Enter the following data in the dialog fields:

Name	`codgen`
Main class	`com.guidewire.tools.json.JsonSchemaWrapperCodegenTool`
Working directory	Enter the root directory of your PolicyCenter installation, for example: `C:/PolicyCenter/10.0.3/`
Use classpath of module	`configuration`
Before launch: Make (including OSGI Bundles), Activate tool window	Select **Make (Including OSGI Bundles)** .

   You can accept the defaults for the remainder of the dialog fields, or, change the fields to meet your specific needs.
6. On the Studio toolbar, select **Run→codegen**.

   Studio opens a pane at the bottom of the Studio screen that logs the activity of the code generation operation. After the code generation operation completes, Studio places the generated wrapper classes in the following directory in Studio:

   configuration→src→jsonschema
7. If you are using source control for the PolicyCenter application files managed by Studio, you need to check the newly generated class files into source control as well.

Working with Swagger files

You define the Swagger schemas in `yaml` files. All Swagger schema files, including any that you create, must exist in the following location in Guidewire Studio:

configuration→config→integration→apis

You can define Swagger schema files in their own namespace within the **apis** folder. You can also nest the Swagger schema files in arbitrary folders. Swagger schemas must always have a version attached to the file name, separated by a '-' (hyphen) character. Do not use the hyphen character in any other place in a Swagger file name except the place listed. Use the following syntax in naming Swagger files:

`name-version``.swagger.yaml`

As a best practice, Guidewire recommends that you place the Swagger schemas that you create in their own namespace. For example, if you create a Swagger schema file named `contact-1.0.swagger.yaml`, place the file in the following location in Studio:

configuration→config→integration→apis→mycompany

The following string is the fully qualified path to the Swagger schema file:

```
mycompany.contact-1.0
```

Guidewire Swagger specification

For details of the Guidewire Swagger specification, see “Guidewire Swagger specification” on page 83.

Overview of the Swagger schema format

The following code sample is a Swagger schema template that you use in creating the Swagger schema for a REST API.

```
swagger: '2.0'
info:
  description: "Description of API"
  version: '1.0'
  title: "Name of API"
basePath: <base_path> # Base path prepended to every path in this API
                        # Generally takes the form /grouping/version, such as /policies/v1
x-gw-schema-import:
  <alias>: <JSON_schema_name> # Can include any number of imports, but this API requires just one
produces: # Used as the default for operations that do not explicitly declare it
- application/json
consumes: # Used as the default for operations that do not explicitly declare it
- application/json
paths:
  /<path>:
    <HTTP_operation>: # Operations are in lower-case: get, post, patch, delete
      summary: <summary> # Any text you like
      description: <description> # Any text
      operationId: <operationId> # Becomes the handler method name, must be unique within this schema
      parameters: # Possible to omit if the method has no parameters.
                  # Parameters is a list, so prefix each element with '-' (dash) to indicate that it is a list item
      - name: <foo>
        in: <query|path|body|header> # Must be either query, path, body, or header
        required: <true|false> # Defaults to false, must be set to true for path parameters
        type: <string|integer|number|boolean> # Use only with query, path, or header parameters
        schema: # Only include if using a parameter of type body
          $ref: <alias>#/definitions/<name>
      responses:
        '<code>': # Value must be enclosed in either single or double quotes.
                  # Use 200 is generic success status, 201 for "created", 204 for responses without a body
          description: <description>
          schema: # Only include for a 201 or 204
            $ref: <alias>#/definitions/<name>
```

In creating your file, replace all terms in brackets (<...>) with actual values. Also note that the # sign in the last line of the example does not indicate a comment (as the other hash marks do). Instead, it is part of the required text.

Working with schema parameters

It is possible for a resource to define, and use, any of the following parameter types:

- `path`
- `query`
- `header`
- `body`

In Swagger files, the `'in'` field on the `parameters` element defines the parameter type, `'in: path'`, or, `'in: query'`, for example.

Depending on the type of parameter, it is possible to specify a constraint that ensures the request conforms to expected schema definition.

The Guidewire x-gw-type property

In addition to the standard parameter options specified in the Swagger schema 2.0, Guidewire supports a custom `x-gw-type` property for the following parameter types:

- `path`
- `query`
- `header`

The value of a `x-gw-type` property must be a typecode from a valid PolicyCenter typekey. The following Swagger code snippet illustrates this concept.

```
/destinations/{destinationId}/retry:
    post:
      summary: "Retry messages for a destination that had errors"
      description: ''
      operationId: retryMessages
      parameters:
      - $ref: "#/parameters/destinationId"
      - name: retryLimit
        in: query
        required: false
        type: integer
        format: int32
      - name: category
        in: query
        required: false
        type: string
        x-gw-type: typekey.ErrorCategory
      ...
```

chapter 4

Implementing a REST API

After you design your REST API (and create the necessary Swagger and JSON files), you then need to implement the handler classes and methods that perform the actual work of the API. Implement any API handler classes that you create in Gosu, as method binding relies on the actual names of the method parameters, which Java compilation does not preserve. However, your Gosu class can subsequently delegate to a Java helper, if desired.

Every REST operation ultimately binds to a particular method on a particular class. You bind a class to an operation by naming the necessary class or classes in an `x-gw-apihandlers` property on the operation, path, or root document. Guidewire recommends that you first attempt to bind the class to the operation, then to the path, and finally to the root document. This means that the REST API framework uses the property definition on the root object only if the operation, then the path, do not specify the property explicitly.

Invocation of the handler class method

The REST framework invokes the method associated with the operation only after it performs the following tasks:

- Authenticates the API request
- Validates the request
- Deserializes the data inputs

If the input is bad at any step along this process, the REST framework does not invoke the handler method.

Constraints on handler classes

There are few constraints for designing an API handler class, except the following:

- Each custom handler class must have a no argument constructor as the REST framework instantiates the handler class on every request.
- Each method name must match the `operationId` value of the operation that called the method.
- Each method parameter must be of type `RequestContext`, or, must be based on a parameter defined for the method as configured in the Swagger schema.

Other than these constraints, you can structure a handler class however you like. Guidewire intentionally designs the REST framework to have minimal binding between the handler class and the REST framework.

Handler class methods

In order for the REST API framework to invoke a particular class method in a customer handler class, the method must conform to a specific structure:

- The method must be public.
- The method must be non-static.
- The method name must match the value of the `operationId` property for the Swagger operation.
- The method parameters must be of type `RequestContext`, or, must have a name and type that matches a parameter declared on the operation configuration.
- The method return type must be either `void`, `gw.api.rest.Response`, or some type that the framework can serialize out.
- The class containing the method must contain a no-argument constructor

If the `operationId` is not a valid identifier for purposes of determining the expected method name, the REST framework does the following:

- It prepends a leading '_' character to the method name if the first character is not a valid identifier start.
- It replaces all other illegal characters with '_'.

Specifying handler classes

You specify the set of possible handler class for an API by either of the following ways:

- Specify the set of handler classes directly on the operation using the `x-gw-apihandler` property.
- Specify the list of handler classes on the `x-gw-apihandler` property of the root object and not at the operation level. The operation then uses the set of handler classes inherited from the root object if you do not specify the handler classes at the operation level.

However, the REST framework does not propagate the `x-gw-apihandler` value on the root object across files during file combination. The defaults on the root object apply only to operations defined in the original file. Any value set for the `x-gw-apihandler` property at the operation level overrides the value of this property set on the root object.

Handler method parameters

Each parameter to the API handler method must be one of the following:

- A `RequestContext` object
- A parameter declared on the Swagger schema for the associated operation

Guidewire does not require that a handler method have a method parameter for every possible operation parameter. The REST framework can retrieve operation parameter values at runtime by name from the runtime `RequestContext` object if there are no explicit arguments to the handler method.

Parameters of type RequestContext

If the method parameter is of type `RequestContext` (regardless of the parameter name), the framework passes in the runtime `RequestContext` object. The runtime `RequestContext` object gives the handler method access to information about the runtime context of the request such as the following:

- The raw `HttpServletRequest` servlet container object, which can provide raw request information that is not accessible in other ways.
- Headers and path parameters by name, in either deserialized form (if explicitly listed in the schema) or raw form. This is often helpful if writing common infrastructure shared across multiple API endpoints.
- The metadata about the request being served such as the SwaggerOperation object, path template, or the fully-qualified name of the request API.
- The negotiated content type and other context information.

See “The RequestContext object” on page 75 for more information.

Other parameter types

All method parameters that are not of type `RequestContext` must have the following characteristics:

- The parameter name must match the name of a Swagger operation parameter.
- The parameter type must match to something the REST framework can deserialize.

For parameters other than the `body` parameter, the type of the method parameter must match the runtime type of the operation parameter as specified by the following properties:

- `type`
- `format`
- `x-gw-type`

If the parameter defines a set of items, the runtime type becomes a list of such objects (`List[`*`objects`*`]`).

Parameters of type body

The REST framework treats the `body` parameter as a special case, as it is possible to deserialize the request `body` data to many different runtime types. If the operation consumes the `application/json` media type, and no other types, then the `body` parameter can be any of the following types:

- `byte[]`
- `String`
- `JsonObject`
- Any subtype of `JsonWrapper` (which are generated schema wrapper types)

If the operation consumes multiple media types, or does not consume `application/json`, then the `body` parameter can be only of type `byte[]` or `String`.

Method return types

A handler method can return any of the following types.

`void`

If an operation returns a 204 response code, the method handler can have no return value. This indicates that the method has no response payload. In this case, the handler method can have a `void` return type, If the return type is `void`, the Swagger operation must not declare any `produces` type.

`String` or `byte[]`

For a return type other than JSON, the handler method can return a `String` or `byte[]` array. The REST framework then writes out as the raw string or bytes as the response.

`JsonObject` or `JsonWrapper` subtype

An operation that produces JSON-formatted data can return a `JsonObject` or any generated schema wrapper type. The method serializes the `JsonObject` object according to the schema (if any) associated with the operation's 2xx response code. For wrapper subtypes, the framework unwraps the wrapper and then serializes the object as JSON.

`TransformResult`

An operation that produces JSON-formatted date and that uses an Integration Mapping to produce that data can return a `TransformResult` directly.

`Response`

If the method handler needs more explicit control over the response's status code, or adds custom headers to the response, the method handler can return a `Response` object that combines the desired status code, response body, and headers.

Any other scalar type

The method invokes the `toString` method on the object.

Working with GET and POST operations

The following examples illustrate how to work with Swagger `GET` and `POST` operations. The examples also show how to bind the Swagger operation to methods on the associated handler class.

Working with GET operations

A `GET` operation does not have a request body. The following Swagger fragment defines a typical `GET` operation on an example `/contacts` API endpoint.

```
/contacts/{contactId}/addresses:
  get:
    summary: "Returns the list of addresses for a given contact"
    description: "Returns the list of addresses for a given contact"
    operationId: getContactAddresses
    produces:
    - application/json
    parameters:
    - name: contactId
      in: path
      required: true
      type: string
    - name: limit
      in: query
      type: integer
      minimum: 1
      maximum: 100
    - name: offset
      in: query
      type: integer
      miniumum: 0
    responses:
      '200':
        description: "Successful creation"
        schema:
          $ref: "contact#/definitions/AddressList"
```

In the code snippet, notice the following:

- The defined operation is a `GET` operation.
- The value of `operationId` is `getContactAddresses`.
- The operation produces JSON objects.
- The operation defines three parameters, `contactId`, `limit`, `offset`. The three defined parameters become arguments to method `getContactAddresses` on API handler class.

The following Gosu fragments illustrates the necessary functions on the handler class for the defined `GET` operation.

```
public function getContactAddresses(contactId : String, limit : Integer, offset : Integer) :
gw.pc.contact.v1_0.AddressList {

  var addresses = queryAddresses(contactId, limit, offset)
  var addressListJson = new gw.pc.contact.v1_0.AddressList()

  for (address : addresses) {
    addressListJson.addToAddresses(addressToJson(address))
  }
  return addressListJson
}


private function queryAddresses(contactId : String, limit : Integer, offset : Integer) : IQueryBeanResult<Address> {
  // Business logic
}


private function addressToJson(address : Address) : gw.pc.contact.v1_0.Address {
  var addressJson = new gw.pc.contact.v1_0.Address()
  addressJson.AddressLine1 = address.AddressLine1
  // More business logic
  return addressJson
}
```

In the code snippet, notice the following:

- The `getContactAddresses` method name matches the `POST` `operationId` value.
- The method arguments are parameters on the `GET` operation: `contactId`, `limit`, and `offset`.
- The method returns an address list as a JSON object.

Working with POST operations

The following Swagger fragment defines a typical `POST` operation on an example `/contacts` API endpoint.

```
/contacts:
  post:
    summary: "Create a contact"
    description: "Creates a new Contact entity and optionally sends it out to AddressBook"
    operationId: createContact
    consumes:
    - application/json
    produces:
    - application/json
    parameters:
    - name: body
      in: body
      required: true
      schema:
        $ref: "contact#/definitions/Contact"
    - name: updateAB
      in: query
      required: false
      type: boolean
    responses:
      '200':
        description: "Successful creation"
        schema:
          $ref: "contact#/definitions/Contact"
```

In the code snippet, notice the following:

- The defined operation is a `POST` operation.
- The value of `operationId` is `createContact`.
- The operation both produces and consumes JSON objects.
- The operation defines two parameters, `body` and `updateAB`, both of which become inputs to the handler class.

The following Gosu fragment illustrates the necessary functions on the handler class for the `POST` operation.

```
public function createContact(body : gw.pc.contact.v1_0.Contact, updateAB : Boolean) :
       gw.pc.contact.v1_0.Contact {
  var contactEntity = createContact(body, updateAB != null && updateAB)
  return contactToJson(contactEntity)
}

private function createContact(contactJson : gw.pc.contact.v1_0.Contact, updateAB : Boolean) : Contact {
  // Actual business logic to construct and commit the entity
  var contactEntity : Contact
  Transaction.runWithNewBundle(bundle -> {
    contactEntity = new Contact(bundle)
    contactEntity.FirstName = contactJson.FirstName
    // More business logic
  })
  return contactEntity
}

private function contactToJson(contact : Contact) : gw.pc.contact.v1_0.Contact {
  // Mapping logic to turn the contact entity back into json
  var contactJson = new gw.pc.contact.v1_0.Contact()
  contactJson.FirstName = contact.FirstName
  contactJson.PublicId = contact.PublicId
  // More business logic
  return contactJson
}
```

In the code snippet, notice the following:

- The `createContact` method name matches the `POST` `operationId` value.
- The method arguments are parameters on the `POST` operation, `body` and `updateAB`.
- The method returns the new contact as a JSON object.

Exception handling

There are multiple reasons that exceptions occur in REST API operations. For example, the exception can be one of the following:

- Exception due to problems with the input sent by the client.
- Exception due to implementation bugs.
- Exception due to other runtime failures specific to the server environment. For example, the API could not access the database, or, another user changed the data concurrently with the API request.

Status codes

PolicyCenter exception handling classes return standard HTTP status codes:

- The REST API framework roughly maps successful operations to the 2xx class of HTTP response codes.
- The REST API framework roughly translates client input errors to the 4xx class of HTTP response codes.
- The REST API framework roughly maps implementation bugs or other runtime problems to the 5xx class of HTTP response codes.

The REST API framework manages many of the types of potential errors automatically. For example, if the request does not validate against the declared parameters or the JSON schema for the request body, the framework returns a 400 response with the details of the issue. The framework does the same with authentication or authorization issues, bad content types, and other similar types of issues.

Standard error format

The standard format for error messages contains the following JSON elements.

Element	Description
`status`	HTTP status code on the response, included on the payload. For errors, the HTTP status code indicates the broad category of failure
`errorCode`	Specific code for the error, to provide more information than that provided by the status code. Typically, the error class sets this value to the name of the exception class, unless a custom exception subclass sets it explicitly to some other value.
`userMessage`, `developerMessage`	Error messages that describe the problem or issue in more detail. If the error message specifies both a `userMessage` and a `developerMessage`, Guidewire recommends the following: • Phrase `userMessage` in such a way as to make it possible to surface the error message directly to the PolicyCenter user. • Phrase `developerMessage` so as to be useful to the developer creating the client that calls the REST API.
`details`	Error message can also contain an array of additional details that describe the problem. The exact properties on the `details` object can vary depending on the exception type

The following example illustrates an error message for a request that failed. In this case, the request payload was missing the '`subject`' property on the root JSON object, which the JSON specification specifically requires.

```
{
  "status": 400,
  "errorCode": "gw.api.rest.exceptions.BadInputException",
  "userMessage": "The request parameters or body had issues",
  "developerMessage": "The request parameters or body had issues. See the details elements for exact details of the
problems.",
  "details": [
    {
      "message": "The 'subject' property is required",
      "properties": {
        "parameterName": "body",
        "parameterLocation": "body",
        "lineNumber": 1
      }
    }
  ]
}
```

Exception handling classes

Whether any exception thrown from within the handler method itself translate to HTTP response codes depends on whether the exception class implements the `HasErrorInfo` interface:

- If a thrown exception implements the `HasErrorInfo` interface, the framework constructs the response based on the status code and error details on the exception.
- If a thrown exception does not implement the `HasErrorInfo` interface, the framework translate the exception into a generic 500 "internal server error" HTTP response.

A handler method can use any exception that implements the `gw.api.exception.HasErrorInfo` interface, including custom exception subtypes. However, in general, the exception classes defined in the `gw.api.rest.exceptions` package are suitable for mapping to common error cases. For example:

`NotFoundException`	A method can throw a `NotFoundException` exception if the REST framework cannot find an entity ID referenced from a path parameter in the database.
`BadInputException`	A method can throw a `BadInputException` exception if the input fails some additional validation performed by the method handler. In this way, it is possible to add additional validations in the method that you cannot declare in the schema. For example, you can throw a `BadInputException` exception that translates to a 400 error.

Exception handler classes

Any exception class that you create must do one of the following:

- Extend the `RestExceptionWithErrorInfo` class
- Implement the `HasErrorInfo` interface

Any API handler class that you create must check for error cases that the REST framework does not handle automatically. The handler class must specifically throw the appropriate exceptions in cases in which the error is caused by the input on the client side.

Stack traces

The API client response returns a stack trace for internal server errors, if the server is not in production mode. Custom exception classes can trigger the inclusion of stack traces by extending the `RestExceptionWithErrorInfo` class and calling the parent (super) constructor with the Boolean `includeCause` argument set to `true`:

```
super(statusCode, userMessage, developerMessage, details, includeCause)
```

For example:

```
super(500, userMessage, developerMessage, details, true)
```

In the base configuration, only the `InternalServerErrorException` class passes `true` for this argument.

Rewriting error messages

It is possible to rewrite error information using the `rewriteErrorInfo` method on the `IRestDispatchPlugin` implementation class.

Handler classes that return a Response object

It is possible for an API handler class to return a `Response` object that explicitly sets the status code to a 4xx or 5xx status code. While this is legal to do, using a `Response` object to set the error code also makes it harder to ensure that the error format is consistent across different classes of errors.

In such cases, the framework considers the request to be successful as no exceptions were thrown and the handler class completed successful. This causes the handler class to invoke the `rewriteResponse` method on the `IRestDispatchPlugin` class instead of the `rewriteErrorInfo` method. In general, Guidewire recommends that you use a custom exception class that implements the `HasErrorInfo` interface and then throw an exception, rather than returning an explicit `Response` object with a 4xx or 5xx status code.

Exception handling examples

BadInputException exception example

The following sample code illustrates a class method that throws a bad input exception.

```
function startBatchProcess(body: JsonObject): Long {
  var processType = body.get("processType") as String
  var batchProcessNames = getDelegate().getValidBatchProcessTypes()
  if (processType == null || !batchProcessNames.contains(processType)) {
    throw new BadInputException("Bad process type " + processType)
  }
  .....
}
```

ConflictingResourceException exception example

The following sample code illustrates a class method that handles a conflicting resource exception.

```
function createContact(body : gw.pc.contact.v1_0.Contact, updateAB : Boolean) : gw.pc.contact.v1_0.Contact {
  if(contactAlreadyExists(body)){
    throw new ConflictingResourceException("Contact already exists", null, null)
  }
  ...
}

private function contactAlreadyExists(contactJson : gw.pc.contact.v1_0.Contact) : Boolean {
  // Business logic to look up contact
  ...
}

private class ConflictingResourceException extends RestExceptionWithErrorInfo {
  protected construct(userMessage: String, developerMessage: String, details: List<RequestErrorDetails>) {
    super(409, userMessage, developerMessage, details)
  }
}
```

Notice the following:

- Method `createContact` throws a `ConflictingResourceException` exception if the contact to add already exists.
- Private class `ConflictingResourceException` defines the message processing logic.

Package gw.api.exceptions.rest

Guidewire provides a number of exception classes in the `gw.api.exceptions.rest` package. This package contains the following super class that many of the other classes in the packaged extend:

- `RestExceptionWithErrorInfo`

The following classes in `gw.api.exceptions.rest` package provide numeric return values.

Class	Return value	Meaning
`BadInputException`	400	Bad Request

Class	Return value	Meaning
`CorsException`	403	Forbidden
`InternalServerErrorException`	500	Internal Server Error
`NotAcceptableException`	406	Not Acceptable
`NotAuthorizedException`	403	Forbidden
`OperationNotSupportedException`	405	Method Not Allowed
`RestAuthenticationException`	401	Unauthorized
`ServiceUnavailableException`	-	Service Unavailable
`UnsupportedContentTypeException`	415	Unsupported Media Type

Note: A 403 error means that the user is not authorized for the requested resource. If you do not want the user to even be aware of the resource, use `NotFoundException` (404) instead.

The `gw.api.exceptions.rest` package also contains the following utility classes, called by `RestExceptionWithErrorInfo`:

- `RequestErrorDetails`
- `RequestErrorInfo`

chapter 5

Publishing a REST API

Merely creating a `swagger.yaml` file does not, by itself, create a network endpoint. To create an API endpoint, you must explicitly publish any APIs that you create by listing each API in file `published-apis.yaml`, located in Studio in the following directory:

configuration→config→Integration→apis

Guidewire requires that you publish REST APIs explicitly in file `published-apis.yaml` for the following reasons:

1. It gives you total control over the network exposure of Guidewire PolicyCenter. The REST API framework does not publish an API merely because the API is part of the base PolicyCenter application.
2. It ensures that if you extend the Guidewire base API Swagger files, you can publish the resulting APIs using your own namespace and versioning schema.
3. It ensure that you do not accidentally publish an API. Thus, it ensures that test, or development, APIs do not accidentally leak into the network.

File published-apis.yaml

Merely creating the API schema files does not, by itself, create a network endpoint. To create an API endpoint, you must explicitly publish any APIs that you create by listing each API in file `published-apis.yaml`, located in Studio in the following directory:

configuration→config→Integration→apis→...

Guidewire requires that you publish REST APIs explicitly in file `published-apis.yaml` for the following reasons:

1. It gives you total control over the network exposure of Guidewire PolicyCenter. The REST API framework does not publish an API merely because the API is part of the base PolicyCenter application.
2. It ensures that if you extend the Guidewire base API Swagger files, you can publish the resulting APIs using your own namespace and versioning schema.
3. It ensure that you do not accidentally publish an API. Thus, it ensures that test, or development, APIs do not accidentally leak into a production network.

File `published-apis.yaml` has the following form.

```
apis:
  - name: gw.pl.framework.api_list-1.0
  - name: fully.qualified.path.to.apifile

defaultTemplate:
  - name: gw.core.pc.framework.v1.dev_template-1.0
  - name: fully.qualified.path.to.templatefile
```

API templates

It is possible for the server environment to affect some aspects of how the REST framework exposes an API to the network. A primary use case is the security section of the Swagger documents. If a server allows for HTTP Basic Authentication, it is useful to include the appropriate declarations in the `swagger.yaml` file so that client tools such as Swagger UI or Postman can automatically interpret that information and give the user the option to specify authentication headers. However, whether the server supports Basic Authentication can depend upon whether the server is a development or production server. It is also possible that you want to expose the `/swagger.json` endpoint that self-documents the REST APIs, but, only for development or test or sandbox systems and not in production environment.

To meet these needs, you can specify an API template that the REST framework includes with the API at runtime. You can specify that an API template affects only a single API, or, you can specify a template as the global default. Both of the API declarations and default template are sensitive to the `server` and `env` properties in a similar manner to other configuration files. Thus, you can specify that the REST framework publish an API to a specific server only, or, only if the server is operating in a specific server mode.

Publishing to different environments

The following example illustrates how to publish an API to a specific environment.

```
apis:
- name: gw.pc.systemtools.system_tools-10.0
  env: dev
- name: gw.pl.framework.api_list-1.0
defaultTemplate:
- name: gw.pl.framework.dev_template-1.0
- name: customer.framework.prod_template-1.0
  env : dev
```

In this example, notice the following:

- The `apis` element lists multiple API names. The first API on the list is active in a `dev` environment only. The last API (`gw.pl.framework.api_list-1.0`) is active in all environments.
- The default template is `gw.pl.framework.dev_template-1.0`. It is active in all environments except a development (`dev`) environment.
- The template to use in a development (`dev`) environment is `customer.framework.prod_template-1.0`.

File published-apis.yaml document objects

File `published-apis.yaml` contains the following document objects:

- Root object
- API object
- Default template object

Root object

Property	Type	Required	Usage
`apis`	*API object*	Yes	The set of APIs to publish. It is possible for the same API name to appear multiple times in the list. At runtime, the REST framework chooses the configuration with the most specific match based on the `env` and `server` properties. It is a configuration error for multiple entries to have the same specificity at run-time. The set of paths for all published APIs must be non-overlapping as well. If you need to publish two different versions of the same API at the same time, those APIs must define different `baseUrl` values to ensure uniqueness. For exam-ple, use `/users/v1.0` and `/users/v1.0.1` to make the APIs different.
`defaultTemplate`	*Default template object*	Yes	The set of default templates to use with the listed APIs. However, at runtime, the REST framework uses only a single entry in the list, based on the server and envi-ronment properties.

Property	Type	Required	Usage
			The REST framework uses the default template for any entry in the APIs list that does not explicitly define a template.

API object

Property	Type	Required	Usage
`name`	`string`	Yes	The fully-qualified name of a Swagger schema to publish.
`template`	`string`	No	The fully-qualified name of a Swagger schema to include as the template for this API. If this value is `null`, the REST framework uses the default template.
`published`	`boolean`	No	The Boolean value indicates whether the REST framework is to publish the API. The default value is `true`. You can use this property in either of the following ways: • If you choose to use the `env` and `server` properties, set this property value to `false` to explicitly indicate that a given API must not be published in a particular environment or server. • If your PolicyCenter installation uses property substitution, you can set the `published` property to an externalized property. In that way you can use property substitution to set the value to `true` or `false`.
`env`	`string`	No	The environment to use in attempting to match this configuration element. If the file specifies the same API multiple times, the REST framework uses the most specific match based on the API `env` and `server` properties.
`server`	`string`	No	The server to use in attempting to match this configuration element. If the file specifies the same API multiple times, the REST framework uses the most specific match based on the API `env` and `server` properties.

Default template object

Property	Type	Required	Usage
`name`	`string`	Yes	The fully-qualified name of the template to use as the default for APIs listed under `apis` that do not explicitly specify a `template` property.
`env`	`string`	No	The environment to use in attempting to match this configuration element. If the file specifies the same API multiple times, the REST framework uses the most specific match based on the API `env` and `server` properties.
`server`	`string`	No	The server to use in attempting to match this configuration element. If the file specifies the same API multiple times, the REST framework uses the most specific match based on the API `env` and `server` properties.

Default API templates

In the base configuration, Guidewire provides an example of a default API template called `dev_template-1.0`, located in the following location in Studio:

configuration→config→Integration→apis→gw→pl→framework

The default template includes the following pieces.

`gw.pl.framework.basic_auth-1.0`

Adds the Swagger security specification that supports HTTP Basic Authentication, necessary so that Swagger UI and other tools can provide an appropriate authentication pop-up.

`gw.pl.framework.standard_definitions-1.0`

Adds standard definitions for all custom headers supported by the REST framework and all standard error codes that the framework returns.

`gw.pl.framework.api_docs-1.0`

Adds a `/swagger.json` endpoint to every API for retrieving the API's Swagger schema at runtime.

Guidewire deliberately implements each of these pieces as its own schema. This permits you to choose to include any or all of these pieces in creating your own templates.

Changes to file published-apis.yaml

Guidewire permits changes to file `published-apis.yaml` during a rolling (configuration) upgrade. The configuration verification mechanism:

- Reports both additions and removals to the set of APIS as `'compatible'`.
- Reports changes to the server, environment, or template values as `'equal'`.

Be aware that it is possible to add or remove APIs by doing any of the following:

- Add and remove API entries in file `published-apis.yaml`.
- Change the value of a property on a published API entry.

Guidewire permits these kinds of changes in a rolling upgrade so as to make it possible to enable a new API during a configuration upgrade, as well to provide the ability to remove an API that is no longer in use.

Forming the API URL

Each API listed in `published-apis.yaml` is addressable through the REST servlet. You form the path for each Swagger operation by adding the API base path to the path for the operation. For example, suppose that an API has a base path (`basePath`) of `/users/v1`. In this case, a `GET /users/{userId}` operation then has a URL of `/users/v1/users/{userId}` relative to the servlet.

The full URL at runtime depends upon the application server URL, the servlet context, and REST servlet mapping. Thus, the full URL format looks something similar to the following:

```
<schema> + <appserver URL> + ":" + <appserver port> + <servlet context>
    + <REST servlet path> + <API basePath> + <operation path>
```

For example, suppose that you have the following values for a PolicyCenter installation:

Application server	`mycompany.com`
Port	`8180`
Servlet context	`/pc`
Default REST servlet mapping	`/rest/*`

Using these values, the following string defines the full URL for the `GET` operation for a user with ID `12345`:

```
https://mycompany.com:8180/pc/rest/user/v1/users/12345
```

chapter 6

Creating a simple Activities API

The following topics describe how to create a simple Activities API using the PolicyCenter REST API framework. The example Activities API provides the following functionality:

- Retrieves a list of activities
- Retrieves a single activity
- Creates a new activity
- Update an existing activity

API design

As you think about what it takes to build an Activities REST API, consider the following:

- What HTTP operations does the API need to provide the required functionality
- What types of endpoints (URLs) and query parameters does the API need
- What kind of JSON inputs and outputs does the API need

For inspiration, you can work either from either the client perspective or the system perspective, or ideally a mix of both. As you work in constructing the API, think about the following items:

- What fields exist on the PolicyCenter `Activity` entity?
- What domain properties exist on the `Activity` entity?
- What fields does the PolicyCenter application expose on the desktop pages, or, in the context of a policy, an account, or a job.

Understanding the Activity API data model

There are a few typical API operations that you would usually want to implement. These operations include the following:

- Retrieving a collection of resources
- Retrieving a single resource
- Creating a new resource
- Updating a resource

The example Activities API provides an example of the types of HTTP operations and endpoints that you would most likely want to create.

API functionality	HTTP operation and endpoint
Retrieve a list of activities	`GET /activities`
Retrieve a single activity	`GET /activities/{activityId}`
Create a new activity	`POST /activities`
Update an activity	`PATCH /activities/{activityId}`

You design the inputs and outputs to the Activities API operations in JSON schema files.

Example GET /activities response

The following code sample illustrates what a possible JSON response to a `GET /activities` operation could look like. Understanding what you want the output to look like helps you determine how to structure the JSON schema file.

```
[
  "assignedUser": {
    "displayName": "Alice Applegate",
    "publicId": "pc:305",
    "username": "aapplegate"
  },
  "escalated": false,
  "escalationDate": "2018-04-23T16:22:23.976Z",
  "mandatory": true,
  "priority": "high",
  "publicId": "pc:203",
  "status": "open",
  "subject": "Action Required",
  "targetDate": "2018-04-23T16:22:23.976Z"
]
```

Example GET /activities/{activityId} response

The following code sample illustrates what a possible JSON response to a `GET /activities/{activityId}` operation for activity `pc:203` could look like. Understanding what you want the output to look like helps you determine how to structure the JSON schema file.

```
{
  "assignedUser": {
    "displayName": "Alice Applegate",
    "publicId": "pc:305",
    "username": "aapplegate"
  },
  "description": "...",
  "mandatory": false,
  "priority": "normal",
  "publicId": "pc:101",
  "relatedAccount": {
    "accountNumber": "C000212105",
    "displayName": "C000212105",
    "publicId": "pc:ds:1"
  },
  "relatedJob": {
    "displayName": "SUB00000002",
    "jobNumber": "SUB00000002",
    "jobType": "Submission",
    "publicId": "pc:11"
  },
  "relatedPolicy": {
    "displayName": "pc:6",
    "publicId": "pc:6"
  },
  "relatedPolicyPeriod": {
    "displayName": "6996053459, 01/18/2017, 01/18/2018, SUB00000002",
    "policyNumber": "6996053459",
    "publicId": "pc:11"
  },
  "status": "open",
  "subject": "New subject 2",
  "targetDate": "2018-04-18T23:05:53.981Z"
}
```

Setting basic API functionality

Creating the Activity JSON schema

The Guidewire REST API framework uses JSON schema files to define the input and output for the REST APIs. Place the JSON schema file that you create in the following directory in Guidewire Studio **Project** window:

configuration→config→Integrations→schemas→mc→activityapi

The example Activities API uses `mc/activityapi` as the name space (base path) for the API configuration files that you need to create in sub-directories of the Studio **Integration** directory. The file names of the API configuration files that you create must also contain an ending `-1.0` designation. For example, name the JSON file that you create something similar to the following example:

`activityAPI-1.0.schema.json`

Use the following fully qualified path to the JSON schema file in code:

`mc.activityapi.activityAPI-1.0`

IMPORTANT It is important to generate the JSON schema wrapper classes before you construct any handler class that uses those JSON objects. See "JSON schema wrapper types" on page 27 for more information.

Example JSON schema for the Activity API

The example JSON schema defines the following JSON objects:

- `ActivityDetail` object
- `ActivitySummary` object
- `AssignedUser` object
- `NewActivity` object
- `RelatedContact` object
- `RelatedAccount` object
- `RelatedJob` object
- `RelatedPolicy` object

The following code sample is an example of the JSON schema for the Activities API.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "definitions": {
    "ActivityDetail" : {
      "type" : "object",
      "properties" : {
        "approvalRationale" : {
          "type" : "string"
        },
        "assignedUser" : {
          "$ref" : "#/definitions/AssignedUser"
        },
        "description" : {
          "type" : "string"
        },
        "escalationDate" : {
          "type" : "string",
          "format" : "date-time"
        },
        "mandatory" : {
          "type" : "boolean"
        },
        "priority" : {
          "type" : "string",
          "x-gw-type" : "typekey.Priority"
        },
        "publicId" : {
          "type" : "string"
        },
        "relatedAccount" : {
          "$ref" : "#/definitions/RelatedAccount"
        },
        "relatedContact" : {
          "$ref" : "#/definitions/RelatedContact"
        },
        "relatedJob" : {
          "$ref" : "#/definitions/RelatedJob"
        },
        "relatedPolicy" : {
          "$ref" : "#/definitions/RelatedPolicy"
        },
        "relatedPolicyPeriod" : {
          "$ref" : "#/definitions/RelatedPolicyPeriod"
        },
        "status" : {
          "type" : "string",
          "x-gw-type" : "typekey.ActivityStatus"
        },
        "subject" : {
          "type" : "string"
        },
        "targetDate" : {
          "type" : "string",
          "format" : "date-time"
        }
      }
    },
    "ActivitySummary" : {
      "type" : "object",
      "properties" : {
        "assignedUser" : {
          "$ref" : "#/definitions/AssignedUser"
        },
        "escalated" : {
          "type" : "boolean"
        },
        "escalationDate" : {
          "type" : "string",
          "format" : "date-time"
        },
        "mandatory" : {
          "type" : "boolean"
        },
        "priority" : {
          "type" : "string",
          "x-gw-type" : "typekey.Priority"
        },
        "publicId" : {
          "type" : "string"
        },
        "status" : {
          "type" : "string",
          "x-gw-type" : "typekey.ActivityStatus"
        },
        "subject" : {
          "type" : "string"
        },
        "targetDate" : {
          "type" : "string",
          "format" : "date-time"
        }
      }
    },
    "AssignedUser" : {
      "type" : "object",
      "properties" : {
        "displayName" : {
          "type" : "string"
        },
        "publicId" : {
          "type" : "string"
        },
        "username" : {
          "type" : "string"
        }
      }
    },
    "NewActivity" : {
      "properties" : {
        "activityPattern" : {
          "type" : "string"
        },
        "accountNumber" : {
          "type" : "string"
        },
        "subject" : {
          "type" : "string"
        },
        "description" : {
          "type" : "string"
        },
        "jobNumber" : {
          "type" : "string"
        },
        "priority" : {
          "type" : "string",
          "x-gw-type" : "typekey.Priority"
        },
        "mandatory" : {
          "type" : "boolean"
        },
        "targetDate" : {
          "type" : "string",
          "format" : "date-time"
        },
        "escalationDate" : {
          "type" : "string",
          "format" : "date-time"
        }
      }
    },
    "RelatedAccount" : {
      "type" : "object",
      "properties" : {
        "accountNumber" : {
          "type" : "string"
        },
        "displayName" : {
          "type" : "string"
        },
        "publicId" : {
          "type" : "string"
        }
      }
    },
    "RelatedContact" : {
      "type" : "object",
      "properties" : {
        "displayName" : {
          "type" : "string"
        },
        "publicId" : {
          "type" : "string"
        }
      }
    },
    "RelatedJob" : {
      "type" : "object",
      "properties" : {
        "displayName" : {
          "type" : "string"
        },
        "jobNumber" : {
          "type" : "string"
        },
        "jobType" : {
          "type" : "string",
          "x-gw-type" : "typekey.Job"
        },
        "publicId" : {
          "type" : "string"
        }
      }
    },
    "RelatedPolicy" : {
      "type" : "object",
      "properties" : {
        "displayName" : {
          "type" : "string"
        },
        "publicId" : {
          "type" : "string"
        }
      }
    },
    "RelatedPolicyPeriod" : {
      "type" : "object",
      "properties" : {
        "displayName" : {
          "type" : "string"
        },
        "policyNumber" : {
          "type" : "string"
        },
        "publicId" : {
          "type" : "string"
        }
      }
    }
  }
}
```

Creating the Activity API Swagger schema

The Guidewire REST API framework uses Swagger (`*.swagger.yaml`) files to define the API schema. Place the Swagger schema file that you create in the following directory in Guidewire Studio **Project** window:

configuration→config→Integrations→apis→mc→activityapi

The example Activities API uses `mc/activityapi` as the name space (base path) for the API configuration files that you need to create in sub-directories of the Studio **Integration** directory. The file names of the API configuration files that you create must also contain an ending `-1.0` designation. For example, name the Swagger file that you create something similar to the following example:

`activityAPI-1.0.swagger.yaml`

Use the following fully qualified path to the JSON schema file in code:

`mc.activityapi.activityAPI-1.0`

Example Swagger schema for the Activity API

The following code sample is a working example of the Swagger schema for the Activities API. Notice that the example does not yet contain all of the required HTTP operations for this API.

```
swagger: '2.0'
info:
  description: "APIs for manipulating activities"
  version: '10.0'
  title: "Activities API"
basePath: /mc/activityapi
x-gw-schema-import:
  activities : mc.activityapi.activityAPI-1.0
produces:
  - application/json
consumes:
  - application/json
paths:
  /activities:
    get:
      summary: "Returns a list of activities"
      description: "Returns a list of activities"
      operationId: getActivities
      responses:
        '200':
          description: "Returns a list of activities"
          schema:
            type: array
            items:
              $ref: "activities#/definitions/ActivitySummary"
    post:
      summary: "Creates a new activity"
      description: "Creates a new activity"
      operationId: createActivity
      parameters:
        - name: body
          in: body
          required: true
          schema:
            $ref: "activities#/definitions/NewActivity"
      responses:
        '200':
          description: "Returns the details for the newly-created activity"
          schema:
            $ref: "activities#/definitions/ActivityDetail"
  /activities/{activityId}:
    get:
      summary: "Returns details for a single activity"
      description: "Returns details for a single activity"
      operationId: getActivity
      parameters:
        - $ref: "#/parameters/activityId"
      responses:
        '200':
          description: "Returns details for a single activity"
          schema:
            $ref: "activities#/definitions/ActivityDetail"
parameters:
  activityId:
    name: activityId
    in: path
    type: string
    required: true
```

IMPORTANT Ensure that you use correct indention to represent the schema hierarchy. Otherwise, Studio indicates errors and the API does not function properly.

Publishing the Activities API

To make an API active, you must publish the API by adding the API information to file `published-apis.yaml`, located in the following directory in the Studio **Project** window:

configuration→config→Integrations→apis

In the base REST API framework, file `published-apis.yaml` takes the following form:

```
apis:
- name: gw.pl.framework.api_list-1.0
- name: gw.pl.system.server_tools-1.0
defaultTemplate:
- name: gw.pl.framework.dev_template-1.0
```

Notice that the file contains two distinct areas:

- `apis` - A list of APIs to expose, with each API listing the fully qualified path to its Swagger schema file
- `defaultTemplate` - One or more default templates to use with the listed APIs

Although not shown in the file example, it is also possible to publish an API or template to a specific environment or environments. You do this by setting an `env` environment variable for the affected API or template.

Adding the Activity API to file published-apis.yaml

As file `published-apis.yaml` exists as part of the Guidewire REST API framework, you need merely to add the Activities API to the file in the correct location. The following code sample adds the fully qualified path to the Activities API schema file to file `published-apis.yaml`:

```
apis:
- name: gw.pl.framework.api_list-1.0
- name: gw.pl.system.server_tools-1.0
- name: mc.activityapi.activityAPI-1.0
defaultTemplate:
- name: gw.pl.framework.dev_template-1.0
```

Creating a basic handler class

Each REST API must have an associated handler class that performs the real work of the API. The handler class must contain one method for each HTTP operation that the Swagger schema specifies. The method name must match the `operationId` value of the associated HTTP operation. The method arguments, if any, must be one of the following:

- The method arguments must match the list of parameters defined for the HTTP operation.
- The method arguments must be of type `RequestContext`.

For example, the Activities schema defines a `GET` operation on the `/activities` endpoint with an `operationId` value of `getActivities`. Thus, your handler class must contain a method named `getActivities`.

Your initial version of the handler class must contain methods for each of the following operations. These operations match those of the example Swagger schema listed in "Creating the Activity API Swagger schema" on page 50.

Operation	Endpoint	OperationId	Class method
`GET`	`/activities`	`getActivities`	`getActivities()`
`GET`	`/activities/{activityId}`	`getActivity`	`getActivity(activityId)`
`POST`	`/activities`	`createActivity`	`createActivity(body)`

Example Activity API handler class

For the example Activities API, you need to create a file named `ExampleActivityApiHandler.gosu` in the following Guidewire Studio directory:

gsrc→mc→activityapi

The initial code is a place-holder API handler class for the Activities API. The class method bodies contain no real functionality at this point. However, it is necessary to have the stub handler class in place in order to publish and test your work so far.

```
package mc.activityapi

uses gw.api.json.JsonObject

class ExampleActivitiesApiHandler {

  function getActivities() : List<JsonObject> {
    print(">> getActivities called")
    return new ArrayList<JsonObject>()
  }

  function createActivity(body : JsonObject) : JsonObject {
    print(">> createActivity called")
    print(">> body is:\n" + body.toPrettyJsonString())
    return new JsonObject()
  }

  function getActivity(activityId : String) : JsonObject {
    print(">> getActivity called")
    print(">> activityId: " + activityId)
    return new JsonObject()
  }
}
```

In this example, each stub method does the following:

- It prints out the input parameters to the method to the console.
- It returns an empty response of an appropriate type as specified by the JSON response schema.

Specifying the API handler class in Swagger schema

You must associate your handler class with the Activities API. You do this in the Swagger schema, by adding a value for `x-gw-apihandler` to the schema, for example:

```
x-gw-apihandlers:
  mc.activityapi.ExampleActivitiesApiHandler
```

Thus, file `activityAPI-1.0.swagger.yaml` must contain an entry for `x-gw-apihandler` similar to the following:

```
swagger: '2.0'
info:
  description: "APIs for manipulating activities"
  version: '10.0'
  title: "Activities API"
basePath: /mc/activityapi
x-gw-schema-import:
  activities : mc.activityapi.activityAPI-1.0
x-gw-apihandlers:
- mc.activityapi.ExampleActivitiesApiHandler
...
```

Next steps

Recompile the configuration code and stop and restart the application server for PolicyCenter to pick up these changes.

Testing your work

At a point, you can test the work that you have done so far. To do so, you need to open Swagger UI tool by navigating to the follow URL in a browser window:

`*host*:8180/pc/resources/swagger-ui`

Replace *`host`* with the actual server name. For example, replace *`host`* with `localhost` if running the PolicyCenter application server on your local machine.

Working with Swagger UI

After you open Swagger UI, you can perform the following types of tasks:

- Set the user name and password credentials in the **Authorize** dialog.
- See the list of published APIs available on this host.
- See the Swagger schema for the currently viewed API.

To view the list of published APIs, click the first **GET** / link in the list. You then need to click **Try it out** and then **Execute**. If successful, Swagger UI returns a list of published APIs available on the application server. The list looks similar to the following example. Notice that the `docs` element lists the URL for each published API.

```
{
  "/apis": {
    "basePath": "/apis",
    "description": "Dynamically lists the APIs that are available",
    "docs": "http://localhost:8180/pc/rest/apis/swagger.json",
    "title": "API List"
  },
  "/example/v1": {
    "basePath": "/mc/activityapi",
    "description": "APIs for manipulating activities",
    "docs": "http://localhost:8180/pc/rest/mc/activityapi/swagger.json",
    "title": "Activities API"
  },
  "/system/v1/server": {
    "basePath": "/system/v1/server",
    "description": "This API is related to system server resources.\n",
    "docs": "http://localhost:8180/pc/rest/system/v1/server/swagger.json",
    "title": "System tools server API"
  }
}
```

Viewing and testing the Activities API

Copy the `docs` URL for the Activities API and paste it into the **Explore** field. After you click **Explore**, Swagger UI generates the HTTP operations for the Activities API. Use Swagger UI functionality to test each API endpoint and operation defined in the Swagger schema.

At this point, there is no backing functionality for the API operations. However, do not continue defining the Activities API if Swagger UI or the API operations generate any kind of error message. Fix any errors with your defined configuration files before continuing the API development process.

Adding more API fuctionality

Getting activities

To add functionality for the `getActivities` method of the API handler class, you need to do the following:

- Create an integration mapping file that maps the API root objects into JSON data that conforms to the JSON schema document for the Activities API.
- Update the `getActivities` method to use the JSON mapper to return the required list of activities.

Creating a JSON mapper file

Each integration mapping file can have any number of integration mappers defined in it. Each integration mapper defines a single entry point into the mapping transformation. An integration mapper takes a single input object and transforms it into JSON that matches an output schema. Each integration mapper contains mapping properties that correspond to each property in the output schema.

For the `getActivities` method, you need to create a mapper file named `activityAPI-1.0.mapping.json` in the following directory in the Studio **Project** window:

configuration→config→Integration→mappings→mc→activityapi

Your updated mapper file needs to contain code similar to the following example:

```
{
  "schemaName": "mc.activityapi.activityAPI-1.0",
  "mappers": {
    "ActivitySummary" : {
      "schemaDefinition" : "ActivitySummary",
      "root" : "entity.Activity",
      "properties" : {
        "assignedUser" : {
          "path" : "Activity.AssignedUser",
          "mapper" : "#/mappers/AssignedUser"
        },
        "escalated" : {
          "path" : "Activity.Escalated"
        },
        "escalationDate" : {
          "path" : "Activity.EscalationDate"
        },
        "mandatory" : {
          "path" : "Activity.Mandatory"
        },
        "priority" : {
          "path" : "Activity.Priority"
        },
        "publicId" : {
          "path" : "Activity.PublicID"
        },
        "status" : {
          "path" : "Activity.Status"
        },
        "subject" : {
          "path" : "Activity.Subject"
        },
        "targetDate" : {
          "path" : "Activity.TargetDate"
        }
      }
    },
    "AssignedUser" : {
      "schemaDefinition" : "AssignedUser",
      "root" : "entity.User",
      "properties" : {
        "displayName" : {
          "path" : "User.DisplayName"
        },
        "publicId" : {
          "path" : "User.PublicID"
        },
        "username" : {
          "path" : "User.Credential.UserName"
        }
      }
    }
  }
}
```

Adding functionality to handler method getActivities

Handler class `ExampleActivitiesApiHandler` defines the methods that support the Activities API. This class exists in the following directory in the Studio **Project** window:

configuration→gsrc→mc→activityapi

The `getActivities` handler method uses the `ActivitySummary` declaration to produce a `List<TransformResult>` as a result. The code for the `getActivities` handler method looks similar to the following example.

```
function getActivities() : List<TransformResult> {
  var query = Query.make(Activity)
  var resultSet = query.select()
  var mapper = JsonConfigAccess.getMapper("mc.activityapi.activityAPI-1.0", "ActivitySummary")

  return mapper.transformObjects(resultSet)
}
```

Next steps

After adding the mapper file in Studio and updating the API handler file, do the following:

- Recompile the PolicyCenter application
- Restart the application server

You need to restart the application server in order for PolicyCenter to recognize the new mapper file. In general, you need to restart the application server once only for PolicyCenter to recognize a newly added file. Thereafter, the server recognizes subsequent changes to the file.

Getting activity detail

To add functionality for the `getActivity` method of the API handler class, you need to do the following:

- Update the integration mapping file with a mapper for the `ActivityDetail` schema and related pieces.
- Update the `getActivity` handler method to load the activity with the specified `activityId` from the database and return it as a `TransformResult` object.

Adding objects to the JSON mapper file

Mapping file `activityAPI-1.0.mapping.json` exists in the following directory in the Studio **Project** window:

configuration→config→Integration→mappings→mc→activityapi

Update your mapper file and add code similar to the following example in an appropriate place:

```
"ActivityDetail" : {
  "schemaDefinition" : "ActivityDetail",
  "root" : "entity.Activity",
  "properties" : {
    "approvalRationale" : {
      "path" : "Activity.ApprovalRationale"
    },
    "assignedUser" : {
      "path" : "Activity.AssignedUser",
      "mapper" : "#/mappers/AssignedUser"
    },
    "description" : {
      "path" : "Activity.Description"
    },
    "escalationDate" : {
      "path" : "Activity.EscalationDate"
    },
    "mandatory" : {
      "path" : "Activity.Mandatory"
    },
    "priority" : {
      "path" : "Activity.Priority"
    },
    "publicId" : {
      "path" : "Activity.PublicID"
    },
    "relatedAccount" : {
      "path" : "Activity.Account",
      "mapper" : "#/mappers/RelatedAccount"
    },
    "relatedContact" : {
      "path" : "Activity.Contact",
      "mapper" : "#/mappers/RelatedContact"
    },
    "relatedJob" : {
      "path" : "Activity.Job",
      "mapper" : "#/mappers/RelatedJob"
    },
    "relatedPolicy" : {
      "path" : "Activity.Policy",
      "mapper" : "#/mappers/RelatedPolicy"
    },
    "relatedPolicyPeriod" : {
      "path" : "Activity.PolicyPeriod",
      "mapper" : "#/mappers/RelatedPolicyPeriod"
    },
    "status" : {
      "path" : "Activity.Status"
    },
    "subject" : {
      "path" : "Activity.Subject"
    },
    "targetDate" : {
      "path" : "Activity.TargetDate"
    }
  }
},
"RelatedAccount" : {
  "schemaDefinition" : "RelatedAccount",
  "root" : "entity.Account",
  "properties" : {
    "accountNumber" : {
      "path" : "Account.AccountNumber"
    },
    "displayName" : {
      "path" : "Account.DisplayName"
    },
    "publicId" : {
      "path" : "Account.PublicID"
    }
  }
},
"RelatedContact" : {
  "schemaDefinition" : "RelatedContact",
  "root" : "entity.Contact",
  "properties" : {
    "displayName" : {
      "path" : "Contact.DisplayName"
    },
    "publicId" : {
      "path" : "Contact.PublicID"
    }
  }
},
"RelatedJob" : {
  "schemaDefinition" : "RelatedJob",
  "root" : "entity.Job",
  "properties" : {
    "displayName" : {
      "path" : "Job.DisplayName"
    },
    "jobNumber" : {
      "path" : "Job.JobNumber"
    },
    "jobType" : {
      "path" : "Job.Subtype"
    },
    "publicId" : {
      "path" : "Job.PublicID"
    }
  }
},
"RelatedPolicy" : {
  "schemaDefinition" : "RelatedPolicy",
  "root" : "entity.Policy",
  "properties" : {
    "displayName" : {
      "path" : "Policy.DisplayName"
    },
    "publicId" : {
      "path" : "Policy.PublicID"
    }
  }
},
"RelatedPolicyPeriod" : {
  "schemaDefinition" : "RelatedPolicyPeriod",
  "root" : "entity.PolicyPeriod",
  "properties" : {
    "displayName" : {
      "path" : "PolicyPeriod.DisplayName"
    },
    "policyNumber" : {
      "path" : "PolicyPeriod.PolicyNumber"
    },
    "publicId" : {
      "path" : "PolicyPeriod.PublicID"
    }
  }
}
```

Adding functionality to handler method getActivity

Handler class `ExampleActivitiesApiHandler` defines the methods that support the Activities API. This class exists in the following directory in the Studio **Project** window:

configuration→gsrc→mc→activityapi

The `getActivty` handler method uses a private method to retrieve the specified activity from the database using the activity ID. The method then uses the mapper declaration in generating the object that the method returns.

```
function getActivity(activityId : String) : TransformResult {
 var activity = loadActivityById(activityId)
  var mapper = JsonConfigAccess.getMapper("mc.activityapi.activityAPI-1.0", "ActivityDetail")
  return mapper.transformObject(activity)
}

private function loadActivityById(activityId : String) : Activity {
  var activity = Query.make(Activity).compare(Activity#PublicID, Relop.Equals, activityId).select().AtMostOneRow
  if (activity == null) {
    throw new NotFoundException("No activity was found with id " + activityId)
  }
  return activity
}
```

Next steps

After adding the mapper file in Studio and updating the API handler file, do the following:

- Recompile the PolicyCenter application
- Restart the application server

Creating an activity

The API schema (file `activityAPI-1.0.swagger.yaml`) defines a `POST` operation on the `/activities` endpoint. This HTTP operation creates a new activity. Gosu handler class `ExampleActivitiesApiHandler` provides the functionality for the example Activities API. You need to update the stub `createActivity` method in this class to perform the following work:

- Accept a `body` argument that provides the details of the activity to create.
- Load the provided `activityPattern` by code and validates it.
- Load the account using the provided `accounNumber`.
- Check that the account permits additional activities to be added to it.
- Create a transaction for the result in a new bundle.
- Use the already defined mapper for `ActivityDetail` to produce the method response.

Adding functionality to handler method createActivity

Updating class `ExampleActivitiesApiHandler` to provide the functionality for creating a new activity requires that you update the `createActivity` method. This class exists in the following directory in the Studio **Project** window:

configuration→gsrc→mc→activityapi

The following code sample illustrates an updated `createActivity` method , as well as, the use of several private functions to perform associated work.

```
function createActivity(body : JsonObject) : TransformResult {
  var activityPattern = loadActivityPattern(body.get("activityPattern") as String)
  var accountNumber = body.get("accountNumber") as String

  if (accountNumber == null) {
    throw new UnsupportedOperationException("Only linking through accountNumber is currently supported")
  }
  var account = loadAccount(accountNumber)
  if (account.AccountStatus == AccountStatus.TC_WITHDRAWN) {
    throw new BadInputException("The associated account has been withdrawn")
  }

  var activity : Activity
  gw.transaction.Transaction.runWithNewBundle(\b -> {
    activity = activityPattern.createAccountActivity(b, activityPattern, account,
          body.get("subject") as String,
          body.get("description") as String,
          null,
          body.get("property") as Priority,
          body.get("mandatory") as Boolean,
          body.get("targetDate") as Date,
          body.get("escalationDate") as Date)
    })

  var mapper = JsonConfigAccess.getMapper("mc.activityapi.activityAPI-1.0", "ActivityDetail")
  return mapper.transformObject(activity)
}

private function loadActivityPattern(activityPatternCode : String) : ActivityPattern {
  var activityPattern = ActivityPattern. finder.getActivityPatternByCode(activityPatternCode)
  if (activityPattern ==  null) {
    throw new BadInputException( "No activity pattern was found with code " + activityPatternCode)
  }

  return activityPattern
}

//----------------------PRIVATE FUNCTIONS-------------------------
private function loadAccount(accountNumber : String) : Account {
  var account = Account. finder.findAccountByAccountNumber(accountNumber)
  if (account ==  null) {
    throw new BadInputException( "No account exists with account number " + accountNumber)
  }
  return account
}
```

Next steps

After adding the mapper file in Studio and updating the API handler file, do the following:

- Recompile the PolicyCenter application
- Restart the application server

Testing your work

To test your work, you need to create a `POST /activities` request with a request body that contains the necessary activity information. To determine what information to include in the request body, review the definition for `NewActivity` in the JSON mapping file.

The following example JSON indicates the information that you need to include in the request body. Replace the placeholder values in the example with actual values.

```
{
  "accountNumber": "string",
  "activityPattern": "string",
  "description": "string",
  "escalationDate": "2019-11-14T18:06:32.150Z",
  "jobNumber": "string",
  "mandatory": true,
  "priority": "string",
  "subject": "string",
  "targetDate": "2019-11-14T18:06:32.150Z"
}
```

In testing your work, you can use Swagger UI or Postman to create and send the request to the application server, for example.

Updating an activity

So far, the Activities API provides a few basic `GET` and `POST` operations for activities. However, it is common to want to update an application resource as well. This action requires a `PATCH` operation on the specified resource. The newly designed API adds a `PATCH /activities/{activityId}` operation that provides the following functionality:

- It marks the schema properties on the `ActivityDetail` JSON object that are not for update as read-only
- It marks the schema properties on the `ActivityDetail` JSON object that are non-nullable as required
- It adds a `checksum` property to the `ActivityDetail` JSON object that maps to the bean version of the activity
- It returns the result as a `PATCH /activities/{activityId}` operation

Adding read-only properties to the JSON schema

The JSON schema file for the Activities API, `activityAPI-1.0.schema.json`, exists in the following directory in the Guidewire Studio **Project** window:

configuration→config→Integrations→schemas→mc→activityapi

The following example schema updates the JSON `ActivityDetail` object to mark some properties as read-only, as well as, add a `checksum` property.

```
"ActivityDetail" : {
  "type" : "object",
  "properties" : {
    "approvalRationale" : {
      "type" : "string"
    },
    "assignedUser" : {
      "$ref" : "#/definitions/AssignedUser",
      "readOnly" : true
    },
    "checksum" : {
      "type" : "string"
    },
    "description" : {
      "type" : "string"
    },
    "escalationDate" : {
      "type" : "string",
      "format" : "date-time"
    },
    "mandatory" : {
      "type" : "boolean"
    },
    "priority" : {
      "type" : "string",
      "x-gw-type" : "typekey.Priority"
    },
    "publicId" : {
      "type" : "string",
      "readOnly" : true
    },
    "relatedAccount" : {
      "$ref" : "#/definitions/RelatedAccount",
      "readOnly" : true
    },
    "relatedContact" : {
      "$ref" : "#/definitions/RelatedContact",
      "readOnly" : true
    },
    "relatedJob" : {
      "$ref" : "#/definitions/RelatedJob",
      "readOnly" : true
    },
    "relatedPolicy" : {
      "$ref" : "#/definitions/RelatedPolicy",
      "readOnly" : true
    },
    "relatedPolicyPeriod" : {
      "$ref" : "#/definitions/RelatedPolicyPeriod",
      "readOnly" : true
    },
    "status" : {
      "type" : "string",
      "x-gw-type" : "typekey.ActivityStatus",
      "readOnly" : true
    },
    "subject" : {
      "type" : "string"
    },
    "targetDate" : {
      "type" : "string",
      "format" : "date-time"
    }
  }
}
```

Adding a PATCH operation to the API schema

The API Swagger schema file, `activityAPI-1.0.swagger.yaml`, exists in the following directory in Guidewire Studio **Project** window:

configuration→config→Integrations→apis→mc→activityapi

The following code sample illustrates how to add a new `PATCH` operation to the `/activities/{activityId}` endpoint in the Swagger schema.

```
/activities/{activityId}:
  patch:
    summary: "Updates the details of a single activity"
    description: "Updates the details of a single activity"
    operationId: updateActivity
    parameters:
    - $ref: "#/parameters/activityId"
    - name: body
      in: body
      required: true
      schema:
        $ref: "activities#/definitions/ActivityDetail"
    responses:
      '200':
        description: "Returns details of the updated activity"
        schema:
          $ref: "activities#/definitions/ActivityDetail"
```

Adding 'checksum' to the JSON mapper file

The integration mapper file, `activityAPI-1.0.mapping.json`, exists in the following directory in the Studio **Project** window:

configuration→config→Integration→mappings→mc→activityapi

To add the new `checksum` property to the mapper file, add the following code underneath `properties` in the `ActivityDetail` definition.

```
"checksum" : {
  "path" : "Activity.BeanVersion.toString()"
}
```

Creating the necessary JSON wrapper classes

The API handler code uses wrapper classes for the JSON objects that the code manipulates. You must create the necessary wrapper classes before you update the Activity API handler class. See "Generate schema wrapper classes" on page 27 for details.

IMPORTANT You must do this step before you start to update the handler class `updateActivity` method.

Adding method updateActivity to the handler class

To provide functionality for the `PATCH /activity/{activityId}` operation, you need to add a method to support the new functionality to the `ExampleActivitiesApiHandler` handler class. As the `operationId` for the `PATCH` operation is `updateActivity`, you need to add an `updateActivity` method to the handler class.

The following code sample illustrates how to construct an `updateActivity` method on `ExampleActivitiesApiHandler` class.

```
function updateActivity(activityId : String, body : JsonObject) : TransformResult {

  var activity = loadActivityById(activityId)
  var activityDetail = ActivityDetail.wrap(body)

  gw.transaction.Transaction.runWithNewBundle(\b -> {
    activity = b.add(activity)
    activity.ApprovalRationale = activityDetail.approvalRationale ?: activity.ApprovalRationale
    activity.Description = activityDetail.description ?: activity.Description
    activity.EscalationDate = activityDetail.escalationDate ?: activity.EscalationDate
    activity.Mandatory = activityDetail.mandatory ?: activity.Mandatory
    activity.Priority = activityDetail.priority ?: activity.Priority
    activity.Subject = activityDetail.subject ?: activity.Subject
    activity.TargetDate = activityDetail.targetDate ?: activity.TargetDate
  })

  var mapper = JsonConfigAccess.getMapper("mc.activityapi.activityAPI-1.0", "ActivityDetail")
  return mapper.transformObject(activity)
}
```

Next steps

You must do the following to test your work:

- Run `codegen` to generate the necessary Java wrapper classes for the JSON objects used by the class methods.
- Recompile your work.
- Restart the application server.

Setting user permissions

To secure and restrict access to Guidewire objects, PolicyCenter assigns various roles to each individual user. Each role is a collection of one or more permissions. Each permission controls what a user is able to see or do in a certain area of the application. Guidewire calls permissions that apply to specific user interface elements or data model entities *system permissions*.

In general, you set system permissions through the use of the `x-gw-permissions` attribute in the API Swagger schema. For example, to give the ability for a user to create an activity, add something similar to the following to the API schema file:

```
x-gw-permissions:
- actcreate
```

Guidewire recommendations

Guidewire recommends the following:

- If the system permission does not depend on an actual entity object, place the system permission in the API Swagger schema.
- If the permission applies to a specific entity object, place the logic for that permission in the appropriate API handler method.

Setting user permissions for GET /activities

There are many ways to update the `ExampleActivitiesApiHandler.getActivities` handler method to check user permissions for certain actions. This example presents a simple solution to the problem:

- Unrestricted user `su` can view activities for all users
- Individual users can view their own activities only

To provide permission checking on the `GET /activities/{activityId}` operation, you need to update the following methods in the `ExampleActivitiesApiHandler` API handler class:

- `getActivities`
- `loadActivityById`

Adding the assignedUser parameter to the Swagger schema

First, you need to add a parameter for `assignedUser` to the `GET /activities` definition in file `activityAPI.swagger.yaml` so that the parameter is available to the handler class.

```
/activities:
  get:
    ...
    parameters:
      - name: assignedUser
      in: query
      type: string
    ...
```

Adding functionality to handler method getActivities

Class `ExampleActivitiesApiHandler` exists in the following directory in the Studio **Project** window:

configuration→gsrc→mc→activityapi

The following code fragment illustrates how to add logic to that permits unrestricted user `su` to view all activities and the calling user to view only activities associated with that user.

```
function getActivities(assignedUser : String) : List<TransformResult> {
  var query = Query.make(Activity)
  if (assignedUser != null) {
    var credential = Query.make(Credential).compare(Credential#UserName, Relop.Equals,
assignedUser).select().AtMostOneRow
    if (credential == null) {
      throw new BadInputException("No user was found with username " + assignedUser)
    }
    var user = Query.make(User).compare(User#Credential, Relop.Equals, credential).select().AtMostOneRow
    query.compare(Activity#AssignedUser, Relop.Equals, user)
  }

  var resultSet = query.select()
  var mapper = JsonConfigAccess.getMapper("mc.activityapi.activityAPI-1.0", "ActivitySummary")
  return mapper.transformObjects(resultSet)
}
```

Adding functionality to private handler method loadActivityById

In modifying the `ExampleActivitiesApiHandler` class, it makes sense to put the `view` permission check in the `loadActivityById` method. In doing so, it provides a means to make the following two conditions return the same error message:

- Bad activity ID
- No permission to view

Thus, a user without the permission to view an activity cannot tell if the URL is valid or not.

```
private function loadActivityById(activityId : String) : Activity {
  var activity = Query.make(Activity).compare(Activity#PublicID, Relop.Equals, activityId).select().AtMostOneRow
  if (activity == null || !perm.Activity.view(activity)) {
    throw new NotFoundException("No activity was found with id " + activityId)
  }
  return activity
}
```

Setting user permissions for POST /activities

In creating a resource, one needs to check for the necessary permissions in the following ways:

- Does the user have ability to view the resource?
- Does the user have the ability to create the resource.

Thus, in order to create activities on an account, the user needs both of the following permissions:

- The ability to view an activity.
- The ability to create an activity.

It is possible to stipulate directly within the Swagger schema file that the user has the necessary system permission to create an activity resource.

Adding a user permission to the Swagger schema

Add the activity system permission directly in file `activityAPI-1.0.swagger.yaml`, underneath the `POST` entry, as shown.

```
/activities:
  post
    summary: "Creates a new activity"
    description: "Creates a new activity"
    operationId: createActivity
    x-gw-permissions:
    - actcreate
    ...
```

Next steps

After adding the Swagger schema in Studio and updating the API handler file, do the following:

- Recompile the PolicyCenter application
- Restart the application server

Adding search and sort capabilities

The `GET` operation on the `/activities` endpoint on the Activities API is a basic operation that returns the list of activities in Guidewire PolicyCenter. However, suppose that you want the API to (optionally) restrict the returned activity list by assigned user and to (optionally) specify a primary sort column and direction. To provide this functionality, you need to update the following items:

- The `GET` operation for `/activities` in the API Swagger schema
- The `getActivities` method in handler class `ExampleActivitiesApiHandler`

Adding additional parameters to the Swagger schema

Guidewire defines the API schema in file `activityAPI-1.0.swagger.yaml` in the following location in the Studio **Project** window:

configuration→config→Integration→apis→mc→activityapi

The following code example adds `sortBy` and `sortDirection` parameters to the `GET /activities` operation. It uses several `enum` constructions to define the behavior of the parameters.

```
/activities
  get:
    summary: "Returns a list of activities"
    description: "Returns a list of activities"
    operationId: getActivities
    parameters:
    - name: assignedUser
      in: query
      type: string
    - name: sortBy
      in: query
      type: string
      enum: ["assignedUser", "escalated", "priority", "status", "subject", "targetDate"]
      default: "priority"
    - name: sortDirection
      in: query
      type: string
      enum: ["asc", "desc"]
      default: "asc"
    responses:
      '200':
        description: "Returns a list of activities"
        schema:
          type: array
          items:
            $ref: "activities#/definitions/ActivitySummary"
```

Adding functionality to handler class method getActivities

The following code sample updates the `ExampleActivitiesApiHandler.getActivities` method to provide sort and search capabilities. This class exists in the following directory in the Studio **Project** window:

configuration→gsrc→mc→activityapi

```
function getActivities(assignedUser : String, sortBy : String, sortDirection : String) : List<TransformResult> {

  var query = Query.make(Activity)
  var credential = Query.make(Credential)
        .compare(Credential#UserName, Relop.Equals, assignedUser)
        .select().AtMostOneRow

  if (credential == null) {
    throw new BadInputException("No user was found with username " + assignedUser)
  }

  var user = Query.make(User)
        .compare(User#Credential, Relop.Equals, credential)
        .select().AtMostOneRow

  query.compare(Activity#AssignedUser, Relop.Equals, user)

  var resultSet = query.select()

  var sortColumn : IQuerySelectColumn
  switch(sortBy) {
    case "assignedUser":
      sortColumn = QuerySelectColumns.path(Paths.make(Activity#AssignedUser))
      break
    case "escalated":
      sortColumn = QuerySelectColumns.path(Paths.make(Activity#Escalated))
      break
    case "priority":
      sortColumn = QuerySelectColumns.path(Paths.make(Activity#Priority))
      break
    case "status":
      sortColumn = QuerySelectColumns.path(Paths.make(Activity#Status))
      break
    case "subject":
      sortColumn = QuerySelectColumns.path(Paths.make(Activity#Subject))
      break
    case "targetDate":
      sortColumn = QuerySelectColumns.path(Paths.make(Activity#TargetDate))
      break
    default:
      throw new IllegalArgumentException("Unexpected sortBy argument " + sortBy)
  }

  if ("desc" == sortDirection) {
    resultSet.orderByDescending(sortColumn)
  } else {
    resultSet.orderBy(sortColumn)
  }

  var mapper = JsonConfigAccess.getMapper("mc.activityapi.activityAPI-1.0", "ActivitySummary")
  return mapper.transformObjects(resultSet)
}
```

Testing search and sort

It is possible to test the updated search and sort functionality in Swagger UI. If running the application server on a local machine, navigate to the following URL:

`localhost:8180/pc/resources/swagger-ui`

Enter the URL for the Activities API in the **Explore** field and test out the `GET /activities` operation. With the updated `getActivities` method, the Swagger UI now shows the following interactive parameter fields:

Field	Test case
`assignedUser`	Enter a user name to filter the list of returned activities so that it only includes activities assigned to the designated user.
`sortBy`	Chose a value from the drop-down list sort activity list by a specific parameter.
`sortDirection`	Chose a value from the drop-down list to determine the list sort order (either ascending or descending).

Filtering activity details

The Guidewire REST API framework supports the use of GraphQL-style filters. This type of filter serves as a white list of properties to include with object fields that the REST framework materializes and serializes. For example, suppose that you want to filter the response to a `GET /activities/{activityID}` operation in the following ways:

- `full` - Returns all activity detail fields in response to the `GET` operation.
- `basic` - Returns only the fields listed in the filter definition file in response to the `GET` operation.

As the `full` filter returns all fields, which is the default action, it is not necessary to create a filter definition file for that specific filter.

To add a filter to the Activities API, you need to perform the following tasks:

- Add a filter definition file in the appropriate subdirectory in the Studio **Integration** directory that defines the fields to return.
- Update the Swagger schema file to add a filter parameter to the `GET /activities/{activityId}` schema definition.
- Update the `getActivity` handler class method to use the schema `filter` parameter to perform business logic

Creating a filter definition file

Use the version scheme for your filter definition file as you did for the rest of your API configuration files. Describe the filter in the file name, such as `activity_details_basic-1.0.gql`. Place your definition file in the following location in the Guidewire Studio **Project** window:

Integration→filters→mc→activityapi

The `basic` filter definition file lists the following activity detail fields:

```
{
  assignedUser {
    username
  }
  description
  priority
  publicId
  status
  subject
  targetDate
}
```

Note: You must explicitly add this file type in Studio if Studio does not recognize the `*.gql` extension. Use **File→Settings→File Types** to add the file type.

Adding functionality for filtering to the API schema

You need to update the `GET /activities/{activityId}` operation in the Swagger API schema to add a new filter parameter. You find the API schema file, `activityAPI-1.0.swagger.yaml`, in the following location in the Studio **Project** window:

configuration→config→Integration→apis→mc→activityapi

The `filter` parameter uses an `enum` to list the names of the available filters, `basic` and `full`. The `full` filter provides the entire list of fields on the `AccountDetail` object. As this is the default behavior, there is no need to create a specific filter for this situation.

```
/activities/{activityId}
  get:
    summary: "Returns details for a single activity"
    description: "Returns details for a single activity"
    operationId: getActivity
    parameters:
    - $ref: "#/parameters/activityId"
    - name: filter
      in: query
      type: string
      enum: ["basic", "full"]
      default: "full"
    responses:
      '200':
        description: "Returns details for a single activity"
        schema:
          $ref: "activities#/definitions/ActivityDetail"
```

Adding functionality to handler class method getActivity

Finally, you need to update the `ExampleActivitiesApiHandler.getActivity` method to handle the new `filter` parameter. This class exists in the following directory in the Studio **Project** window:

configuration→gsrc→mc→activityapi

The following code sample is an example of how to implement the `filter` parameter in code. Notice that there is no code to handle the `full` filter as it is the default behavior.

```
function getActivity(activityId : String, filter : String) : TransformResult {
  var activity = loadActivityById(activityId)
  var mapper = JsonConfigAccess.getMapper("mc.activityapi.activityAPI-1.0", "ActivityDetail")
  var mappingOptions = new JsonMappingOptions()
  if (filter == "basic") {
    mappingOptions.withFilter("mc.activityapi.activity_details_basic-1.0")
  }
  return mapper.transformObject(activity, mappingOptions)
}
```

Next steps

After adding updating the API handler file, do the following:

- Recompile the PolicyCenter application.
- Restart the application server.

Setting validation constraints

It is possible to add validation constraints to the JSON objects defined in file `activityAPI-1.0.schema.json`. For example, you can define a certain JSON property as being required, or, stipulate the maximum length of a user-supplied field.

File `activityAPI-1.0.schema.json` exists in the following directory in the Studio Project window:

configuration→config→schemas→mc→activityapi

Adding validation constraints to the JSON schema

The following example code updates the `NewActivity` object defined in file `activityAPI-1.0.schema.json` to set the length of the `subject` property to a maximum of 64 characters

```
"NewActivity" : {
  "properties" : {
    "activityPattern" : {
      "type" : "string"
    },
    "accountNumber" : {
      "type" : "string"
    },
    "subject" : {
      "type" : "string",
      "maxLength" : 64
    },
    "description" : {
      "type" : "string"
    },
    "jobNumber" : {
      "type" : "string"
    },
    "priority" : {
      "type" : "string",
      "x-gw-type" : "typekey.Priority"
    },
    "mandatory" : {
      "type" : "boolean"
    },
    "targetDate" : {
      "type" : "string",
      "format" : "date-time"
    },
    "escalationDate" : {
      "type" : "string",
      "format" : "date-time"
    }
  }
}
```

chapter 7

The IRestDispatchPlugin plugin

The `IRestDispatchPlugin` plugin is an optional plugin interface that you can implement to do the following:

- Preprocess incoming REST API requests
- Rewrite outgoing API responses
- Control how and what PolicyCenter logs for each API request

Default plugin implementation

If you do not implement your version of the `IRestDispatchPlugin` plugin, PolicyCenter uses the default `DefaultRestDispatchPlugin` class instead. The default class does the following:

- Performs a minimal rewrite of 401 and 403 errors to remove details of authorization or authentication failure.
- Adds an intentional logging pattern (`PLLoggingMarker.REST_REQUEST`) that provides detailed information about each API request suitable for feeding to a modern log system.

It is also possible to use the `DefaultRestDispatchPlugin` class as the superclass for your plugin implementation as this class implements the `IRestDispatchPlugin` interface.

Example of a log message for a successful request

The following server log message is an example of a successful REST API request.

```
aapplegate e1148f7b-a606-40f5-b43a-f86b3652279b 11:36:54,585 INFO
      REST.Request <RestRequest> REST API Request {path="/activities/v1/activities",
      query="includeTotal=true", pathTemplate="/activities/v1/activities",
      apiFqn="gw.pc.activities.activities-10.0", method="GET", from="0:0:0:0:0:0:0:1",
      user="aapplegate", status=200, error="", elapsedTimeMs=25}
```

Example of a log message for an unsuccessful request

The following server log message is an example of a REST API request that did not succeed.

```
aapplegate 7081b3c7-62ce-4897-9987-f7c7226af992 11:39:39,707 INFO
      REST.Request <RestRequest> REST API Request
      {path="/activities/v1/activities/pc:203/notes", query=null,
      pathTemplate="/activities/v1/activities/{activityId}/notes",
      apiFqn="gw.pc.activities.activities-10.0", method="POST", from="0:0:0:0:0:0:0:1",
      user="aapplegate", status=400, error="gw.api.rest.exceptions.BadInputException",
      elapsedTimeMs=2}
```

Using the IRestDispatchPlugin plugin

The `IRestDispatchPlugin` interface contains methods to preprocess, handle, and log REST API requests. Guidewire provides a default implementation class for the plugin, `DefaultRestDispatchPlugin`. However, if you want to implement your own implementation class for the `IRestDispatchPlugin` plugin, keep the following details in mind.

Preprocessing the request

Use the following methods on the `IRestDispatchPlugin` interface to perform preprocessing work on the incoming requests:

- `handleReceiveRequest`
- `handlePreExecute`

The preprocessing of the request can occur either prior to the start of the work of handling the request or prior to invoking the API handler that performs the actual work of the request.

Use preprocessing to set global variables on the method's `RequestContext` object, or to set any other global context that needs to be thread-local, such as logging contexts. You can also use preprocessing to log request information prior to the start of the work of processing the request.

Rewriting the response to the request

Use the `rewriteResponse` method on the `IRestDispatchPlugin` interface to rewrite outgoing responses generated by the request. For example, you can create a lightweight servlet filter to inject global custom response headers or to implement CORS (cross-origin resource sharing) by rewriting the responses for OPTIONS requests to include the appropriate CORS headers.

Rewriting errors generated by a request

Use the `rewriteErrorInfo` method on the `IRestDispatchPlugin` interface to control exactly what error and exception information the plugin returns while handling the REST request. It is important to understand that the plugin invokes the `rewriteErrorInfo` method only if the response framework or the API handler throws an error or exception during request processing.

Use your implementation of the `rewriteErrorInfo` method to define your own specific best practices around information exposure in error messages. Different environments or different teams have different tolerances for exposing detailed error information to client systems. As it is possible to provide implementation classes for the plugin that behave differently in production and development environments, you can specifically set a more restrictive set of policies to apply to production servers, while allowing development systems to return more detailed information.

Logging the request activity

The `handleRequestProcessed` method on the `IRestDispatchPlugin` interface uses arguments whose values provide information and context about the processing of the REST request, for example:

- The amount of time that the request took
- The response the request generated
- Any errors or exceptions thrown during the course of processing the request

Default implementation class DefaultRestDispatchPlugin

In the base configuration, Guidewire provides a default implementation class for the `IRestDispatchPlugin` interface, Gosu class `DefaultRestDispatchPlugin`. This class implements the following `IRestDispatchPlugin` interface methods:

- `handleReceiveRequest`
- `handlePreExecute`
- `rewriteResponse`
- `rewriteErrorInfo`
- `handleRequestProcessed`

See "Processing REST requests" on page 71 for information on these methods.

Note: It is also possible to use the `DefaultRestDispatchPlugin` class as the superclass for your custom implementation of this plugin.

Processing REST requests

PolicyCenter calls the methods on the plugin `IRestDispatchPlugin` interface during the processing of all REST requests. Use the interface methods to control output and logging. As the methods callbacks occur during the processing of a request, Guidewire recommends that these methods not throw exceptions as it is possible for a an exception to hide an operational or serialization error.

Public interface `IRestDispatchPlugin` contains the following public methods.

`handleReceiveRequest(requestContext)`

PolicyCenter calls the `handleReceiveRequest` method upon receipt of each REST request. At this point in the processing of the request, the REST API framework:

- Has not yet determined the proper operation to take. (The API request can request an unsupported path.)
- Has not authenticated the client that called the request.
- Has not checked permissions for the request.

This method takes the following argument only.

Method argument	Null?	Description
`requestContext`	No	A `RequestContext` object that contains information about this request. See class `RequestContext` for useful properties on this object.

If this method throws an exception, it passes the associated `ErrorInfo` JSON object as the argument to the plugin `handleRequestProcessed` method:

- If the exception thrown by the plugin method implements the `HasErrorInfo` interface, the method passes an `ErrorInfo` object to the `operationError` argument of the `handleRequestProcessed` method.
- If the exception does not implement the `HasErrorInfo` interface, the method passes a generic `ErrorInfo` object with a response code of 500.

Note: It is possible to implement your own version of this method so that it extracts tracking information, if you also implement the necessary traceability IDs. In your version of this method, you can also implement logging of the incoming request, as well. If you do implement your own version of this method, you need to take into account that, at this stage, the REST request can potentially be an attack and not a legitimate request.

`handlePreExecute(requestContext, user)`

PolicyCenter calls the `handlePreExecute` method just prior to setting up the execution environment for the requested operation. If the request requires authentication to perform, this method performs authentication of the user requesting the operation. The `handlePreExecute` method also determines the correct handler to use to process the request.

This method takes the following arguments.

Method argument	Null?	Description
requestContext	No	A `RequestContext` object that contains information about this request. See class `RequestContext` for useful properties on this object.
user	Yes	A `User` object that represents the user requesting the operation. It is possible for this value to be `null` if the request does not require user authentication.

If this method throws an exception, it passes the associated `ErrorInfo` JSON object as the argument to the plugin `handleRequestProcessed` method:

- If the exception thrown by the plugin method implements the `HasErrorInfo` interface, the method passes an `ErrorInfo` object to the `operationError` argument of the `handleRequestProcessed` method.
- If the exception does not implement the `HasErrorInfo` interface, the method passes a generic `ErrorInfo` object with a response code of 500.

`rewriteResponse(requestContext, response)`

If PolicyCenter calls the operation handler on this request and the handler returned successfully, PolicyCenter calls this plugin method just prior to serializing (writing) the response to the request.

This method takes the following arguments.

Method argument	Null?	Description
requestContext	No	A `RequestContext` object that contains information about this request. See class `RequestContext` for useful properties on this object.
response	No	The `Response` object that represents the response to the request. Generally, the method creates the `Response` object from the operation handler return value.

The `rewriteResponse` method returns a `Response` object, which can simply be the `Response` object passed to the method initially. If the request handler was successful in processing the request, the handler returns a `Response` object with a 4xx or 5xx code status.

If serialization fails for some reason:

- The `rewriteResponse` method generates an error of type *`serializationError`*.
- The *`handlerResponse`* and *`serializedResponse`* objects represent different things:
  - The *`handlerResponse`* object contains whatever the request handler returned.
  - The *`serializedResponse`* object contains a 5xxx error code that represents the reason for the server-side failure.

`rewriteErrorInfo(requestContext, errorInfo)`

PolicyCenter calls the `rewriteErrorInfo` method if processing the client request generates an error. This error can occur during any of the following phases of the request:

- During set up of the request
- During execution of the requested operation
- During serialization (writing) of the response returned from a successful completion of executed operation

This method takes the following arguments.

Method argument	Null?	Description
requestContext	No	A `RequestContext` object that contains information about this request. See class `RequestContext` for useful properties on this object.
errorInfo	No	An `ErrorInfo` object that represents the error that occurred.

Method argument	Null?	Description
		Do not attempt to change the value of the passed-in *errorInfo* object in this method, as the `rewriteErrorInfo` method passes the passed-in object to method `handleRequestProcessed`.

The `rewriteErrorInfo` method returns an `ErrorInfo` object that is one of the following:

- The passed-in `ErrorInfo` object
- A custom `ErrorInfo` object that you create

If you want to create your own version of this method that returns a custom `ErrorInfo` object, add the passed-in value of the *errorInfo* argument as a detail of your newly created `ErrorInfo` object.

PolicyCenter serializes the `ErrorInfo` object and sends it as the response to the REST request.

`handleRequestProcessed(requestContext, elapsedMs, handlerResponse, operationError, serializedResponse, serializationError, writeException)`

PolicyCenter explicitly invokes the `handleRequestProcessed` method after it finishes processing the request and after it closes the response output stream. This is to ensure the time spent in this method does not contribute to the latency of the request call.

This method takes the following arguments.

Method argument	Null?	Description
requestContext	No	A `RequestContext` object that contains information about this request. See class `RequestContext` for useful properties on this object.
elapsedMs	No	The total number of milliseconds (as a primitive `long` value) that it took for PolicyCenter to execute any of the other `IRestDispatchPlugin` plugin methods as well as the API call itself.
handlerResponse	Yes	The response from the API handler.
operationError	Yes	An `ErrorInfo` object generated from determining and then executing the request operation handler.
serializedResponse	Yes	Any serialized content returned as the response of the operation handler.
serializationError	Yes	An `ErrorInfo` object generated by the serialization process.
writeException	Yes	An `ErrorInfo` object generated by an exception in the write operation. Typically, this is a IO exception at this stage of the request.

The set of arguments for the `handleRequestProcessed` method depends on whether the request succeeded or whether the request threw an exception during the processing, serialization, or writing of the response. The following list describes the overall control flow of a REST request.

Process flow	Possible actions
Perform initial set up of request: • Determine operation handler • Perform user authentication • Check permissions	If an error at this stage, the method does the following: • It populates the *operationError* argument. • It sets the values of *handlerResponse*, *serializationResponse*, and *serializationError* to `null`.
Invoke the operation handler	If successful, the operation handler returns the *handlerResponse* argument. If an error occurs at this stage, the method does the following: • It populates the *operationError* argument. • It sets the values of *handlerResponse*, *serializationResponse*, and *serializationError* to `null`.

Process flow	Possible actions
Serialize the operation result	The method serializes the result returned in the *handlerResponse* argument and stores the serialization in method argument *serializedResponse*. If an error occurs at this stage, the method does the following: • It populates the *serializationError* argument. • It sets the value of *serializationResponse* to `null`
Write the response to the output stream	As the final step in the REST request processing flow, PolicyCenter writes one of the following to the output stream in serialized form: • *serializedResponse* • *errorInfo* If an error occurs at this stage, the method does the following: • It populates *writeException* argument with the error information.

Thus, by looking at the values of the arguments for this method, you can reconstruct whether PolicyCenter successfully processed the request. For example, if the request failed, you can determine whether the request failed because of a client error (such as bad input or a resource that does not exist) or a server error (such as improper operation of the request handler or the handler returned invalid output). You can also determine whether the error occurred during request handling, during the attempt to serialize the handler response, or while writing the serialized response to the output stream.

Logging request activity

In the base configuration, only the `handleRequestProcessed` method in the `DefaultRestDispatchPlugin` class performs logging of the REST request activity by default. The method writes the `PLLoggingMarker.REST_REQUEST` traceability marker to the application server log using the information provided by the method arguments.

PolicyCenter explicitly invokes the `handleRequestProcessed` method after it finishes writing out the request response. Calling this method after the write operation completes (and the output stream closes) ensures that even if logging adds some amount of server overhead, it does not impact response latency.

Protected logging methods

Class `DefaultRestDispatchPlugin` contains a number of protected methods for manipulating the information to output to the server log files.

The `DefaultRestDispatchPlugin` class provides the following protected methods.

`logClientError(requestContext, elapsedMs, operationError, serializedResponse)`

If there is any issue with REST request itself, the issue generates an error of type `ErrorInfo` (*operationError*). The `handleRequestProcessed` method passes the error to the `logClientError` method, which, in turn, passes the error to the `standardLogResponse` method for output to the server log.

`logServerError(requestContext, elapsedMs, operationError, serializedResponse)`

If there is any issue with the server processing the request, meaning any error in determining the operation handler or an error with the handler executing the request, it generates an error of type `ErrorInfo` (*operationError*). The `handleRequestProcessed` method passes the error to the `logServerError` method, which, in turn, passes the error to the `standardLogResponse` method for output to the server log.

`logSerializationError(requestContext, elapsedMs, handlerResponse, operationError, serializedResponse)`

If the attempt to serialize the request response fails, it generates an error of type `ErrorInfo` (*serializationError*). The `handleRequestProcessed` method passes the error to the `logSerializationError` method, which, in turn, passes the error to the `standardLogResponse` method for output to the server log.

`logWriteException(requestContext, elapsedMs, handlerResponse, operationError, serializedResponse, serializationError, writeException)`

If the attempt to write the request response fails, it is unknown if the client received any part of the response. The failure generates an error of type `Throwable` (*writeException*). The `handleRequestProcessed` method calls the `logWriteException` method (instead of the `standardLogResponse` method) to log information about what failed in the write operation.

`logSuccessfulResponse(requestContext, elapsedMs, handlerResponse, serializedResponse)`

If the result of the REST request is successful, the `handleRequestProcessed` methods passes the request context, the elapsed time, the handler response, and the serialized response to the `logSuccessfulResponse` method. This method, in turn, passes this information to the `standardLogResponse` method for output to the server log.

`standardLogResponse(requestContext, elapsedMs, serializedResponse, error)`

The other logging methods (all except for the `logWriteException` method) call the `standardLogResponse` method to perform the actual work of writing to the server log. This method writes the `PLLoggingMarker.REST_REQUEST` marker to the server log by default.

In the base configuration, this method logs the following information:

- path
- query
- pathTemplate
- apiFqn
- method
- user
- status
- error
- elapsed time

For examples of log entries for successful and unsuccessful REST requests, see "The IRestDispatchPlugin plugin" on page 69

The RequestContext object

The REST API framework uses a `RequestContext` object to represent information about a single REST API request. The framework uses `RequestContext` objects as inputs to the `IRestDispatchPlugin` methods. It is possible to pass a `RequestContext` object to any API handler method by including a parameter of type `RequestContext` on the method.

Usage

The `RequestContext` object serves a number of different functions at runtime. For example, you can use the request `RequestContext` object to access the following kinds of information:

- The original `HttpServletRequest` object for the request in order to obtain raw request information that might not surface in other ways.
- Headers and path parameters by name, in either deserialized (if explicitly listed in the schema) or raw form.

  This type of information is often helpful in writing common infrastructure that multiple API endpoints share.
- Metadata about the request being served such as the `SwaggerOperation` object, path template, or the fully-qualified name of the API.
- Other information about the request being handled, such as the negotiated response content type

Working with RequestContext objects

Public interface `RequestContext` contains the following useful methods for working with `RequestContext` objects.

`getApiFqn()`

The method returns the fully-qualified name of the Swagger schema that published this API, if any. The `getApiFqn` method is not available with the `IRestDispatchPlugin.handleReceiveRequest(RequestContext)` method.

`getBodyAsBytes()`

The method returns the body of the request as a `byte[]` array.

`getBodyAsString()`

The method returns the body of the request as a `String` object.

`getHeaderParameterValue(headerName)`

The method returns the deserialized value of the named header parameter. This method does not work for arbitrary request headers. It only works for header parameters listed explicitly as part of the operation. The `getHeaderParameterValue` method is not available with the `IRestDispatchPlugin.handleReceiveRequest(RequestContext)` method.

`getLogger()`

The method returns the logger (`Logger`) in use for logging within this request context.

`getOperation()`

The method returns the operation information needed to execute this request. The `getOperation` method is not available with the `IRestDispatchPlugin.handleReceiveRequest(RequestContext)` method.

`getPathParameterValue(pathName)`

The method returns the deserialized value of the given path parameter. The `getPathParameterValue` method is not available with the `IRestDispatchPlugin.handleReceiveRequest(RequestContext)` method.

`getPathTemplate()`

The method returns the template-formatted path string that the REST API framework matched for this request, if any. For example, suppose the following:

- The Swagger schema has a base path of `/contacts/v1`.
- The path item has a path of `/contacts/{contactId}`.
- The incoming request is to path `/contacts/v1/contacts/cc:123`.

In this case, the method returns the template-formatted path string as the following:

```
/contacts/v1/contacts/{contactId}
```

The `getPathTemplate` method is not available with the `IRestDispatchPlugin.handleReceiveRequest(RequestContext)` method.

`getProperty(property)`

The method returns a previously stored property value given the property name as input. The input name cannot be `null`.

`getQueryParameterValue(parameterName)`

The method returns the deserialized query parameter value, rather than the raw String value. The method returns `null` for both of the following cases:

- Query parameters that appear with no value.
- Query parameters that do not appear at all.

The `getQueryParameterValue` method is not available with the `IRestDispatchPlugin.handleReceiveRequest(RequestContext)` method.

`getRawHeaderValue(headerName)`

The method takes the name of a request header as input (`headerName`). The return value is one of the following:

- The raw String value for the given header.
- A comma-separated list of the raw String values if the header appears multiple times in the request.
- A `null` value if the header is not included in the request.

To retrieve the deserialized value of a header explicitly declared in the schema as a header parameter, use the `getHeaderParameterValue` method instead of the `getRawHeaderValue` method.

`getRawPathParameters()`

The method returns the set of raw path parameter values extracted from the request path as `Map<String, String>`.

`getRawQueryParameterNames()`

The method returns the query parameter names found in this request as `Set<String>`.

`getRawQueryParameterValues(queryParameter)`

The method takes the name of a query parameter as input (`queryParameter`). The return value is one of the following:

- A `List` object that contains the values with which the parameter appeared.
- An empty list if the parameter appears without any value.
- A `null` value if the query parameter does not appear anywhere in the query `String`.

To retrieve the deserialized value rather than the raw String value, use the `getQueryParameterValue` method instead of the `getRawQueryParameterValues` method.

`getRawRequest()`

The method returns the raw `HttpServletRequest` object associated with this request.

`getRequestContentType()`

The method returns the request content type (as a `MediaType` object), as specified in the `Content-Type` header of the request.

`getRequestStartTime()`

The method returns the timestamp (as a `Date` object) for when the API received the REST request.

`getResponseContentType()`

The method returns the negotiated content type of the API response.

`hasQueryParameter(queryName)`

The method returns one of the following:

- `true` - The specified query parameter exists in the request's query string.
- `false` - The query parameter does not appear on the request's query string.

`pushCloseable(block)`

The method adds an `AutoCloseable` object that the REST API framework executes upon completion of the REST request.

`putProperty(property, value)`

The method stores property values. It uses the following inputs:

- `property` - The property name, which cannot be `null`.
- `value` - The property value.

The method returns the data type of the value stored in this operation.

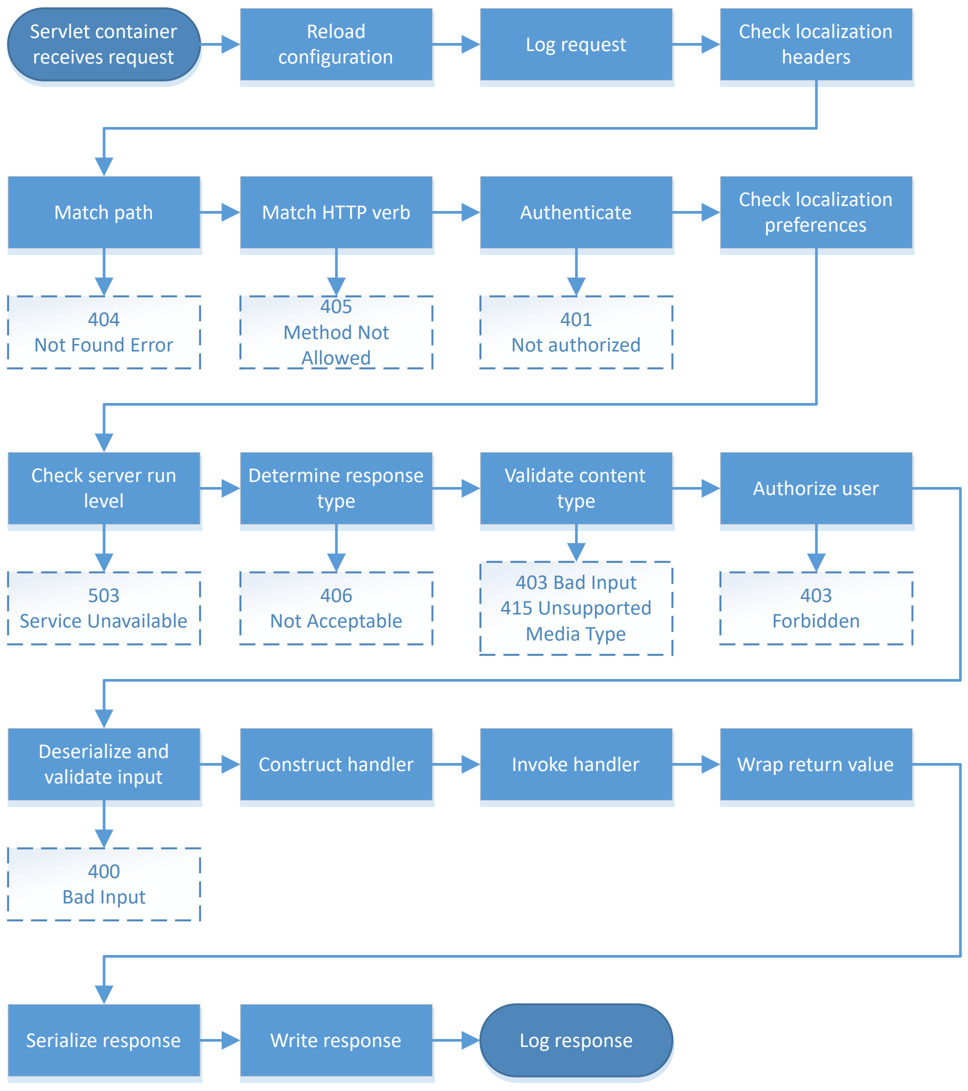
`setLogger(logger)`

The method sets the logger to use for logging with this request context.

chapter 8

REST servlet processing flow

The following diagram illustrates the processing flow for the REST servlet.



The REST processing flow diagram uses the following steps.

Servlet container receives API request

The servlet container handles the first step in the processing chain, before the servlet container invokes the REST servlet.

Reload configuration if necessary

If the application server is running in development mode:

- The REST framework checks every incoming request to determine if any relevant configuration files have changed since the last request.
- The REST framework checks whether a hot swap of Gosu classes has occurred.

For either of these development cases, the REST framework reloads the servlet configuration before processing the request. Guidewire does not support the reload of the servlet configuration in a production environment.

Log request

The default implementation class for the `IRestDispathPlugin` plugin provides a means to log REST API activity.

Check localization headers

Before processing the API request, the REST framework attempts to set up the current language and locale based on one of the following:

- The values set for custom headers `GW-Language` and `GW-Locale` headers
- The value set for the `Accept-Language` header

Setting the localization for the API request ensures that it is possible to properly localize the rest of the API request (including any error messages).

Match path

The REST framework determines whether the request URL matches to a defined Swagger path:

- If the framework finds a match, the framework extracts the path parameters.
- If the framework does not find a match, the framework returns a '404 Not Found' status code.

Match HTTP verb

After the framework determines that the path matches, it next determines whether the HTTP verb used in the API request is one that Guidewire supports for the given path.

- There is a match if the request verb corresponds to a Swagger operation declared on the path, or, if the verb corresponds to one of the `HEAD` or `OPTIONS` HTTP methods that the framework supports automatically.
- There is no match in all other circumstances. If the framework does not find a verb match, the framework returns a '405 Method Not Allowed' status code.

Authenticate

The REST framework then authenticates the request, if the endpoint in question requires authentication. The framework authenticates all Swagger operations by default, with the exception of the `OPTIONS` method, which the framework supports automatically. The framework returns a '401 Unauthorized' response code if authentication fails for any reason.

In the base configuration, the framework marks as unauthenticated the `/apis` endpoint and the `/swagger.json` paths that the default development template adds to the API. The framework determines whether to authenticate `HEAD` method requests based on whether it authenticates the corresponding GET request.

It is also possible to specify the `x-gw-authentication` property on an operation as `false`. This setting instructs the framework to disable authentication for that particular operation.

Check localization preferences

In working with the request localization, the framework only uses any language preferences set by the user in the following circumstances:

- The user must have past authentication.
- The API request itself did not specify any `GW-Language` or `GW-Locale` headers.

If both of these conditions are true, the framework sets the language and locale for the rest of the request handling based on the preferences of the authenticated user.

Check run level

The REST framework checks the run level of the server against the value set for the `x-gw-runlevel` attribute specified on the operation, path, or root Swagger document.

The framework returns a '503 Service Unavailable' response code if the framework determines that the current run level of the server is inadequate to service the request.

Determine response type

The framework determines the response type, if any, based on a process known as content negotiation. Content negotiation attempts to match the Accept header of the request with the content types that the API operation knows how to produce. If the Accept header is present but does not match any of the produced types, the framework returns a '406 Not Acceptable' response code. If the Accept header is not a valid comma-separated list of MIME types, the framework returns a '400 Bad Input' response code.

Validate content type

If the API request contains a request body, then REST framework attempts to match the Content-Type of the request against the types that the API operation can consume.

The framework returns a '400 Bad Input' response code under either of the following circumstances:

- The request contains a body and the API operation does not accept a request payload.
- The specified Content-Type is not a valid MIME type.

The framework returns a '415 Unsupported Media Type' response code if the specified Content-Type does not match one of the types the operation can consume.

Authorize user

If the operation passes authentication, and, the operation declares any system permissions using the `x-gw-permissions` property, the framework checks to see if that the user has all of the permissions required by the property. The framework returns a '403 Forbidden' response code if the authenticated user is missing any of the required permissions.

Deserialize and validate input

Before passing the request data to the handler method on the operation, the REST framework validates and deserializes the input values into POJOs (Plain Old Java Objects).

The framework validates any passed-in parameters according to any validation constraints defined in the Swagger schema, such as:

- Values that are required
- Values must not exceed, or fall below, a certain numveric value
- Values that follow a pattern

The framework deserializes the input values based on following schema properties:

- `type`
- `format`
- `x-gw-type`

The framework validates and deserializes the input data according to an associated schema under the following circumstances:

- If the API request produces a response of type `application/json`
- If the 2xx response code for the operation has an associated schema

The framework returns a '400 Bad Input' exception, with details, if any part of validation or deserialization proces fails.

Construct handler

The REST framework constructs a new instance of the handler class for each API request.

Invoke handler

If the REST framework deems that the API request is valid, the framework invokes the handler method for the operation. In terms of error handling:

- If the handler method throws an exception that implements the `HasErrorInfo` interface, the method uses an `ErrorInfo` object to produce the response and determine the response code.
- If an exception does not implement the `HasErrorInfo` interface, the method returns a generic '500 Internal Server Error' response code.

Wrap return value

After the handler method completes successfully, the REST framework wraps the method return value in a `Response` object:

- If the handler method returns a `Response` object directly, the framework uses that object. The framework then adding a content type automatically based on the negotiated response type, if the content type is not already set.
- If the handler method returns `null` (or `void`) or some other value, the framework creates a Response object automatically, with a response code based on the first 2xx response defined on the operation in the Swagger schema.

If the schema does not define a 2xx responses, it is an error for the handler to return anything other than a `Response` object, as otherwise, the framework does not know what status code to assign to the response.

Serialize response

Before the REST framework can return the data, the framework needs to serialize the data into to a `String` or a `byte[]` object:

- If the response code has an associated schema, and the handler methods returns either a `JsonObject` or `JsonWrapper` object, the framework serializes the data according to the specified schema.
- If the framework serializes the data based on the schema, and the handler method returns data that does not conform to the schema, the framework returns a '500 Internal Server Error' response code. Thus, the framework can return a 500 error if the handler method declares properties not declared in the schema, or, if the method sets a value for a property that is not the correct type specified by the schema.

Writer response

After the REST framework completes the serialization of the output from the request handler method, the framework writes the output to the request output stream.

Log response

As a final step, the REST framework logs the request and its response after the framework writes out the response, which prevents the client request from having to wait on the log action. In the base configuration, the framework logs each API request automatically with summary information about the path and operation requested, the response code, and the time taken to service the request.

chapter 9

Guidewire Swagger specification

The Guidewire InsuranceSuite REST API framework supports a subset of the Swagger 2.0 specification. All properties prefixed with `x-gw-` are Guidewire extensions to the core Swagger 2.0 specification. The Swagger 2.0 specification specifies all other properties that are not Guidewire extensions.

> **Note:** Review the individual topics for the Swagger elements for a listing of the properties that Guidewire does not support from the Swagger 2.0. specification.

Swagger file combination

An API contract can span multiple Swagger files. PolicyCenter combines the separate Swagger files as it generates the final, complete contract for a REST API. The simplest method of combining files is a straight, linear concatenation of each file's contents, appending one file after the other.

It is possible for PolicyCenter to interpret the concatenated contents of the contract file on an individual property basis. PolicyCenter uses this behavior to intelligently merge property definitions that span multiple files using a variety of combination techniques or styles. The following list describes the combination styles that PolicyCenter supports.

Merge style	Description
Merge	PolicyCenter merges child objects based on the rules for combining the object.
Merge by key	PolicyCenter matches objects based on keys in a map. For example, PolicyCenter merges elements under "paths" with the same key.
Merge by <x>	PolicyCenter uses a key (a tag name or a parameter such as `$ref/name/id`) to match equivalent items to merge. PolicyCenter primarily uses this combination style for lists of objects.
Merge of extensions	PolicyCenter merges extensions objects by taking the first *defined* value for each key, even if the defined value is null.
First non-null	As PolicyCenter merges *N* multiple objects, PolicyCenter assigns the first non-null value to the property that it finds for that object.
Not inherited	PolicyCenter uses this style, in general, for default values defined in a root object that it must not propagate with the file. Thus, the property explicitly does not inherit across files.
N/A	PolicyCenter treats the entire object that contains the property as an atomic unit and does not merge the object. PolicyCenter primarily uses this style for a child object contained within a parent object. PolicyCenter never merges the properties of the child object individually. However, it is possible to replace the entire child object as a whole.

Specifying properties while combining object definitions

If an object definition spans multiple files, properties defined in one file do not need to be respecified in the other files. For example, if a file adds a property to an object defined in another file, you only need to specify the added property in the second file.

A best practice is to specify the required properties for an object in the file that initially defines the object.

'Required if published'

PolicyCenter marks a few schema properties as **Required if published**, which has the following meaning:

- If file `published-apis.yaml` lists the schema as published, PolicyCenter requires that property.
- If file `published-apis.yaml` does not list the schema, PolicyCenter does not require that property.

File `published-apis.yaml` can specify a property as required in the following ways:

- By specifying the property directly
- By inheritance from a combined schema

You see the phrase **Required if published** in **Required properties** reference tables in the documentation on the Swagger specification.

'Documentation only'

The Swagger specification defines many properties that have no effect on the resulting REST API contract or the framework's runtime behavior. It is still possible to specify the properties for purposes that lie outside the domain of the framework, such as consumption by third-party tools. The REST API framework ignores the values of these properties and passes their values through unmodified while parsing the Swagger file. Guidewire indicates this type of property as **Documentation only**.

You see the phrase **Documentation only** in **Optional properties** reference tables in the documentation on the Swagger specification.

Swagger document objects

The Swagger root object

The Swagger specification defines a root document object for the API specification. This topic describes the manner in which the object's properties affect the REST API framework. The topic also describes Guidewire-specific framework properties added to extend the root object. This topic does not duplicate root object information already described in the Swagger specification.

Default propagation of property values

Some properties on the root object serve as default values for operations on the root object. It is possible to specify properties for an operation explicitly, or, it is possible for a property to inherit its value from the root document. However, PolicyCenter only propagates a property default within a given Swagger file.

Thus, if a particular operation does not explicitly define a certain property, PolicyCenter uses the corresponding value specified in the root object as the property default value. For example, suppose that the contact-1.0 schema defines the root-level `produces` property as `["application/json"]`. PolicyCenter treats the value of the root-level `produces` property as the default value for any operation defined in that file that does not explicitly specify its own `produces` property. Any operation within the file that does not explicitly define the `produces` property uses the root object `["application/json"]` value as its default value.

However, if file `contact_ext-1.0` combines with the `contact-1.0` schema file, file `contact_ext-1.0` does not inherit the `produces` property on the root document. This behavior ensures that root-level default values are always local to the file in which they exist and that the default values never bleed through to combining files in unexpected ways.

PolicyCenter only applies default values locally within a single file. Default properties do not propagate to operations defined in other files.

Required properties

Property	Type	Description	Combination style
`swagger`	`string`	**Required.** Documentation only. Each Swagger file must specify the `swagger` property. Guidewire currently requires this value to be `2.0`.	Not inherited
`info`	*Info object*	**Required if published.** Documentation only. Property provides metadata about the API.	Merged
`basePath`	`string`	**Required if published.** Property specifies the base path for the API when it is published. The value must start with a leading slash (/).	First non-null
`paths`	`map<string, `*`Patch item object`*`>`	**Required if published.** Property specifies the available paths and operations for the API. The property must start with a '/' character. The property can also include multiple parts, each separated by a '/' character. Use the '{ }' syntax to specify a parameter name. PolicyCenter constructs the full URL by concatenating the following items: • Application servlet context + • REST servlet `url-mapping` string (default is '/rest') + • `basePath` property + • Key of the `paths` property For example, given the following components. • A servlet context of 'http://localhost:8180/pc' • A default `url-mapping` string ('/rest') • A `basepath` of '/contact/v1' • A `paths` key of '/contact/{contactId}' • A `contactId` value of 'johnsmith' The concatenation of these components constructs the following URL: http://localhost:8180/pc/rest/contact/v1/contact/johnsmith	Merge by key

Optional properties

Property	Type	Description	Combination style
`consumes`	`string[]`	Property specifies the acceptable input content types at runtime. It also becomes the default value for any operation defined in the same file that does not explicitly define the `consumes` property. Each member of the array must be a valid MIME type.	Not inherited
`externalDocs`	*External documentation object*	Documentation only.	First non-null
`definitions`	`map<string, `*`JSON schema object`*`>`	Property defines JSON schema definitions inline. You reference schema definitions defined inline without an alias prefix, for example: `#/definitions/`*`name`*	Merged using the rules defined in .
`host`	`string`	Property hard codes the host property as PolicyCenter retrieves the Swagger file at runtime. If not provided, PolicyCenter uses the host serving the Swagger schema as the value. The `host` property value: • Must be a valid host name, with no schema or sub-path. • Can optionally include a port number.	First non-null

Property	Type	Description	Combination style
`parameters`	`map<string,` *`Parameter object`*`>`	Property defines shared parameter definitions. To reference a parameter definition, use the following syntax: `#/parameters/`*`name`* The *`name`* variable identifies the actual parameter definition to reference.	Merge by key
`produces`	`string[]`	Property specifies the output types that the caller can request in the `Accept` header. The negotiated content type can affect how PolicyCenter serializes certain object types. For example, this property can affect whether PolicyCenter serializes a `JsonObject` or `TransformResult` object type as JSON or XML. The property acts as the default value for any operation defined in this same file that does not explicitly define the `produces` property. Each member of the array must be a valid MIME type.	Not inherited
`responses`	`map<string,` *`Response object`*`>`	Property defines shared response definitions. To reference a response definition, use the following syntax `#/responses/`*`name`* The *`name`* variable identifies the actual response definition to reference.	Merge by key
`schemes`	`string[]`	Property hard-codes the `schemes[]` property as PolicyCenter retrieves the Swagger schema at runtime. If not provided, PolicyCenter uses the schema requesting the Swagger schema as the value. Each array entry must one of the following types: • `http` • `https` • `ws` • `wss`	First non-null
`security`	*Security requirements object*[]	Documentation only.	First non-null
`securityDefinitions`	`map<string,` *`Security scheme object`*`>`	Documentation only. Property defines the keys that specify the names of the security schemes.	Merge by key
`tags`	*Tag object*[]	Documentation only.	Merged by tag name

Guidewire extension properties

Property	Type	Description	Combination style
`x-gw-apihandlers`	`string[]`	Property defines the default list of API handler classes. PolicyCenter searches the list, in order, to determine the handler class to use as it binds an operation to a handler. Each array entry must be a valid Gosu or Java class name. PolicyCenter uses the root object list only if the individual operation does not specify an overriding `x-gw-apihandlers` property.	First non-null
`x-gw-combine`	`string[]`	Property defines additional APIs to combine with the contents of this file.	Not inherited

Property	Type	Description	Combination style
`x-gw-cors-policy`	`string`	Property specifies the default CORS policy for all paths in the API, unless overridden at the path level. If you do not specify a CORS policy for the API or path, PolicyCenter disables CORS for that endpoint.	First non-null
`x-gw-cors-policies`	`map<string,` *`CORS policy object`*`>`	Property defines the CORS policies that you can reference in the document root or in the path.	Merge by key
`x-gw-parameters-sets`	`map<string,` *`Parameter object`*`[]>`	Property defines a named list of parameters that PolicyCenter can then use in an operation.	Merge by key - PolicyCenter then combines the array of parameters in the set as specified by the rules for parameters on a path or operation
`x-gw-permissions`	`string[]`	Property defines the default list of system permissions a user must have to request a particular operation. Each member of the array must reference a valid `SystemPermissionType` typecode. If the authenticated user does not have the required permissions to request an operation, the operation returns a 403 Forbidden error. PolicyCenter uses the root object list only if the individual operation does not specify an overriding `x-gw-permissions` property.	Not inherited
`x-gw-runlevel`	`string`	Property specifies the required run level of the server in order to process an operation. If the server is not at the specified run level for the requested operation, the operation returns a 503 Service Unavailable error. PolicyCenter uses this property value as the default value for any operation that does not specify the `x-gw-runlevel` explicitly. PolicyCenter supports the following server run levels: • `NONE` • `GUIDEWIRE_STARTUP` • `NODAEMONS` • `DAEMONS` • `MULTIUSER` If the root document or operation does not specify the `x-gw-runlevel` property, PolicyCenter uses the `NODAEMONS` (maintenance) run level as the default value.	Not inherited
`x-gw-schema-combine`	string[]	Property specifies additional JSON schemas to combine inline into this Swagger schema. Adding schemas in this fashion: • Makes the JSON schema definitions available without a prefix. • Combines the JSON definitions with any schema definitions in the file. Each member of the array must be a fully qualified name of a JSON schema.	Not inherited
`x-gw-schema-import`	`map<`*`string1`*`,` *`string2`*`>`	Property maps an alias (*`string1`*) to a JSON Schema name (*`string2`*). You can then use the alias to reference definitions in the Schema. The property key (*`string1`*) can be any string. The property value (*`string2`*) must be a fully-qualified JSON Schema.	Merge by key
`x-gw-serialization`	Object for serialization	Property defines default serialization options for any operation that does not specify the `x-gw-serialization` property explicitly.	Not inherited

See also

- Refer to the following document for details of the Swagger 2.0 specification:
  The Swagger 2.0 Specification definition of the root document object

The Swagger Contact object

This object provides contact information for the exposed API.

Property	Type	Description	Combination style
`email`	`string`	Documentation only. The string value must be in the form of a valid email address.	N/A
`name`	`string`	Documentation only. Property provides the identifying name of the contact person or organization.	N/A
`url`	`string`	Documentation only. The string value must be in the form of a valid URL.	N/A

The Swagger External Documentation object

The External Documentation object provides a means to reference an external resource for extended documentation.

Property	Type	Description	Combination style
`description`	`string`	Documentation only. Property provides a short description of the target documentation.	N/A
`url`	`string`	Documentation only. Property provides the URL for the target documentation. The string value must be a valid URL.	N/A

The Swagger Header object

The Header object defines header characteristics.

Required properties

Property	Type	Description	Combination style
`items`	*Items object*	**Required if type is array.** Do not use otherwise.	First non-null
`type`	`string`	**Required.** Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values. Valid values are: • `array` • `boolean` • `integer` • `number` • `string`	First non-null

Optional properties

Property	Type	Description	Combination style
`collectionFormat`	`string`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null

Property	Type	Description	Combination style
		The value must be one of the following: • `csv` • `ssv` • `tsv` • `pipes`	
`default`	*`Any object type`*	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values. Value needs to be a valid according to the combination of `type`, `format`, and `x-gw-type`.	First non-null
`description`	`string`	Documentation only.	First non-null
`enum`	*`anyType`*`[]`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values. Value needs to be a valid according to the combination of `type`, `format`, and `x-gw-type`.	First non-null
`exclusiveMaximum`	`boolean`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`exclusiveMinimum`	`boolean`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`format`	`string`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`maximum`	`number`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`maxItems`	`integer`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`maxlength`	`integer`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`minimum`	`number`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`minItems`	`integer`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`minLength`	`integer`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`multipleOf`	`number`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`pattern`	`string`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null
`uniqueItems`	`boolean`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null

Guidewire extension properties

Property	Type	Description	Combination style
`x-gw-export-enumeration`	`boolean`	If set to `true`, PolicyCenter writes out the typekey values as an `enum` property while creating the Swagger schema for external clients. A value of `true` is only valid if the `enum` property is not set and `x-gw-type` is a typekey type. The default is `false`.	First non-null

Property	Type	Description	Combination style
`x-gw-type`	`string`	Documentation only. PolicyCenter returns header values on responses as they are and does not validate the header values.	First non-null

The Swagger Info object

This object provides metadata abut the REST API.

Property	Type	Description	Combination style
`title`	`string`	**Required if published.** Documentation only. The title of the application.	First non-null
`version`	`string`	**Required if published.** Documentation only. The version of the application API, which is not the same as the version of the API specification.	First non-null
`contact`	*Contact object*	Documentation only.	First non-null
`description`	`string`	Documentation only.	First non-null
`license`	*License object*	Documentation only.	First non-null
`termsOfService`	`string`	Documentation only.	First non-null

The Swagger Items object

The Items object provides a limited subset of the properties on the JSON schema Items object. Parameter definitions that do not have an `in` property of `body` use this object. Guidewire does not support the following properties from the Swagger 2.0 specification on the `Items` object:

- `collectionFormat`
- `default`
- `items`
- `minItems`
- `maxItems`
- `uniqueItems`

Guidewire specifically does not support nested arrays for parameter types.

Required properties

Property	Type	Description	Combination style
`type`	`string`	**Required.** Defines the base JSON type for the items. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter deserializes the data into a Java object at runtime. The value must be one of the following: • `boolean` • `integer` • `number` • `string`	First non-null

Optional properties

Property	Type	Description	Combination style
`enum`	*any*[]	Specifies a list of values that the input must match. PolicyCenter turns the `enum` values in the schema into Java objects at runtime. It then compares these values against the input values using `equals` and `hashCode` methods. Each element of the array must be a JSON value that PolicyCenter can parse based on the `type`, `format`, and `x-gw-type` values of the item.	First non-null
`exclusiveMaximum`	`boolean`	Determines if PolicyCenter treats the maximum value as inclusive or exclusive for purposes of comparison between two values. Only set a value for this property if you also specify a value for the `maximum` property as well.	First non-null
`exclusiveMinimum`	`boolean`	Determines if PolicyCenter treats the minimum value of this property as inclusive or exclusive for purposes of comparison between two values. The default value is `false`, which means that PolicyCenter treats the value as inclusive. Only set a value for this property if you also specify a value for the `minimum` property as well.	First non-null
`format`	`string`	Defines the base JSON type format for the parameter. The combination of the `type`, `format`, and `x-gw-type` properties determines how PolicyCenter deserializes the data into a Java object at runtime.	First non-null
`maximum`	`number`	Specifies the maximum numeric value allowed for an item. PolicyCenter parses this value as a fixed-point value, with no loss of precision. PolicyCenter then converts the value to an internal Java representation for the purpose of comparison at runtime. The comparison is either inclusive or exclusive of the `maximum` value, depending upon the value of the `exclusiveMaximum` property. **Example 1.** If the parameter has a `type` value of `integer` and a `format` value of `int32`, the `maximum` value must be a valid integer. At runtime, PolicyCenter converts the value to an `integer` value. **Example 2.** If the parameter has a `type` value of `string` and a `format` value of `gw-bigdecimal`, PolicyCenter converts the value to `BigDecimal` at runtime. Only set a value for this property if the parameter's runtime type is a numeric type.	First non-null
`maxLength`	`integer`	Determines the maximum length of the property value, inclusive. Only set a value for the `maxLength` property if the runtime type is `string`. If specified, the value cannot be a negative number.	First non-null
`minimum`	`number`	Specifies the minimum numeric value allowed for an item. PolicyCenter parses this value as a fixed-point value, with no loss of precision. PolicyCenter then converts the value to an internal Java representation for the purpose of comparison at runtime. The comparison is either inclusive or exclusive of the `minimum` value, depending upon the value of the `exclusiveMinimum` property. This property operates in an analogous manner to the `maximum` property. Only set a value for this property if the parameter's runtime type is a numeric type.	First non-null
`minLength`	`integer`	Determines the minimum length of the property value, inclusive. Only set a value for the `minLength` property if the runtime type is `string`. If specified, the value cannot be a negative number.	First non-null
`multipleOf`	`number`	Specifies that the parameter value must be a multiple of the value of this parameter. PolicyCenter parses this value as a fixed-point value, with no loss of precision. PolicyCenter then converts the value to an internal Java representation for the purpose of comparison at runtime.	First non-null

Property	Type	Description	Combination style
		Set a value for this property only if the runtime type of the item is a numeric type.	
`pattern`	`string`	Specifies a regular expression that the input must match. PolicyCenter does not explicitly anchor the regular expression by default. If you want the regular expression to match the entire input string, then you need to explicitly anchor the expression with ^ and $. PolicyCenter evaluates the `pattern` string using the Java regular expression engine. Thus, the regular expression must match the Java syntax. The Java syntax can have some minor differences from the JavaScript syntax, and, therefore, represent a slight deviation from the Swagger specification. Only set a value for the `pattern` property if the parameter has a runtime type of `string`.	First non-null

Guidewire extension properties

Property	Type	Description	Combination style
`x-gw-type`	`string`	Defines the base JSON type for the parameter. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter deserializes the data into a Java object at runtime. Do not specify this value for parameters of type `body`.	First non-null
`x-gw-export-enumeration`	`boolean`	If set to `true`, PolicyCenter writes out the typekey values as an `enum` property while creating the Swagger schema for external clients. A value of `true` is only valid if the `enum` property is not set and `x-gw-type` is a typekey type. The default is `false`.	First non-null
`x-gw-extensions`	`map<string, *any*>`	The values in `x-gw-extensions` are available to the `IRestValidatorFactoryPlugin` plugin if you invoke the plugin to create custom validators for this parameter. This value can be an arbitrary map of property keys to values. The key values can be any string, and the *`object`* values can be any object, including nested JSON objects.	Merge of extensions

The Swagger License object

The License object provides license information for the exposed REST API.

Property	Type	Description	Combination style
`name`	`string`	**Required.** Documentation only. The license name used for the API.	N/A
`url`	`string`	Documentation only. URL to the license used for the API. The string value must be in the form of a valid URL.	N/A

The Swagger Operation object

This object describes a single REST API operation on a path.

Required properties

Property	Type	Description	Combination style
`operationId`	`string`	**Required.** If not using the `AbstractApiHandler` class, PolicyCenter uses this value as the expected name of a method on the API handler class. The value must be the following: • A valid Java or Gosu method name • Unique within this Swagger schema	First non-null
`responses`	`map<string,` *`Response object`*`>`	**Required.** Defines the set of responses for the operation. All operations must have at least one response defined: • If you define a single response with a 2xx response code, PolicyCenter uses that response code as the default on successful requests. • If you do not specify a 2xx response code, or if you specify multiple 2xx response codes, the API handler must return a `Response` object that specifies the appropriate response code. PolicyCenter assumes that 4xx and 5xx response codes are for documentation purposes only, and are subject to less stringent validation rules. The keys for this map must be valid HTTP response code strings (for example, 200, or similar) or the string "default".	Merge by key

Optional properties

Property	Type	Description	Combination style
`consumes`	`string[]`	Overrides any default `consumes` property declared on the Swagger root object. An operation that specifies a body parameter must define also at least one consumed type, either directly or by inheriting a default value from the root level of the document. This property determines what input content types are acceptable at runtime. Each member of the array must be a valid MIME type.	First non-null
`deprecated`	`boolean`	Documentation only.	First non-null
`description`	`string`	Documentation only.	First non-null
`externaDocs`	*External documentation object*	Documentation only.	First non-null
`parameters`	*Parameter object*[]	Defines the set of input parameters that this operation accepts. The operation can override parameters defined at the Path Item level by defining a parameter with the same name and location.	Merge by logical ID, which is `$ref`, if specified, otherwise, it is `name + in`.
`produces`	`string[]`	Overrides any default `produces` property declared on the root object. An operation that specifies a 200 or 201 response must specify also at least one produced type, either directly or by inheriting a default value from the root level of the document. This property determines what output types the caller can request using the Accept header. The negotiated content type can affect how PolicyCenter serializes some types. For example, it can affect whether PolicyCenter serializes a `JsonObject` object or `TransformResult` object as JSON or as XML. Each member of the array must be a valid MIME type.	First non-null

Property	Type	Description	Combination style
`schemes`	`string[]`	Documentation only. Members of the array must be one of the following: • `http` • `https` • `ws` • `wss`	First non-null
`security`	*Security requirement object*	Documentation only.	First non-null
`summary`	`string`	Documentation only.	First non-null
`tags`	`string[]`	Documentation only.	First non-null

Guidewire extension properties

Property	Type	Description	Combination style
`x-gw-apihandler`	`string`	Overrides the list of `x-gw-apihandler` classes defined on the root Swagger object.	First non-null
`x-gw-authenticated`	`boolean`	Determines whether this operation requires authentication. If you do not specify a value, the default is `true`.	First non-null
`x-gw-extensions`	`map<string,` *anyType*`>`	The values in `x-gw-extensions` are available to the `IRestValidatorFactoryPlugin` plugin as well as available at runtime using `SwaggerOperation` on the `RequestContext` object. This value can be an arbitrary map of property keys to values. The key values can be any string, and the *anyType* values can be any object, including nested JSON objects.	Merge of extensions
`x-gw-parameter-sets`	`string[]`	Defines a list of parameter sets. PolicyCenter includes the parameters from the named sets as if the parameters were defined inline in this operation. The parameters defined explicitly through the `parameters` property on the operation override any parameters included from a parameter set if the parameters have the same logical key (either `$ref` or `name + id`). Each member of the array must be the name of a parameter set defined in the `x-gw-parameter-sets` property on the document root.	Merge by name
`x-gw-permissions`	`string[]`	Overrides any value set for the `x-gw-permissions` property on the Swagger root object. The authenticated user must have all of the specified permissions in order to make a request against this operation. Otherwise, PolicyCenter returns a 403 Forbidden error. Each member of the array must be a valid `SystemPermissionType` typecode.	First non-null
`x-gw-reserve-db-connection`	`boolean`	If set to `true`, PolicyCenter reserves a database connection for the duration of the handling of this request. If you do not specify a value, the default is `false`.	First non-null
`x-gw-runlevel`	`string`	Overrides any value for `x-gw-runlevel` set on the Swagger root object. This value specifies the run level that the server must be at in order to process requests for a given operation. If the server is not at that run level, PolicyCenter returns a 503 Service Unavailable error.	First non-null

Property	Type	Description	Combination style
		PolicyCenter supports the following server run levels: • `NONE` • `GUIDEWIRE_STARTUP` • `NODAEMONS` • `DAEMONS` • `MULTIUSER` If you do not specify a value for `x-gw-runlevel` in the root Swagger document or in the operation, the default rune level value is `maintenance`.	
`x-gw-serialization`	*X-GW-Serialization object*	Overrides any value set for `x-gw-serialization` on the root Swagger object.	First non-null

The Swagger Parameter object

The Parameter object describes a single operation parameter. The combination of the following properties uniquely define a parameter:

- `in`
- `name`

Required properties

Property	Type	Description	Combination style
`in`	`string`	**Required.** Determines the location of the parameter specification. The value must be one of the following: • `body` - The body of the request • `formData` - As form data • `header` - As a custom header • `path` - As part of the path • `query` - On the query string	First non-null
`items`	*Items object*	**Required if type equals `array`.** Defines the type of items, if the parameter is of type `array`.	Merge
`name`	`string`	**Required.** Specifies the name of the parameter: • `body` parameters - The value must be `body`. • `formdata` parameters - • `header` parameters - The name of the custom header. HTTP header names are case-insensitive. • `path` parameters - The value must match a template variable within the path whose operation includes this parameter. • `query` parameters - The name of the query parameter on the URL. In all cases, PolicyCenter uses this value to look for a matching parameter in the API handler method to which it can pass the runtime value.	First non-null
`schema`	*Schema object*	**Required if `in equals body`.** Defines the schema to which the request body must conform, if any.	First non-null
`type`	`string`	**Required unless `in equals body`.** Defines the base JSON type for the parameter. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter deserializes the data into a Java object at runtime. You must specify a value for this property if the parameter is in the query, path or header. Do not specify a value for this property for body parameters.	First non-null

Property	Type	Description	Combination style
		The property value must be one of the following: • `array` • `boolean` • `file` • `integer` • `number` • `string`	

Optional properties

Property	Type	Description	Combination style
`$ref`	`string`	References a parameter defined on the Swagger root document. It is not possible to combine this property with any other properties. Use the following syntax for the string value: `#/parameters/`*`name`*	First non-null
`allowEmptyValue`	`boolean`	Determines if it is permissible to give this parameter an empty value. Use this property with `query` parameters only. The default value is `false`.	First non-null
`collectionFormat`	`string`	Determines what separator PolicyCenter uses for splitting parameter values if the parameter is of type `array`. The value must be one of the following types: • `csv` - Comma separator (default) • `ssv` - Space separator • `tsv` - Tab separator • `pipes` - The '|' character • `multi` - Multiple specifications of the parameter, which Guidewire permits for query parameters only	First non-null
`default`	*Any*	Specifies a default value for the parameter if the parameter is not specified on input. Do not use with `body` parameters, or, if the value of the `required` property is `true`. At runtime, if you do not specify a value for the parameter, the `default` property value is available in the `RequestContext` object and passed to the API Handler method as if the parameter was specified on the request.	First non-null
`description`	`string`	Documentation only.	First non-null
`enum`	*Any*[]	Specifies a list of values that the input must match. PolicyCenter turns the `enum` values in the schema into Java objects at runtime. It then compares these values against the input values using `equals` and `hashCode` methods. Each member of the array must be a JSON value that PolicyCenter can parse based on the `type`, `format`, and `x-gw-type` values of the parameter.	First non-null
`exclusiveMaximum`	`boolean`	Determines if PolicyCenter treats the maximum value of this property as inclusive or exclusive for purposes of comparison between two values. The default value is `false`, which means that PolicyCenter treats the value as inclusive. Only set a value for this property if you also specify a value for the `maximum` property as well.	First non-null
`exclusiveMinimum`	`boolean`	Determines if PolicyCenter treats the minimum value of this property as inclusive or exclusive for purposes of comparison between two values. The default value is `false`, which means that PolicyCenter treats the value as inclusive. Only set a value for this property if you also specify a value for the `minimum` property as well.	First non-null

Property	Type	Description	Combination style
`format`	`string`	Defines the base JSON type format for the parameter. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter deserializes the data into a Java object at runtime. Do not specify a value for this property if this is a `body` parameter.	First non-null
`maximum`	`number`	Specifies the maximum numeric value allowed for this parameter. PolicyCenter parses this value as a fixed-point value, with no loss of precision. PolicyCenter then converts the value to an internal Java representation for the purpose of comparison at runtime. The comparison is either inclusive or exclusive of the `maximum` value, depending upon the value of the `exclusiveMaximum` property. **Example 1.** If the parameter has a `type` value of `integer` and a `format` value of `int32`, the `maximum` value must be a valid integer. At runtime, PolicyCenter converts the value to an `integer` value. **Example 2.** If the parameter has a `type` value of `string` and a `format` value of `gw-bigdecimal`, PolicyCenter converts the value to `BigDecimal` at runtime. Only set a value for this property if the parameter's runtime type is a numeric type.	First non-null
`maxItems`	`integer`	Specifies the maximum number of array members, inclusive. Only set a value for this property if the parameter is of type `array`. If specified, the value cannot be a negative number.	First non-null
`maxLength`	`integer`	Determines the maximum length of the parameter value, inclusive. Only set a value for the `maxLength` property if the runtime type is `string`. If specified, the value cannot be a negative number.	First non-null
`miniItems`	`integer`	Specifies the minimum number of array members, inclusive. Only set a value for this property if the parameter is of type `array`. If specified, the value cannot be a negative number.	First non-null
`minimum`	`number`	Specifies the minimum numeric value allowed for this parameter. PolicyCenter parses this value as a fixed-point value, with no loss of precision. PolicyCenter then converts the value to an internal Java representation for the purpose of comparison at runtime. The comparison is either inclusive or exclusive of the `minimum` value, depending upon the value of the `exclusiveMinimum` property. This property operates in an analogous manner to the `maximum` property. Only set a value for this property if the parameter's runtime type is a numeric type.	First non-null
`minLength`	`integer`	Determines the minimum length of the parameter value, inclusive. Only set a value for the `minLength` property if the runtime type is `string`. If specified, the value cannot be a negative number.	First non-null
`multipleOf`	`number`	Specifies that the parameter value must be a multiple of the value of this property. PolicyCenter parses this value as a fixed-point value, with no loss of precision. PolicyCenter then converts the value to an internal Java representation for the purpose of comparison at runtime. Set a value for this property only if the runtime type of the parameter is a numeric type.	First non-null
`pattern`	`string`	Specifies a regular expression that the input must match. PolicyCenter does not explicitly anchor the regular expression by default. If you want the regular expression to match the entire input string, then you need to explicitly anchor the expression with ^ and $. PolicyCenter evaluates the `pattern` string using the Java regular expression engine. Thus, the regular expression must match the Java syntax. The Java syntax can have some minor differences from the JavaScript syntax, and, therefore, represent a slight deviation from the Swagger specification.	First non-null

Property	Type	Description	Combination style
		Only set a value for the `pattern` property if the parameter has a runtime type of `string`.	
`reqired`	`boolean`	If set to `true`, PolicyCenter requires this parameter on the request. If this parameter is missing from the request, PolicyCenter rejects the request with a 400 error Set this property to `true` for `path` parameters.	First non-null
`uniqueItems`	`boolean`	If set to `true`, it indicates that each member of the array must be unique, as defined by the `equals` and `hashCode` methods of the deserialized values. The default value is `false`. Only set a value for this property if the parameter is of type `array`.	First non-null

Guidewire extension properties

Property	Type	Description	Combination style
`x-gw-export-enumeration`	`boolean`	If set to `true`, PolicyCenter writes out the typekey values as an `enum` property while creating the Swagger schema for external clients. A value of `true` is only valid if the `enum` property is not set and `x-gw-type` is a typekey type. The default is `false`	First non-null
`x-gw-extensions`	`map<string, `*`AnyType`*`>`	The values in `x-gw-extensions` are available to the `IRestValidatorFactoryPlugin` plugin if you invoke the plugin to create custom validators for this parameter. This value can be an arbitrary map of property keys to values. The key values can be any string, and the *`AnyType`* values can be any object, including nested JSON objects.	Merge of extensions
`x-gw-type`	`string`	Defines the base JSON type for the parameter. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter deserializes the data into a Java object at runtime. Do not specify this value for `body` parameters.	First non-null

The Swagger Path Item object

The Path Item object describes the operations available on a single path. Guidewire does not support the following property on the Swagger Path Item object:

- `$ref`

Property	Type	Description	Combination style
`delete`	*Operations object*	The specified operation that responds to the DELETE HTTP method.	Merge
`get`	*Operations object*	The specified operation that responds to the GET HTTP method.	Merge
`head`	*Operations object*	Overrides the default HEAD method implementation	Merge
`options`	*Operations object*	Overrides the default OPTIONS method implementation	Merge
`parameters`	*Parameters object*[]	PolicyCenter assumes all parameters in the array list apply to all operations by default. It is possible to override these parameter definitions for individual operations by re-specifying a parameter with the same name and location	Merge by logical ID, which is `$ref`, if specified, otherwise, it is `name + in`.
`patch`	*Operations object*	The specified operation that responds to the PATCH HTTP method.	Merge

Property	Type	Description	Combination style
`post`	*Operations object*	The specified operation that responds to the POST HTTP method.	Merge
`put`	*Operations object*	The specified operation that responds to the PUT HTTP method.	Merge

Guidewire extension properties

Property	Type	Description	Combination style
`x-gw-cors-policy`	`string`	Property specifies the CORS policy for all operations on this path, overriding any value set for the `x-gw-cors-policy` property on the document root. The string value must be the name of a CORS policy defined by the `x-gw-policies` property on the root document.	First non-null

The Swagger Response object

The Response object describes a single response from an API operation. Guidewire does not support the following properties from the Swagger 2.0 specification on the Swagger Response object:

- `examples`

Required properties

Property	Type	Description	Combination style
`description`	`string`	**Required if not using `$ref`.** Documentation only.	First non-null

Required properties

Property	Type	Description	Combination style
`$ref`	`string`	References a response defined on the Swagger root document. It is not possible to combine this property with any other properties. Use the following syntax for the string value: `#/responses/`*`name`*	First non-null
`headers`	`map<string, header>`	The keys (`string` values) of the map are the names of custom HTTP headers that it is possible to return with the response.	Merge by key
`schema`	*Response schema object*	Defines the schema to which the response body must conform, if any.	First non-null

The Swagger Response Schema object

The Response Schema object defines a link to a JSON schema definition.

Required properties

Property	Type	Description	Merge style
`items`	*Schema items object*	**Required if value of `type` property is `array`.** Otherwise, do not use. If you set the value of the `type` property to `array`, then you must also set the `items` property to point to a Schema Items object that contains the actual members of the array.	N/A

Optional properties

Property	Type	Description	Combination style

`$ref`	`string`	Provides a link to the JSON schema definition defined in the file imported as `alias` using property `x-gw-schema-import`. For example, suppose that `x-gw-schema-import` contains the following key/value pair:
		`contact : gw.pl.contact-1.0`
		Then, the `$ref` reference string becomes the following string:
		`contact#/definitions/Contact`
		In this string, `Contact` references the definition defined in the `gw.pl.contact-1.0` JSON schema.
		PolicyCenter then uses the referenced schema to drive the serialization and validation of any JSON and XML types returned the API handler method.
		Use the following format for the string value:
		`alias#/definitions/name`
			N/A

`enum`	*anyType*[]	Documentation only. The `enum` values must be valid values of the type defined by the combination of `type`, `format`, and `x-gw-type`.	N/A
`format`	`string`	Use only if the `type` property is a scalar type. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter serializes scalar values returned by the API handler.	N/A
`type`	`string`	Specifies the type of the response body. Valid values are: • `array` • `boolean` • `integer` • `number` • `object` • `string` Only set this property to `object` if property `$ref` is also set to a value. If you set this value to `Array`, then you must also specify a value for the `items` property. For any other allowed value, the combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter serializes scalar values returned by the API handler method. For example, if `type=string` and `format=date-time`, then PolicyCenter serializes a `java.util.Date` object returned by the API handler method as an ISO 8601 Date.	N/A

Guidewire extension properties

Property	Type	Description	Merge style
`x-gw-export-enumeration`	`boolean`	If set to `true`, PolicyCenter writes out the typekey values as an `enum` property while creating the Swagger schema for external clients. A value of `true` is only valid if the `enum` property is not set and `x-gw-type` is a typekey type. The default is `false`.	N/A
`x-gw-type`	`string`	Use only if the `type` property is a scalar type. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter serializes scalar values returned by the API handler.	N/A

The Swagger Schema object

The Schema object allows for the definition of input and output data types. These types can be objects, but also primitives and arrays. Guidewire does not allow inline schema definitions. Thus, only the following properties are valid. You must specify at least one of these properties if you use this object.

Property	Type	Description	Combination style

`$ref`	`string`	**Required if type not specified.** Provides a link to the JSON schema definition defined the file imported as *`alias`* using property `x-gw-schema-import`. For example, suppose that `x-gw-schema-import` contains the following key/value pair:
		`contact : gw.pl.contact-1.0`
		Then, the `$ref` reference string becomes the following:
		`contact#/definitions/Contact`
		In this string, `Contact` references the definition defined in the `gw.pl.contact-1.0` JSON schema.
		If the operation consumes the `'application/json'` media type, PolicyCenter uses this schema to validate the incoming data.
		If the API handler method takes a `JsonObject` object or `JsonWrapper` object as the `body` argument, the operation requires the `$ref` property and PolicyCenter uses the referenced schema to deserialize the input data.
		The string must be one of the following formats:
		`alias``#/definitions/`*`name`*
			N/A

`type`	`string`	**Required if `$ref` not specified.** Defines the base JSON type for the schema. The `type` value must be one of the following: • `array` • `boolean` • `integer` • `number` • `string`	N/A

The Swagger Schema Items object

The Schema Items object provides the array members for the `items` property on the Response Schema object. The `type` property on the Response Schema object specifies the type of response body. If the value of the `type` property on the Response Schema object is `array`, then the `items` property on the Response Schema object must point to a Schema Items object that specifies the array members.

Optional properties

Property	Type	Description	Combination style
`$ref`	`string`	Provides a link to the JSON schema definition defined the file imported as *`alias`* using property `x-gw-schema-import`. Set this property in cases in which the API returns a top-level JSON array of data. For example, suppose the following circumstances exist: • The response schema has the following values set: `type=array` and `$ref=contact#/definitions/Contact`. • The API handler method returns an `Iterable<JsonObject>` In this case, PolicyCenter serializes the `Iterable` object as a JSON array, in which each array element is an object serialized and validated according to the `Contact` JSON Schema.	N/A
`enum`	*anyType*[]	Documentation only. The `enum` values must be valid values of the type defined by the combination of `type`, `format`, and `x-gw-type`.	N/A
`format`	`string`	Use only if the `type` property is a scalar type. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter serializes scalar values returned by the API handler.	N/A

Property	Type	Description	Combination style
type	string	Do not set this property if you set a value for `$ref`, unless the value of `$ref` is `object`. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter serializes scalar values returned by the API handler. Valid values are: • `boolean` • `integer` • `number` • `object` • `string`	N/A

Guidewire extension properties

Property	Type	Description	Combination style
x-gw-export-enumeration	boolean	If set to `true`, PolicyCenter writes out the typekey values as an `enum` property while creating the Swagger schema for external clients. A value of `true` is only valid if the `enum` property is not set and `x-gw-type` is a typekey type. The default is `false`.	N/A
x-gw-type	string	Use only if the `type` property is a scalar type. The combination of `type`, `format`, and `x-gw-type` determines how PolicyCenter serializes scalar values returned by the API handler.	N/A

The Swagger Security Requirement object

The Security Requirement object lists the security schemes necessary to execute this operation. It is possible to declare multiple security schemes, all of which are required. (That is, there is a logical AND between the listed schemes.)

Property	Type	Description	Merge style
name	string	The *name* value must match the name of a security scheme declared under the `securityDefinitions` property of the Swagger root object.	N/A

The Swagger Security Scheme object

The Security Scheme object provides a means to define a security scheme for use by the various REST API operations. Guidewire supports the following schemes:

- Basic authentication
- API key (either as a header or as a query parameter)
- OAuth2 common flows, which are implicit, password, application, and access code

Required properties

Property	Type	Description	Combination style
type	string	**Required.** Documentation only. Value must be one of the following: • `apiKey` • `basic` • `oauth2`	N/A

Optional properties

Property	Type	Description	Combination style
`description`	`string`	Documentation only.	N/A

Conditional properties

Property	Type	Description	Combination style
`authorizationUrl`	`string`	**Required if all of the following are true:** • `type` is `oauth2` • `flow` is either `implicit` or `accessCode` Documentation only. Do not use unless all the listed conditions are `true`.	N/A
`flow`	`string`	**Required if `type` is `oauth2`.** Documentation only. Use only if `type` is `oauth2`. Otherwise, do not use. The value must be one of the following: • `accessCode` • `application` • `implicit` • `password`	N/A
`in`	`string`	**Required if `type` is `apiKey`.** Documentation only. Use only if `type` is `apiKey`. Otherwise, do not use. If used, the value must be one of the following: • `header` • `query`	N/A
`name`	`string`	**Required if `type` is `apiKey`.** Documentation only. Use only if `type` is `apiKey`. Otherwise, do not use.	N/A
`scopes`	`map<string1, string2>`	**Required if `type` is `oauth2`.** Documentation only. Use only if `type` is `oauth2`. Otherwise, do not use.	N/A
`tokenUrl`	`string`	**Required if all of the following are true:** • `type` is `oauth2` • `flow` is either `accessCode` or `application` or `password` Documentation only. Do not use unless all the listed conditions are `true`.	N/A

The Swagger Tag object

The Tag object provides a means to add metadata to a single tag used by the Operation object. It is not necessary to have a Tag object for every tag on the Operation object.

Property	Type	Description	Combination style
`name`	`string`	**Required.** Documentation only.	N/A
`description`	`string`	Documentation only.	N/A
`externalDocs`	*External documentation object*	Documentation only.	N/A

The Swagger X-GW-CORS-policy object

The Guidewire Swagger schema currently allows only CORS policies to use external property substitution. You can set any of the properties in a CORS Policy object using the standard external property syntax, with properties under the `swagger` namespace. For example, the following code defines a complete CORS policy whose values you can substitute at runtime using the external properties provided by the `ExternalConfigurationProviderPlugin` plugin, with the string "swagger" prepended to each of the property names.

```
x-gw-cors-policies:
  account:
    enabled: ${cors.account.enabled:true}
    allowOrigins: ${cors.account.allowOrigins:any}
    allowMethods: ${cors.account.allowMethods:null}
    allowHeaders: ${cors.account.allowHeaders:null}
    allowCredentials: ${cors.account.allowCredentials:true}
    exposeHeaders: ${cors.account.exposeHeaders:null}
    maxAge: ${cors.account.maxAge:1200}
```

For information on the `ExternalConfigurationProviderPlugin` plugin, see the *System Administration Guide*.

Required properties

Property	Type	Description	Combination style
`allowOrigins`	`string`	Specifies which property types PolicyCenter allows for CORS requests. The value must be one of the following: • The character * • The string "any" • A comma-separated list of origins or regular expressions	First non-null

The string vale of the `allowOrigins` property must be one of the following values.

String value	Means
*	The "*" string indicates the following: • PolicyCenter allows all CORS request origins. • PolicyCenter returns the Access-Control-Allow-Origin response header as *. • Property `allowCredentials` is set to `false`.
any	The "any" string indicates the following: • PolicyCenter allows all CORS request origins. • PolicyCenter sets the Access-Control-Allow-Orign response header to the value of the Origin request header.
Comma-separated list	A comma-separated list provides a list of explicit origin values or regular expressions. PolicyCenter allows CORS requests for origins that match an element in the list, either exactly matching an allowed origin, or, matching a regular expression in the list. Use the following format for an origin string: `*scheme*://*domain*[:*port*]` If the element is an explicit origin, omit the optional *port* value (and colon) if the port is the default port for the scheme. PolicyCenter sets the value of the Access-Control-Allow-Origin response header to the value of the Origin request header.

Optional properties

Property	Type	Description	Combination style
`enabled`	`boolean`	If set to `false`, PolicyCenter disables CORS for any endpoints using this policy. The property defaults to `true` if you do not specify a value.	First non-null

Property	Type	Description	Combination style
`allowHeaders`	`string`	A comma-separated list of request headers to allow for CORS request headers: • If you do not specify a value, PolicyCenter allows all header types for CORS requests. • If you specify a value, PolicyCenter allows the following header types: ◦ Headers that the property explicitly specifies ◦ Standard CORS safe-listed headers ◦ Headers defined by Guidewire such as X-Correlation-ID and GW-Language	First non-null
`allowMethods`	`string`	A comma-separated list of methods to allow for CORS requests: • If you do not specify a value. PolicyCenter allows all methods. • If you specify a value, PolicyCenter permits only the specific methods that you identify.	First non-null
`allowCredentials`	`boolean`	If set to `true`, PolicyCenter adds the Access-Control-Allow-Credentials header to the CORS responses. If you do not specify a value, the default is `false`.	First non-null
`exposeHeaders`	`string`	A comma-separated list of header names to return as the value of the Access-Control-Expose-Header responses. If you do not specify a value (or specify `false`), PolicyCenter does not add the Access-Control-Expose-Header header to the responses.	First non-null
`maxAge`	`integer`	The value to return for the Access-Control-Max-Age header: • If you do not specify a value, the default value is 600. • If you specify -1, PolicyCenter removes the Access-Control-Max-Age response header entirely.	First non-null

The Swagger X-GW-Serialization object

This object provides information that affects how PolicyCenter serializes the response body. See "REST API responses" on page 14 for more information.

Property	Type	Description	Combination style
`includeEmptyArrays`	`boolean`	If set to `true`, PolicyCenter preserves empty arrays on serialization in `JsonObject` or `TransformResult` objects. If set to `false`, PolicyCenter treats a property as `null` if it contains an empty array. (This value then works in concert with the `includeNullProperties` option). If not specified, PolicyCenter uses the default behavior for the data type. This means that PolicyCenter treats this value as `true` for `JsonObjects` and `false` for `TransformResults`.	N/A
`includeNullItems`	`boolean`	If set to `true`, PolicyCenter preserves items with a `null` value in array properties on `JsonObject` or `TransformResult` objects and sets the corresponding output JSON arrays to have a `null` element. If set to `false`, PolicyCenter ignores elements with a `null` value. This value works in concert with the `includeEmptyArrays` option. Thus, if this value is false and an array contains only null `items`, PolicyCenter handles the array as if it were empty. The default value is `false`. Setting this to value to `true` is only valid if the items that have null values also specify `x-gw-nullable` on the associated JSON Schema.	N/A

Property	Type	Description	Combination style
`includeNullProperties`	`boolean`	If set to `true`, PolicyCenter preserves properties with `null` values on `JsonObject` or `TransformResult` objects and sets those properties to `null` in the output. If set to `false`, PolicyCenter omits properties with a `null` value in the output. The default value is `false`. Setting this value to `true` is only valid if the properties that have `null` values also specify `x-gw-nullable` on the associated JSON Schema.	N/A

Combining Swagger schema files

Guidewire provides a mechanism for creating Swagger schemas that combine together multiple pieces to form a single, logical resulting schema document. It is possible to use this combination mechanism for a number of different purposes. The following list describes the possible ways that you can combine Swagger schemas.

Simple hierarchical extension	Works by extending an application schema, which, in turn, can extend a platform schema.
Content composition	Works by combining together several different content pieces. For example, it is possible to combing LOB-specific schemas, along with their own extensions, and, perhaps, a base schema.
Schema aggregation	Works by breaking a large, complex schema into multiple files for maintenance by different teams. You can then aggregate the individual files together again for publishing as a single logical API with a single, consistent base URL and version.
Minor version change	Works by creating a new minor version of a schema that extends the previous major or minor version. This new version can minimize schema duplication and ensure the changes are backwards-compatible
API template	Works by implicitly adding the API template to the list of combined schemas for a given, published schema.

One way to think about schema combination is as a well-defined, ordered, textual merge of the documents. For example, suppose that schema A combines schemas [B, C], listing the two schemas in that order in the `x-gw-combine` property. After combination, schema A essentially looks exactly as if you did the following:

- Merged the contents of schema B on top of schema C, overwriting any properties common to both documents with those in schema B.
- Merged the contents of schema A on top of the first result, again overwriting any properties that schema A respecified.

The result is that schema A can override items in either schema B or schema C, and can add entirely new items, but the merge cannot remove any items from either schema B or schema C. Schema B can also behave the same way to schema C because of the ordering defined in `x-gw-combine` ([B, C]). However, ideally, any two schemas that you want to combine in this manner are disjoint.

After the schema combination happens, PolicyCenter then validates the resulting combined schema against its component pieces to ensure that the changes are logical extensions or otherwise backwards compatible. In this example, PolicyCenter needs to validate schema A against schema B and against schema C. Thus, if schema A changes the type of a parameter that was defined in Schema B, PolicyCenter reports that as an error. And, if schema A introduced a new validation constraint on top of a parameter defined in schema B, PolicyCenter reports that as a warning.

Schema naming

One important consequence of schema combination is that PolicyCenter does not modify the original schemas. Schema A is a different schema, with a different name, from either schema B or schema C. The end result can also have an entirely different versioning schema. For example, you can extend a schema version `gw.pc.policy-10.0` as `mycompany.pc.policy-1.0`.

It is also possible to extend the same schema an arbitrary number of times. For example, you can extend `gw.pc.policy-1.0` as `myCompany.pc.policy-1.0`, `myCompany.pc.policy-2.0`, and so on.

For this reason (among others), Guidewire requires that you publish your schema files explicitly in `published-apis.yaml`. If you extend a base configuration schema, you mostly likely want to publish your extension only, not the base configuration schema.

Any schema that you create based on a Guidewire schema does not automatically pick up updates to the schema provided by Guidewire. If you want to pick up changes in a Guidewire-provided schema, you must explicitly migrate your changes to the new Guidewire schema.

Swagger file combination ordering

Although it theoretically possible to construct any arbitrary chain of schema combination, it is important that you do not create cycles within your schema extensions. Thus, do not do the following:

- A combines with B.
- B combines with C.
- C combines with A.

PolicyCenter combines the various schemas in the order in which each appears in the `x-gw-combine` property. PolicyCenter also loads and combines all referenced schemas before processing the full schema chain.

Example

To provide a concrete example, suppose that schema A and schema A' have the following designated meanings:

- Schema A - The file as it is on disk.
- Schema A' - The logical schema after PolicyCenter combines the file with everything that it references using property `x-gw-combine`.

Thus, if schema A combines [B, C] and B and C both combine with D, PolicyCenter produces A' from A using the following sequence:

1. PolicyCenter loads B' (B combined with D) and C' (C combined with D) into memory.
2. PolicyCenter creates a list of the root objects from A, B', and C', in that order.
3. PolicyCenter follows the full document tree, starting with these root objects, according to the merge rules defined in “Guidewire Swagger specification” on page 83.

The root objects always exist in all three schemas (A, B, C). However, it is possible that a given logical object appears in one, two, or all three of the schemas. For each collection of matching objects, PolicyCenter always considers A (if present), then B' (if present), and finally C' (if present), in that order. For example, suppose C' defines a `path` property of `"/contacts/{contactId}"` with a `GET` operation. If C' is the only schema that defines that operation, A inherits the definition from C' exactly as it is. If B' also defines that operation, B's definition of the property overrides the definition in C'. If A' also defines that operation, A then overrides any properties also specified in B' or C'.

Swagger file combination rules

The exact rules that PolicyCenter uses to combine multiple schemas depends on the specific Swagger object and property. Thus, for a full understanding, you need to consult the documentation on each individual object and property. As a general rule, though, the vast majority of properties and objects work as described in the following sections.

Scalar values

A scalar value is a simple string, number or Boolean value. If there is a given set of objects defined in more than one schema, PolicyCenter chooses the first non-null value for the scalar property that it encounters in the schema files. For example, if A, B', and C' all define the `description` property on the same logical parameter, PolicyCenter uses the value defined in A for the property in A'. However, if A specifies the parameter, but not the `description` property, PolicyCenter uses the value of the `description` property from B'. Or, PolicyCenter uses the value from C' if it is the only schema that specifies the value.

Leaf nodes

PolicyCenter sometimes treats the following types of objects as a leaf nodes in the document tree:

- Contact objects
- License objects
- External Documentation objects
- Security Definition objects
- Security Requirement objects
- Schema Security objects

In such cases, PolicyCenter treats the object in the same fashion as a scalar value. This means that PolicyCenter does not try to merge the values but takes the first non-null value it finds.

Complex objects

PolicyCenter merges more complex objects (arrays, for example), across the different files.

Equivalent objects

PolicyCenter uses a fairly intuitive process to determine which objects to consider equivalent and thus to merge together, for example:

- PolicyCenter merges operations that have identical verbs (`GET` or `POST`, for example) on identical paths.
- PolicyCenter merges parameters on operations if the parameters have the same `$ref` values, or, if the parameters have the same combination of `name` and `in` values, and so on.

Validating Swagger schemas

The general philosophy in validating Swagger schema combination is that the schema that results from a combination of files needs to be a logical, backwards-compatible extension of the combined pieces. For example, if schema A combines with schema B, Guidewire requires the following restrictions:

- Schema A must not change the type of a parameter defined in schema B.
- Schema A must not introduce new, required parameters to operations defined in schema B.
- Schema A can remove or loosen validation constraints on input parameters defined in schema B, but schema A must not add or tighten validation constraints

However, even with these restrictions on changes to schema A, it is possible that custom code written against schema B does not work against schema A. The reason is that requests that are valid with respect to B are not necessarily valid against A. However, in general, most validation errors that occur during schema combination are warnings rather than hard errors, as there may be no way to avoid the error. For example,

Schema verification errors

Standard schema validation occurs after schema combination happens. Thus, validation rules such as the following all happen in the context of the combined schemas:

- Which fields are required
- Which fields can or cannot be used in combination
- Which parameter, response, or schema references are valid

As schema validation takes place after schema combination, it can sometimes make tracking down the root source of a problem more complicated. For example, if a parameter specifies an invalid combination of `type`, `format`, and `x-gw-type` properties, the error can be the result of the combination of schema A with schema B. In this case, the validation error messages report the line numbers of the parameter from all files that contained that parameter object.

Global overrides

As part of schema composition, it is possible to extend or compose the JSON schemas referenced from body parameters and responses. To make this kind of change, use the `x-gw-schema-import` property to make the change globally.

Schema references for parameters and responses are always relative to an alias. Thus, it is possible to override the alias to point a different JSON schema. For example, suppose that the `contact_base-1.0` Swagger schema defines the `contact` alias as pointing to schema `contact_base-1.0`. The `GET /contacts/{contactId}` method can then have a response schema `$ref` value of `contact#/definitions/Contact`, which then resolves to the `Contact` definition within the `contact_base-1.0` schema.

Example

The following sequence of steps illustrate how to create a `contact_ext-1.0` Swagger schema that extends the return schema for `GET /contacts/{contactId}`.

1. Create the `contact_ext-1.0` JSON schema, combine `contact_base-1.0`, and extend the `Contact` schema definition to add in the new properties.
2. Create the `contact_ext-1.0` Swagger schema and combine the `contact_base-1.0` schema.
3. Add the following import to the `contact_ext-1.0` schema to override the `contact` alias to point to the extension schema.

```
x-gw-schema-import
  contact : contact_ext-1.0
```

The result is that all references inherited from `contact_base-1.0` that reference the `contact` alias now point to the equivalent definitions in the `contact_ext-1.0` JSON schema instead. During combination, PolicyCenter compares the JSON schemas referenced from each `body`-type parameter or response schema to ensure they are appropriately backwards compatible.

Handler composition

Composing schemas together also requires that the API handler classes be compositional as well. Use the `x-gw-apihandlers` property on the root object to list the API handler classes that you want to compose together. The property takes an array of strings, each of which is a valid Gosu or Java class name. To determine the method for any given operation, PolicyCenter searches the listed classes, in order of listing, for the method.

It is useful to having multiple handler classes for multiple reasons.

1. You can split the implementation of a large, complex API across several logical classes.
2. If creating a Swagger schema that primarily serves to aggregate several other schema files together, you can simply list all the handler classes associated with those component schemas.
3. If a Swagger schema is purely additive relative to the combined schemas, you can simply create a new handler class for the new operations, and then list both that new handler class and the handler classes for the combined schemas.
4. If you need to override only one or two methods from an existing handler class, you can:
   a. Create a new class with the same method names.
   b. List both your new class and the existing handler class in property `x-gw-apihandlers`.
   c. Place the new class first in the class list so that PolicyCenter uses its methods preferentially

In addition, PolicyCenter looks preferentially for explicitly declared methods before looking at inherited methods. This behavior makes it easier to use class inheritance to extend an existing handler class.