IBM Security

IBM

IBM Resilient SOAR Platform
Custom Action Developer's Guide
V33

Date: August 2019

acknowledgment

Resilient Security Orchestration, Automation and Response (SOAR) Platform User Guide

Platform Version	Publication	Notes
33.0	August 2019	Initial publication.

Contents

Chapter 1. Objective

This guide provides the information to integrate the Resilient Security Orchestration, Automation and Response (SOAR) Platform with your organization's existing security and IT investments using the functions feature. Integrations makes security alerts instantly actionable, provides valuable intelligence and incident context, and enables adaptive response to complex cyber threats.

This guide is intended for programmers, testers, architects and technical managers interested in developing and testing extensions with the Resilient platform. It assumes a general understanding of the Resilient platform, message-oriented middleware (MOM) systems, and a knowledge of writing scripts in Python.

Chapter 2. Overview

A *custom action* is a type of extension that allows the Resilient platform to send a snapshot of the incident data automatically to external code, which can then act upon the data to perform integration work, and, optionally, send data to the Resilient platform.

You should familiarize yourself with the Resilient architecture and the relevant Resilient features, as described in the following sections, before designing and writing custom action integrations.

Resilient SOAR Platform

The Resilient platform is a central hub for incident responses. It is customizable so that it can be tailored to meet the needs of your company or organization. The focus of these customizations is the dynamic playbook, which is the set of rules, conditions, business logic, workflows and tasks used to respond to an incident. The Resilient platform updates the response automatically as the incident progresses and is modified.

You should be familiar with your organization's customized Resilient playbook when designing a custom action. In particular, you should be familiar with the following playbook components:

- Rule. A set of conditional statements that identify relationships and run responses accordingly. Rules define a set of activities that are triggered when conditions are met. Activities include setting incident field values, inserting tasks into the task list, launching workflows, running internal scripts to implement business logic and placing items on message destinations to be acted upon by remote programs.
- Workflow. A graphically designed set of activities that allows you to create a complex set of operations. You can use workflows to implement sophisticated business processes that can be invoked by rules. Workflows can contain various components, such as scripts, functions, and message destinations.
- Script. For users familiar with writing Python scripts, you can write scripts to access incident data (same data as accessed by rules) then perform activities more complex than can be handled by rules. Scripts can be triggered by rules or workflows.
- Message destination. The location where data is posted and made accessible to remote programs. The message includes details about an object and the activity taken. You can configure rules, workflows and functions to send messages to one or more message destinations.
- Custom field. Design element used in incident layouts to capture specific data. An integrated system can populate a custom field.
- Data table. Design element that organizes data in a tabular format. An integrated system can populate the table. Depending on the integration, a Resilient user may be able to access limited capabilities of that security program directly from a row in the data table.
- Artifact. Data that supports or relates to an incident. The Resilient platform organizes artifacts by type, such as file name, MAC address, suspicious URL, MD5 and SHA1 file hashes, and more. An integrated system can send an artifact to a Resilient incident.

For more information about the Resilient platform and dynamic playbooks, refer to the *Playbook Designer Guide*.

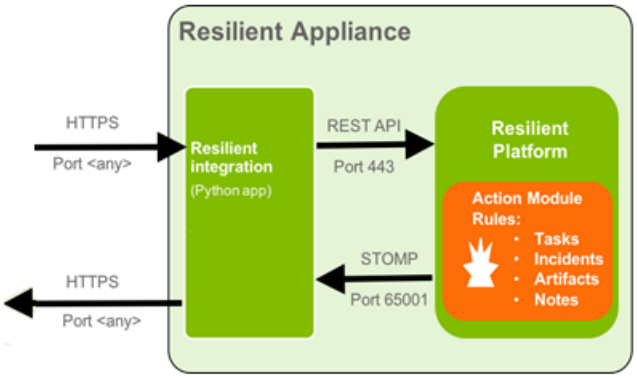
Integration architecture

The Resilient platform has a full-featured REST API that sends and receives JSON formatted data. It has complete access to almost all Resilient features, including but not limited to: creating and updating incidents and tasks, managing users and groups, and creating artifacts and attachments.

To perform an integration with functions, your Resilient platform must subscribe to the Action Module. This is an extension to the Resilient platform that allows implementation of custom behaviors beyond

what is possible in the Resilient internal scripting feature. It is built on Apache ActiveMQ. The STOMP message protocol is used for Python based integrations. Custom behaviors are triggered by adding a message destination to a rule defined in the Resilient platform and subscribing your integration code to that message destination.

The following diagram shows the relationship between the integration component, REST API, Action Module and Resilient platform.



The Resilient platform contains an interactive Rest API browser that allows you to access the Resilient REST API and try out any endpoint on the system. When logged into the Resilient platform, click on your account name at the top right and select **Help/Contact**. Here you can access the complete API Reference guide, including schemas for all of the JSON sent and received by the API, and the interactive Rest API.

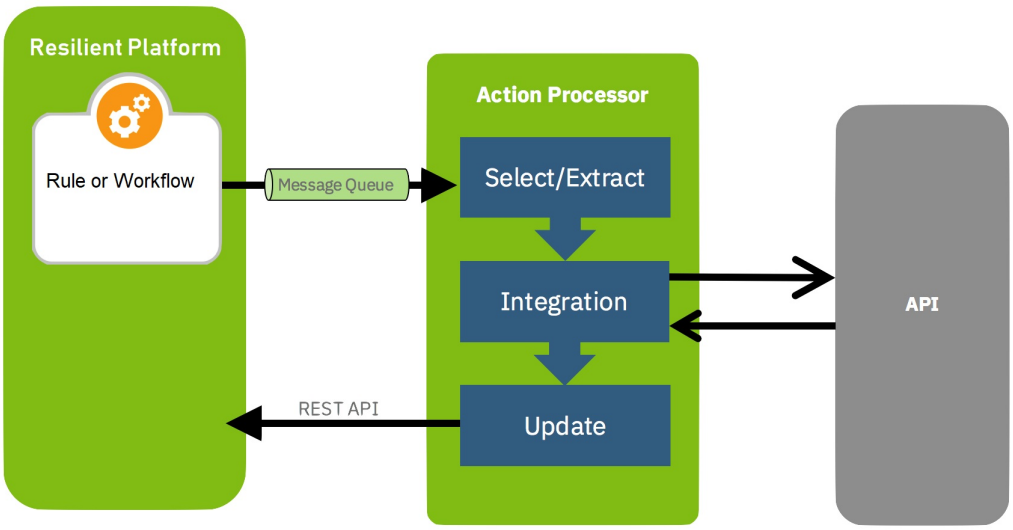
For information about JSON structures used in the Resilient API, see Chapter 6, “JSON structures in the Resilient API,” on page 35 later in this guide.

Custom actions

Custom action is a type of integration that allows the Resilient platform to send a snapshot of the incident data automatically to external code, called *action processors*. This external code can then perform integration work, for example:

- Perform a lookup for information about a user or machine in an asset database then update a Resilient data table with the result.
- Search SIEM logs for additional information related to an IP address, a URL or a server name, create a file with the result, and add the file as an artifact to a Resilient incident.
- Use information from the incident, task or artifact to open a ticket in an ITSM system, then track the ticket for updates.

When a Resilient rule or workflow fires, it sends data about its object to a message destination. The action processor retrieves that message, acts on it, and updates the Resilient platform with the result.



Resilient Circuits

Resilient Circuits is a Python circuits framework that automatically manages authenticating and connecting to the STOMP connection and REST API in the Resilient platform. It simplifies creating integrations by allowing you to focus on writing the behavior logic. It is the preferred method for writing integrations.

The Resilient Circuits framework makes it simple to develop and deploy action processors using Python. When using the Resilient Circuits framework, the action processor component is a Python class that implements *message handlers*. These handlers are called by the framework when an action message arrives on a message destination.

You can use Resilient Circuits to manage your functions as well as other types of integrations. Each integration has its own section in the app configuration file. This file stores information about the Resilient platform, such as user credentials, as well as variables for your functions.

If you wish to create your custom actions in another programming language, you can use the API directly instead of using Resilient Circuits.

Development Overview

The following list provides a high-level overview of the development process. The subsequent sections in this guide provide the details.

- Before you write a custom action processor, you must understand its purpose, the data it needs from the Resilient platform, and the actions or decisions to be made based on the results.
- Determine whether to use Resilient Circuits. If writing in Python, this is the preferred method and the Resilient Circuits framework can simplify your development.
- If using Resilient Circuits, make sure you have Resilient Circuits framework installed and configured on a Resilient Integration Server, as described in the *Integration Server Guide*.
- At the Resilient platform (preferably in a test environment), define and implement the message destination and other components, such as rules, workflows, or both.
- Write the action processor, which includes your integration code. If you have access to integrations that are similar to the one you wish to create, use that integration as a template to save time.
- Test the integration by triggering the workflow and checking the results.
- Package your integration and make it available for deployment to the Resilient platform in your production environment.

Developer websites and documentation

The Resilient developer web site contains the core Resilient helper module and Resilient Circuit packages, additional integration packages, documentation and examples. The links are provided below.

- IBM Resilient Success Hub. If you have not already, use this link to request access.
- IBM Resilient Developer website. Provides overview information and access to various areas of development, such as developing playbooks and publishing integrations.
- IBM Resilient Github. Provides access to library modules, community-provided extensions, example scripts, and developer documentation. It also contains the Resilient Circuits and helper module packages. This is also accessible from the developer website reference page.
- IBM X-Force App Exchange. Provides access to the Resilient community apps on IBM X-Force.
- Releases. Lists the apps by Resilient platform release. You can also download from this page.

You can access the Resilient product documentation, including the REST API Reference guide, from the **Help/Contact** page, which is accessible by logging in to the platform and clicking the menu icon next to your user name.

In addition, you can view the Resilient product guides, such as the Playbook Designer Guide, and additional information in the IBM Knowledge Center. (This link takes you to a page where you can choose the version of the Resilient platform.)

Chapter 3. Creating Resilient components

Regardless of whether you use Resilient Circuits or the API directly, you need to create the Resilient rules, workflows, message destinations and any custom fields or data tables that will interact with your custom action.

There are several ways to trigger your custom action:

- A menu item rule that posts the transaction to a message destination. When the conditions are met, the rule adds an action to the Action menu of its object, such as an incident, task, or artifact. With a menu item rule, you can add Activity Fields for additional user input, which is also sent with the message.
- An automatic rule that posts the transaction to a message destination. The message is sent when an object, such as an incident, task or artifact, is created or modified and meets the conditions that you specify. For example, you might automatically send “IP Address” artifacts to a particular destination if the incident is not yet triaged.
- A workflow that posts the transaction to a message destination. Workflows provide flexibility in how these custom actions are coordinated, and are ideal for complex scenarios including task completion, decision logic, scripts, and timers. If you create a workflow, you should also create one or more rules that call the workflow.

Determine which method is best for you then perform the following to create the Resilient components. For detailed procedures, see the *Playbook Designer Guide*.

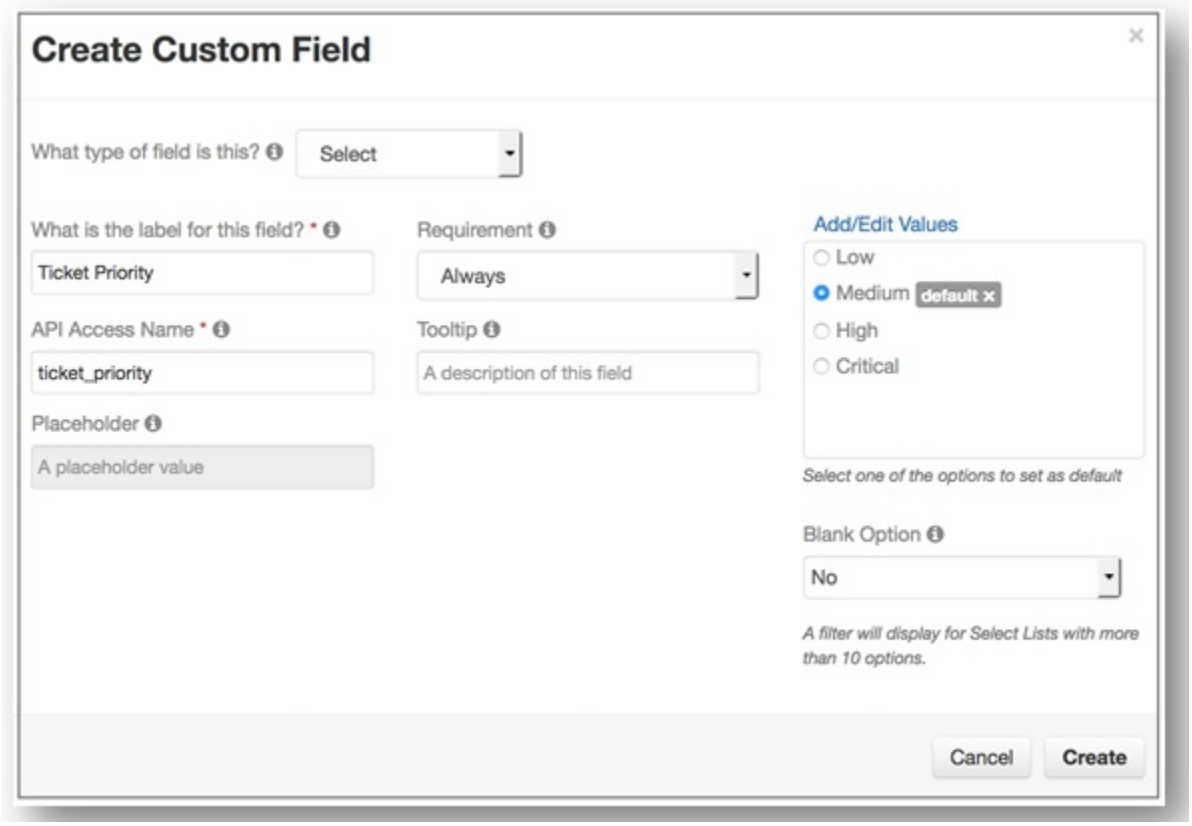
1. Log in to the Resilient platform as a user with permission to view and modify the customization settings.
2. Go to the Message Destinations tab and create a message destination as follows:
   a. Set the Type to **Queue**.
   b. Set **Expect Acknowledgement** to Yes.
   c. Add the Resilient account that you use for integrations as an authorized user in the **Users** field.
3. If you require custom fields to gather or receive specific data for your integration, perform the following:
   a. Go to the **Layouts** tab.
   b. Determine where to place the fields by selecting the tab or **New Incident Wizard**.
   c. Create the fields. Take note of the API Access name of each field for use in your code.
4. If you require a data table to receive data from the integration, perform the following:
   a. Go to the **Layouts** tab.
   b. Determine where to place the data table by selecting the tab.
   c. Create the data table. Take note of the API Access name of the data table for use in your code.
5. If using a rule, go to the Rules tab and create the rule. Configure it as an automatic or menu item rule. Make sure that the rule includes your message destination. Note the programmatic name of the rule, which is the same as the display name with underscores instead of spaces.
6. If using a workflow, go to the Workflows tab and create a workflow. Make sure to add the message destination at an appropriate place in the workflow. Note the programmatic name of the workflow.

Menu Item Rules and Activity Fields

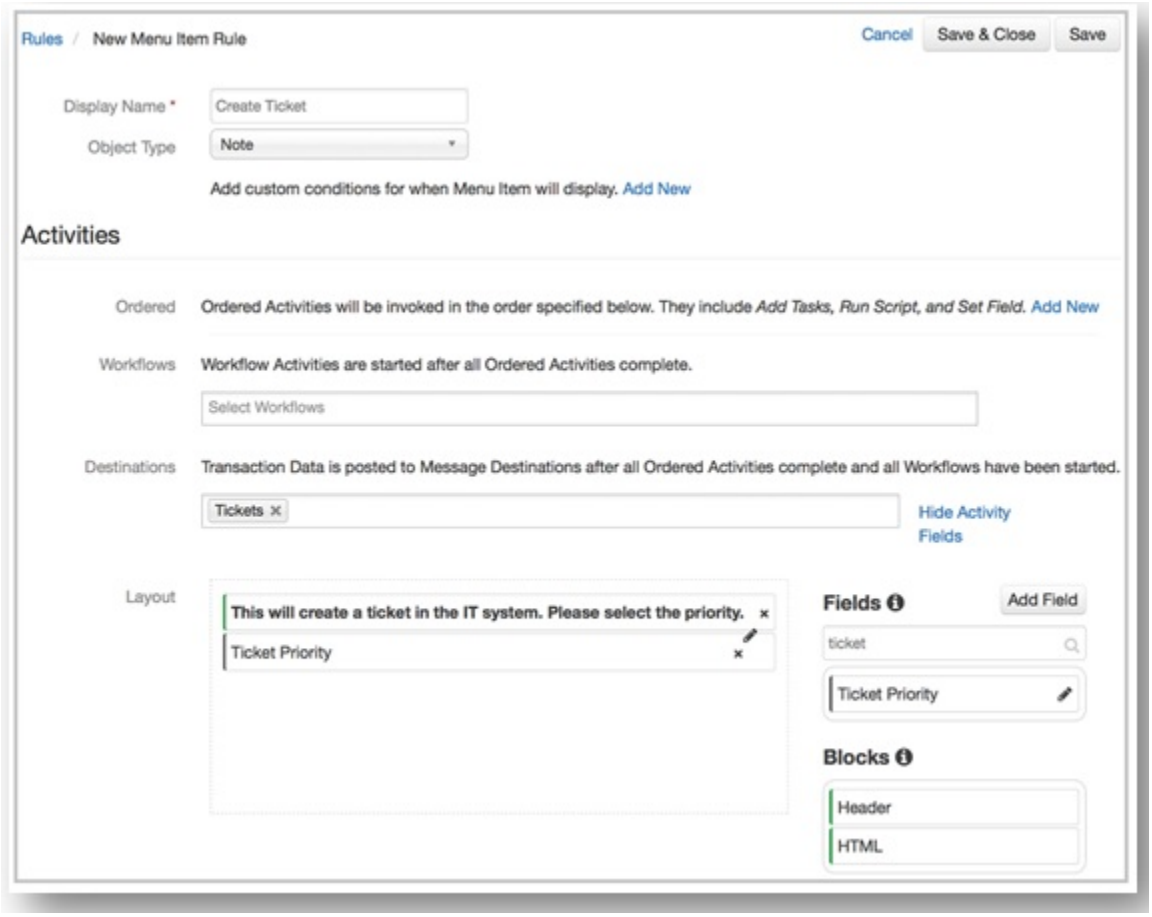
The menu item rule displays as an action in an object’s **Actions** drop-down menu and executes only when a user invokes it. In some cases, it is necessary for the user to enter additional information when selecting an action. For example, if you are developing a “Create ticket” action, you may need to allow the user to

select a priority for the ticket that is to be created. You do this by creating an activity field. Activity fields are managed through the Resilient platform as part of the menu item rule. Refer to the *Platform Playbook Designer Guide* for details.

The following figure illustrates the creation of a Ticket Priority field that is added to a menu item rule. It is a select field that has four values, Low, Medium, High and Critical. The field value is required.



Once you create the Ticket Priority field, you drag it to the rule's layout as shown in the next figure. You can also create a header that gives the user some additional information. Once you create a field, it is available to all other menu item rules.


Rules / New Menu Item Rule
Cancel
Save & Close
Save
Display Name *
Create Ticket
Object Type
Note
Add custom conditions for when Menu Item will display. Add New
Activities
Ordered
Ordered Activities will be invoked in the order specified below. They include Add Tasks, Run Script, and Set Field. Add New
Workflows
Workflow Activities are started after all Ordered Activities complete.
Select Workflows
Destinations
Transaction Data is posted to Message Destinations after all Ordered Activities complete and all Workflows have been started.
Tickets
Hide Activity Fields
Layout
This will create a ticket in the IT system. Please select the priority.
Ticket Priority
Fields
Add Field
ticket
Ticket Priority
Blocks
Header
HTML

Chapter 4. Using Resilient Circuits

The Resilient Circuits framework makes it simple to develop and deploy custom action processors using Python. An action processor component, in this framework, is a Python class that implements *message handlers*. These handlers are called by the framework when a message arrives on a message destination.

Prerequisites

Before starting, make sure your environment meets the following prerequisites:

- Resilient platform V31 or later with the Resilient Action Module.
- If using Resilient Circuits, a Resilient integration server where you deploy and run the integration code. See the Integration Server Guide for the information to install and configure Resilient Circuits.
- You have created the components you need, as described in Chapter 3, "Creating Resilient components," on page 7.
- IBM Resilient recommends that you use a Resilient platform in a test environment to create the function, message destination, rules, workflows and other components needed for your integration. Once tested, you can deploy the integration into any Resilient platform that is at the same or later version as your test platform.

If you have access to an integration that is similar to the one you wish to create, you can use that integration as a template to save time.

Create a directory

Before writing the action processor, perform the following:

1. Create a directory on your integration server. This is the directory where Resilient Circuits looks to load your module.
2. Add the absolute path to the directory in your app.config file.

For example, create a directory called "components" and then add a value for "componentsdir" to your app.config file and set it to the absolute path of this components directory.

NOTE: If on a Windows system and you edit the app.config file with Notepad, please ensure that you save it as type **All Files** to avoid a new extension being added to the filename, and use UTF-8 encoding.

Write the action processor

The action processor is a Python module. Perform the following:

1. Create a Python module in your components directory that contains your integration code. You can use any of the example Resilient Circuits component modules as a starting point. The module name and component class name may be anything you wish, but it is advisable to give them a name reflective of their behavior or purpose.
2. Set the channel member to the programmatic name of the message destination.
3. Rename the _framework_function method of your class to something descriptive for what the action should do.
4. Match the @handler to the API name of the rule or workflow.

5. Update the handler code to perform your desired actions. You can update incident fields, add tasks or artifacts, or anything else supported by the Resilient REST API. For details, see “Add functionality with decorators” on page 15.
6. Make sure to end your logic by yielding a status string. This is used for the action status message in the Resilient platform. For example:

```
yield StatusMessage “Task added successfully”
```

7. Add a section to your app.config file with a name that is reflective of your component to store configuration values. Put any configuration values you need here.
8. Update the CONFIG_DATA_SECTION variable in your module with the name of the section you created. For example: CONFIG_DATA_SECTION = “example_action”

The following example is a simple script that is a component that subscribes to a message destination named “example”, and handles a rule named “example_action”. The name of the Python class (in this example, “MyExampleComponent”) is not important, nor is the filename.

In the __init__ method of the component class, the “channel” member is set to the programmatic name of the message destination you created. The class has a method, decorated with **`@handler()`** that determines the action(s) to be sent to the Resilient platform.

```
# Simple example component for resilient-circuits

import json
import logging
from circuits.core.handlers import handler
from resilient_circuits.actions_component import ResilientComponent, ActionMessage

logger = logging.getLogger(__name__)

class MyExampleComponent(ResilientComponent):

    # Subscribe to the Action Module message destination named "example"
    channel = "actions.example"

    @handler("example_action")
    def _example_handler_function(self, event, *args, **kwargs):
        # This function is called with the action message,

        # In the message we find the whole incident data (and other context)
        incident = event.message["incident"]
        logger.info("Called from incident {}: {}".format(incident["id"], incident["name"]))
```

The handler function can access additional context, for example:

```
        # The message also contains information about the user who triggered the action
        who = event.message["user"]["email"]

        # Post a new artifact to the incident, using the provided REST API client
        new_artifact = {
            "type": "String",
            "value": "Test artifact from {}".format(who)
        }
        new_artifact_uri = "/incidents/{}/artifacts".format(incident["id"])
        self.rest_client().post(new_artifact_uri, new_artifact)
```

Any string returned by the handler function is shown to the Resilient user in the Action Status dialog:

```
return "Action Processed OK"
```

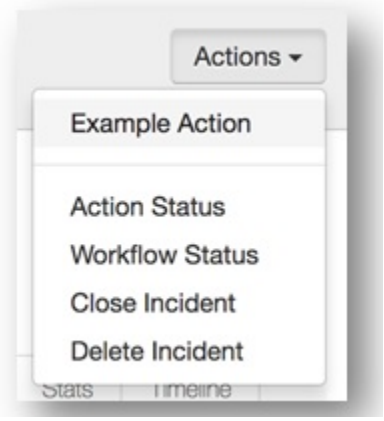
Run the action processor

Run the integration code from the command-line, with **resilient-circuits run**. The framework reads your configuration file, connects to the Resilient platform, finds and loads your components, then subscribes

to the message destination for each action processor component. Leave it running; when an event is triggered, the code handles it.

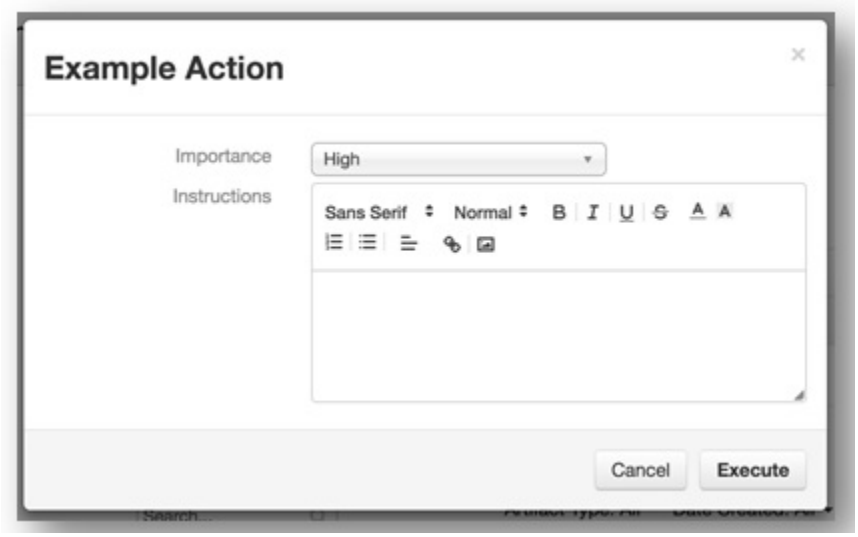
```
resilient-circuits run
2017-11-21 09:23:52,288 INFO [app] Configuration file: /home/integration/.resilient/app.config
2017-11-21 09:23:52,291 INFO [app] Resilient server: culture.example.com
2017-11-21 09:23:52,293 INFO [app] Resilient user: api@example.com
2017-11-21 09:23:52,295 INFO [app] Resilient org: Special Circumstances
2017-11-21 09:23:52,296 INFO [app] Logging Level: INFO
2017-11-21 09:23:52,840 INFO [app] Components auto-load directory: /home/integration/components
2017-11-21 09:23:52,857 INFO [stomp_component] Connect to culture.example.com:65001
2017-11-21 09:23:52,966 INFO [app] App Started
2017-11-21 09:23:52,969 INFO [actions_component] Component registered to actions.example
2017-11-21 09:23:52,970 INFO [component_loader] Loaded and registered component 'example'
2017-11-21 09:23:52,971 INFO [actions_component] STOMP attempting to connect
2017-11-21 09:23:52,972 INFO [app] Components loaded
2017-11-21 09:23:52,973 INFO [stomp_component] Connect to Stomp...
2017-11-21 09:23:52,974 INFO [client] Connecting to culture.example.com:65001 ...
2017-11-21 09:23:53,069 INFO [client] Connection established
2017-11-21 09:23:53,221 INFO [client] Connected to stomp broker
 [session=ID:culture-40894-1508509684399-5:81, version=1.2]
2017-11-21 09:23:53,223 INFO [stomp_component] Connected to failover:(ssl://
culture.example.com:65001)?maxReconnectAttempts=1,startupMaxReconnectAttempts=1
2017-11-21 09:23:53,224 INFO [stomp_component] Client HB: 0  Server HB: 15000
2017-11-21 09:23:53,225 INFO [stomp_component] No Client heartbeats will be sent
2017-11-21 09:23:53,226 INFO [stomp_component] Requested heartbeats from server.
2017-11-21 09:23:53,229 INFO [actions_component] Subscribe to message destination 'example'
2017-11-21 09:23:53,230 INFO [actions_component] STOMP connected.
2017-11-21 09:23:53,232 INFO [stomp_component] Subscribe to message destination
 actions.203.example
```

In this example, the “Example Action” can be found on the **Actions** menu at the top right of the incident.



If your menu item is for artifact objects, you can find the menu available from the “...” action button beside the artifact; similarly for tasks, notes, and so on. For details, see the *User Guide*.

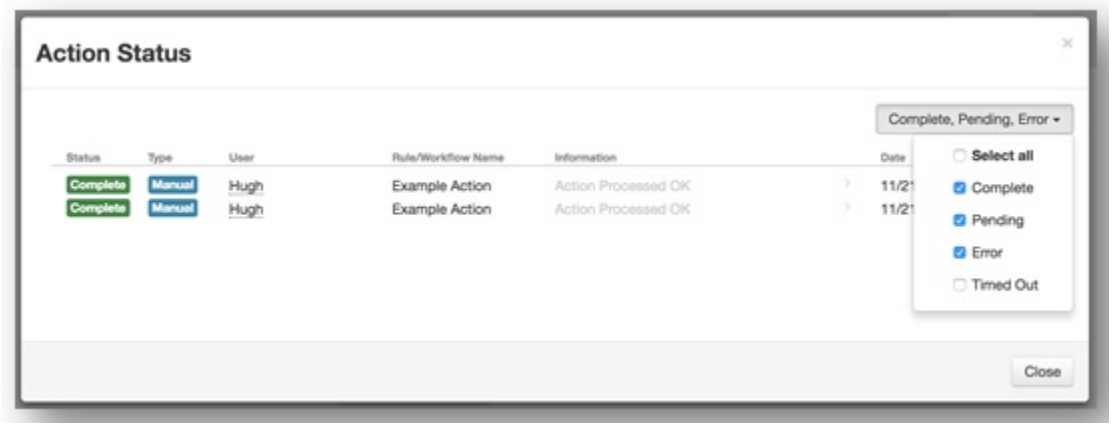
Select **Example Action** from the Actions menu. Where this is a menu item rule in the example, the Resilient user is prompted to enter the fields as shown below. When the user clicks **Execute**, the Resilient platform sends the data to the message destination.



At the integration console, you can see the message arrive, including the logging message to print the incident name as part of the example code.

```
2017-11-21 09:24:23,235 INFO [actions_component] Event:  Channel: actions.example
2017-11-21 09:24:23,237 INFO [example] Called from incident 2496: The New Incident
```

The Action Status menu shows whether each action is pending (queued for delivery to the action processor), processed successfully, or with an error. Here you can see that the action completed with success, and included a status message.



Run during development

During development, it would be inconvenient to have to re-install your package every time you want to test a change. Fortunately, you can install your project in “unbuilt” mode, which links directly against the source code in your project directory rather than installing a copy in site-packages. Now your changes take effect immediately with no need to re-install. There are two ways to do this. From within your project directory (at the same level as your setup.py script), run one of the following commands:

```
python setup.py develop
```

or

```
pip install -e .
```

This creates an “egg-info” directory in your project directory and links your site-packages to it.

While developing your Resilient Circuits integration, it is very useful to be able to run it from your IDE (PyCharm and so on) so you can use tools like a debugger.

Instead of the “resilient-circuits run” command that you would normally use at the command line, have your IDE run Resilient Circuits with the command “python resilient-circuits/resilient_circuits/app.py”. This is best used in combination with the “develop” installation mode. If you have not packaged your integration, make sure the “componentsdir” parameter is set correctly in your app.config file to point to the directory containing the component you are developing.

Add functionality with decorators

Resilient Circuits provides various Python “decorators” that you can use to add functionality to the handler functions in your component.

required_field

The required_field class decorator allows you to require that a custom field with a particular name is present in the Resilient platform. If that field does not exist, the component fails to load and provides an appropriate error message.

Sample Usage:

```
@required_field("last_updated")
class SetLastUpdated(ResilientComponent):
    """Set a last updated timestamp on incident"""

    @handler("incident_updated")
    def _set_last_updated(self, event, source=None, headers=None, message=None):
        inc_id = event.message["incident"]["id"]
        timestamp = int(headers.get("timestamp"))
        def update_func(inc):
            inc["properties"]["last_updated"] = timestamp
            return inc
        self.rest_client().get_put("/incidents/%d" % inc_id, update_func)
        yield "last_updated set"
```

required_action_field

This class decorator allows you to require that an activity field with a particular name is present in the Resilient platform. If that field does not exist, then the component fails to load and provides an appropriate error message. Its usage is the same as for the required_field decorator.

defer

This method decorator allows you to postpone handling an action for a specified number of seconds. This is useful for situations where you need to accommodate a delay in the availability of a resource. For example, allowing time for incident updates to be reflected in the Resilient newsfeed before querying that API endpoint. The defer decorator should be placed ABOVE the handler decorator on your method.

The defer decorator works only with handlers that specify the action they are handling. Methods that are being used as a default handler, with @handler(), are called for all types of circuits events, most of which do not relate to the Resilient Action Module. There is an alternate method to defer action handling in these types of handlers which is accessed by calling a defer method on the event itself.

Sample Usage:

```
@defer(delay=3)
@handler("my_action")
def _do_deferred_action1(self, event, source=None, headers=None, message=None):
    # Code to handle action here!
    return "action handled"

@handler()
def _do_deferred_action2(self, event, *args, **kwargs):
    """Defer handling action on generic handler"""
    if not isinstance(event, ActionMessage):
        # Some event we are not interested in
        return
```

```
    if event.defer(self, delay=3):
        return
    # Code to handle action here!
    return "action handled"
```

debounce

There are times when an action handler is likely to be triggered multiple times in quick succession, but you do not want to handle the events until all of them are done firing. The debounce method decorator allows you to “accumulate” these events and defer handling them until they stop firing. Similar to the defer decorator, a delay value is specified. If another event with the same key occurs within that delay period, then the timer is reset. All events are processed once the timer expires.

In most scenarios, it is only the last event in the series that is of interest. If the “discard” option is specified, then only the most recent event is handled when the timer expires and any earlier ones are discarded. This is useful in cases where all the events would have triggered the same action, resulting in “noise” on an incident’s newsfeed.

The defer decorator works only with handlers that specify the action they are handling. Methods that are being used as a default handler, with @handler(), cannot use this feature.

Sample usage:

```
@debounce(delay=30, discard=True)
@handler("task_changed")
def _who_owns_next_task(self, event, source=None, headers=None, message=None):
    inc_id = event.message["incident"]["id"]
    url = '/incidents/{0}/tasks?handle_format=names'.format(inc_id)
    tasks = self.rest_client().get(url)
    for _task in tasks:
        if _task['status'] == 'O':
            owner_fname = _task["owner_fname"] or ""
            owner_lname = _task["owner_lname"] or ""
            break
    else:
        owner = "All Tasks Complete"
    def update_func(inc):
        inc["properties"]["next_task_owned_by"] = "%s %s" % (owner_fname, owner_lname)
        return inc
    self.rest_client().get_put("/incidents/%d" % inc_id, update_func)
    yield "next_task_owned_by set"
```

Long-running actions

Some types of actions, like running a database query in another system, can take a long time to complete. A Resilient Circuits handler is blocking, meaning it can only handle one action at a time. To free the handler to take care of the next incoming event, you can use a circuits “worker” to run the lengthy task. A worker can be a separate thread or a separate process, depending on your needs.

The original action handler method is triggering a secondary task to do the real work of running the action and then returning (which acknowledges the event in the Resilient Action Module). This results in the Action Status in the Resilient platform showing up as “complete” even though the action is still being run.

Example:

```
def do_expensive_thing(incident_id):
    time.sleep(60)
    return "finished"

class expensive_thing(circuits.Event):
    pass

class MyComponent(ResilientComponent):

    def __init__(self, opts):
        super(MyComponent, self).__init__(opts)
        circuits.Worker(process=False, workers=5, channel=self.channel).register(self)

    @handler("expensive_thing")
    def _do_expensive_thing(self, inc_id):
```

```
        yield self.call(circuits.task(do_expensive_thing, inc_id))

    @handler("my_action")
    def _start_expensive_action(self, event, source=None, headers=None, message=None):
        """ Handler that kicks off long-running task """
        inc_id = event.message["incident"]["id"]
        self.fire(expensive_thing(inc_id))
        yield "Started expensive action"
```

Web UI and RESTful components

The Resilient Circuits framework comes with a built-in web framework and webserver to create your own REST API or Web UI.

Some applications, particularly ticketing systems, utilize webhooks as a means of integrating with other applications. These types of integrations work by allowing a user access to a URL that data is posted to when certain events occur, such as ticket creation and ticket update. A circuits based REST API is well suited to this use case.

Another use case for the circuits web framework is building a custom webform to facilitate incident creation by people who are not direct users of the Resilient platform. Refer to the circuits.web documentation for more information.

The first step in building a web component for Resilient Circuits is to install the rc-webserver package. From the same directory where you downloaded the package, run:

```
pip install rc-webserver --find-links .
```

The webserver requires a few configuration items in your app.config file, so next run:

```
resilient-circuits config -u
```

This adds the required configuration section with functional defaults, but you may wish to change them.

Your web component must inherit from the circuits class BaseController. If you need access to the Resilient REST API, you need to inherit from the ResilientComponent class. The “channel” your component listens on corresponds to the first path element from your URL. For example, if you set “self.channel=”/example”, then all requests starting with www.<hostname>:<port>/example are routed to your component for handling.

The ‘exposeWeb’ decorator is then applied to methods to handle routes more specifically. For example, putting “@exposeWeb(“test”)” above your method causes it to be called for all requests to www.<hostname>:<port>/example/test.

Package the integration

Once you have finished developing your component, you can package it so that it is installable and automatically discoverable by Resilient Circuits. Your project structure should look similar to the following:

```
my-circuits-project/
|-- setup.py
|-- README
|-- MANIFEST.in
|-- my_circuits_project/
|   |-- data/
|   |   |-- LICENSE
|   |   |-- sample_data.txt
|   |-- components/
|   |   |-- my_custom_component.py
|   |-- lib/
|   |   |-- helper_module1.py
|   |   |-- helper_module2.py
```

For an example of a setup.py file, see the Resilient community examples GitHub repository, choose an "rc-" integration and view its setup.py file. Your setup.py file should look similar. The name of each project always has an "rc-" prefix. That is for convenience so that they are readily identifiable as Resilient Circuits integrations, but is not required.

The "entry_points" section of setup.py makes your integration discoverable by Resilient Circuits as a component to run. The "resilient.circuits.components" key should be a set to a list of all component classes defined in your integration. The "resilient.circuits.configsection" key should point to a function in your integration package that returns a string containing a sample config section. This is called to generate data for a config file when a user runs "resilient-circuits config –u app.config".

Once your integration is packaged and tested, you can share it with other Resilient users, as described in "Publish your integration" on page 22.

Test the integration

Testing a Resilient Circuits component begins during development. Once you have a minimal component running, you can use the standalone res-action-tool to submit test action data to your component to quickly test changes to your logic. Support for running a suite of unit and/or integration tests using the Pytest framework is also provided.

res-action-test

The res-action-test tool is an interactive command-line tool for manually submitting actions to a component outside of a Resilient rule. The most common use case for this is to record real action data from a Resilient rule, and then "replay" it via the command line tool.

To record a session interacting with the Action Module, first make a directory to log the data. Then, run Resilient Circuits with the log-http-responses option.

```
mkdir logged_responses
resilient-circuits run -r --log-http-responses logged_responses/
```

Trigger the rule you want to record. Once you have seen the action received by the application, you can kill Resilient Circuits. In the logged_responses directory, you should see a filename that starts with "ActionMessage".

```
ls logged_responses/ActionMessage*
logged_responses/ActionMessage_AddTask_2017-03-07T09:24:41.822231
```

Run Resilient Circuits again with the test-actions option so that it listens for test actions to be submitted.

```
resilient-circuits run --test-actions
```

When Resilient Circuits is running, start the res-action-test tool in another shell. In the following example, the saved action message is submitted as if it came in from the "add_task" queue. The response that would have gone to the Resilient platform over the STOMP connection instead displays in the test tool.

```
res-action-test
Welcome to the Resilient Circuits Action Test Tool. Type help or ? to list commands.
(restest) submitfile add_task logged_responses/ActionMessage_AddTask_2017-03-07T09:24:41.822231
(restest)
Action Submitted<action 1>
(restest)
RESPONSE<action 1>: {"message": "action complete. task posted. ID 2253452", "message_type": 0,
 "complete": true}
2253452", "message_type": 0, "complete": true}
```

Because the res-action-test tool is a separate process running independently from the main Resilient Circuits application, it keeps running when the Resilient Circuits process is killed or otherwise terminated. You see a "disconnected" message appear. As soon as Resilient Circuits starts back up with the test-

actions option, it automatically reconnects. This makes it easy to submit a test action, make a change to your component and restart Resilient Circuits, and quickly re-run the test action.

For a complete list of actions available in rest-action-test, type “help”. For usage of any individual command, type “help <command>”.

Write and run tests using pytest

Once an integration is packaged as an installable component, you can create a suite of tests for your integration package. Several of our example components have tests written using the pytest framework. Learn about using pytest by reading the documentation here. IBM Resilient provides a plugin for pytest with several test fixtures that make writing Resilient Circuits tests easier.

You can download the pytest plugin from the Resilient GitHub repository and install it as follows:

```
pip install pytest_resilient_circuits-x.x.x.tar.gz
```

Resilient Pytest Fixtures

Once the plugin is installed, it makes several fixtures available in pytest. Each of these fixtures is “class-scoped”, so it is initialized once per class of tests. The following describes each fixture:

- **circuits_app**: Starts up Resilient Circuits with the specified appliance and credentials. The appliance location and credentials are pulled from the following environment variables if they are set. Otherwise, they must be provided as command line options when the test is run, as described in **Run Tests** below.
  - test_resilient_appliance
  - test_resilient_org
  - test_resilient_user
  - test_resilient_password
- **configure_resilient**: Clears out all existing configuration items from the organization and then automatically creates new ones as defined by your test class. Class members should be set as follows to describe required configuration elements. Any that are not necessary can be excluded.

```
destinations = ("<destination1 name>", "<destination2 name>", ...)
action_fields = {"<programmatic_name>": ("<number, text, etc...>",
        "<display_name>", None),
        "<programmatic_name>": ("select", "<display_name>",
        ("<option1>", "<option2>")), ...}
custom_fields = {"<programmatic_name>": ("<number, text, etc...>",
        "<display_name>", None),
        "<programmatic_name>": ("select", "<display_name>",
         ("<option1>", "<option2>")), ...}
automatic_actions = {"<display_name>": ("<destination name>",
        "<Incident, Artifact, Task, etc>", (condition1, condition2, etc)),
        "<display_name>": ("<destination name>", "<Incident, Artifact,
        Task, etc>", (condition1, condition2, etc))}
        *note that conditions are a dict in ConditionDTO format
manual_actions ={"<display_name>": ("<destination name>",
        "<Incident, Artifact, Task, etc>", ("<action field1>",
        "<action field2>", ...)),
        "<display_name>": ("<destination name>",
        "<Incident, Artifact, Task, etc>", ("<action field1>",
        "<action field2>", ...))}
```

- **new_incident**: Provides a python dictionary containing data suitable for doing a PUT against the /incidents endpoint in the Resilient platform. It has something valid populated for all required fields and simplifies creating test data in the Resilient platform.

Run Tests

All test modules should be in a “tests” directory at the top level of your package.

Assuming you have configured a “test” command in your setup.py, you should now be able to start your tests with the “setup.py test” command. This runs setup and your test suite in your current Python environment. Use of a Python virtual environment is recommended.

```
python setup.py test -a "--resilient_email <user email> --resilient_password <password> --
resilient_host <ip or hostname> --resilient_org '<org name>' tests"
```

If you have already installed your plugin, and thus do not need to run setup, you can kick off pytest directly with:

```
pytest –s --resilient_email <user email> --resilient_password <password> --resilient_host <ip or
 hostname> --resilient_org “<org name>” tests
```

Run tests with tox

Running with “setup.py test” runs your test suite in your current environment. Tox is a great way to test your package in a clean environment across all supported Python versions. It generates a new virtual environment for each supported Python version and runs setup and your tests. Read more about tox here and install it with:

```
pip install tox
```

To get started, create a tox.ini file in your package at the same level as the setup.py script. Set “envlist” to all the Python versions you want to support. Note that it can only run tests for those versions you actually have installed on your system. Because the “Resilient”, “resilient_circuits”, and “pytest_resilient_circuits” packages are all dependencies, make sure they are listed in the “deps” section. Copy those packages to a pkgs directory and set an environment variable so that pip can find them.

```
export PIP_FIND_LINKS=”/path/to/pkgs/"
```

Your package should look something like this:

```
my-circuits-project/
|-- setup.py
|-- tox.ini
|-- README
|-- MANIFEST.in
|-- my_circuits_project/
|   |-- data/
|   |   |-- LICENSE
|   |   |-- sample_data.txt
|   |-- components/
|   |   |-- my_custom_component.py
|   |-- lib/
|   |   |-- helper_module1.py
|   |   |-- helper_module2.py
|-- tests/
|   |-- tests_for_my_project.py
```

Now run your tests with:

```
tox -- --resilient_email <user email> --resilient_password <password> --resilient_host <ip or
 hostname> --resilient_org '<org name>' tests
```

Mock Resilient API

It is not always practical or possible to run tests against a live Resilient instance. The Resilient package includes a simple framework built on Requests-Mock to enable mocking a subset of the Resilient REST API. Only the endpoints used by your component need to be mocked. Some endpoints, like /session, always needs to be mocked because the Resilient helper module and Resilient Circuits packages use them.

Create a class derived from resilient.resilient_rest_mock.ResilientMock. Define a function for each endpoint you wish to mock, returning a requests.Response object. To register which endpoint you are

mocking, use the @resilient_endpoint decorator on the function, passing it the request type and a regex that matches the desired URL.

In this example, the /incident/<inc_id>/members endpoint is mocked for PUT and GET requests:

```
from requests_mock import create_response
from resilient.resilient_rest_mock import ResilientMock, resilient_endpoint
class MyMock(ResilientMock):

    def __init__(self, *args, **kwargs):
        super(MyResilientMock,self).__init__(*args, **kwargs)
        self.members = []

    @resilient_endpoint("GET", "/incident/[0-9]+/members$")
    def get_members(self, request):
        member_data = {"members": self.members, "vers": 22}
        return create_response(request, status_code=200, json=member_data)

    @resilient_endpoint("PUT", "/incident/[0-9]+/members$")
    def put_members(self, request):
        data = request.json()
        if "members" not in data or "vers" not in data or not isinstance(data.get("members"),
 list):
            error_data = {"success": False, "message": "Unable to process the supplied JSON."}
            return create_response(request, status_code=400, json=error_data)
        self.members = data["members"]
        member_data = {"members": self.members, "vers": 22}
        return create_response(request, status_code=200, json=member_data)
```

Override configuration values

In a development environment, you may find it necessary to override one or more values from your app.config file when running Resilient Circuits. For example, you may want to temporarily run with the log level set to DEBUG. To accomplish this, run Resilient Circuits with:

```
resilient-circuits run --loglevel DEBUG
```

You can also use optional parameters to run the application when the values being overridden are required and missing from the config file. For a complete list of optional arguments for overrides, run:

```
resilient-circuits run -- --help
```

Deploy your integration

You deploy your package to the Resilient platform using the following command:

```
resilient-circuits customize
```

Alternatively, you can install as specific package as follows:

```
resilient-circuits customize -l <package_name>
```

You are prompted deploy the components, such as functions, message destinations, workflows and rules. You can use the optional parameter, -y, in the command line to not be prompted.

Optionally, if your package has multiple functions, you can specify which functions to deploy as follows.

```
resilient-circuits customize -l <function_name1> <function_name2>
```

For details on deploying packages, see the *Integration Server Guide*.

Publish your integration

In addition to deploying your integration to other Resilient platforms in your environment, you can share your integration with the Resilient community.

IBM Resilient provides two locations for sharing functions, IBM Security X-Force App Exchange, and Resilient Community Apps on Github.

The IBM Security X-Force App Exchange allows you to make your integration available to others in the Resilient community. You have the option to update the integration as needed. For more information on submission requirements, see the Write Your Own page.

The Resilient Community Apps on Github also allows you to share your source code with others, who can then copy, modify and enhance your integration using the pull request mechanism. See the Resilient Community Apps page for a list of apps, with developer information at the bottom of this web page.

You can choose to submit to one or both locations.

Chapter 5. Using the API directly

The following sections describe the steps you need to consider if not using the Resilient Circuits framework.

You can write action processors in any language that allows TLS connections to a message broker using the STOMP or ActiveMQ (OpenWire) protocol.

If you use a Java-based language, typically you would use the ActiveMQ client library, which uses the OpenWire protocol. There are libraries that support STOMP and are available for most modern programming languages. The STOMP Clients web page includes many different STOMP client library options.

Before starting, you should be familiar with the Resilient API. To access the API Reference guide, including schemas for all of the JSON sent and received by the API, log into the Resilient platform, click your account name at the top right and select **Help/Contact**. For additional information, see Chapter 6, "JSON structures in the Resilient API," on page 35.

Prerequisites

Before starting, make sure your environment meets the following prerequisites:

- Resilient platform V31 or later with the Resilient Action Module.

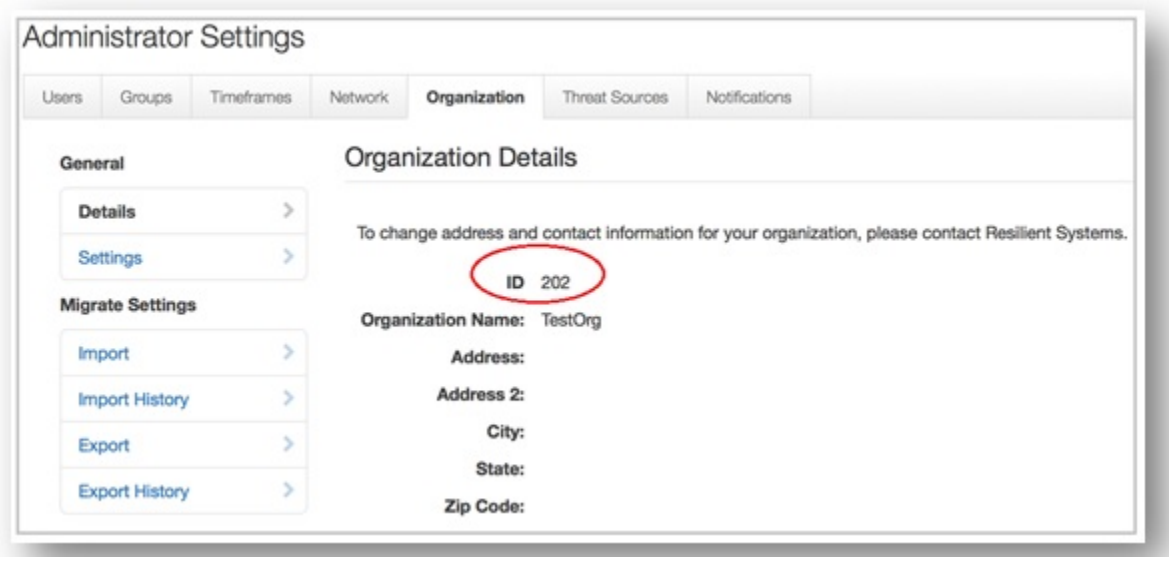
  IBM Resilient recommends that you use a Resilient platform in a test environment to create the function, message destination, rules, workflows and other components needed for your integration. Once tested, you can deploy the integration into any Resilient platform that is at the same or later version as your test platform.
- Dedicated Resilient account to use as the API user. The Resilient system administrator provides the account and its credentials. The administrator provides one of the following accounts.
  - User account with user name (in the form of an email address) and password. With most extensions, the account must have the permission to view and edit incidents, and view and modify administrator and customization settings. If the Resilient organization has Two Factor Authentication enabled, the account is configured as excluded.
  - API key account with ID and secret, only if using V33 or later of the Resilient platform and Resilient Circuits. This is a more secure account. With most extensions, the account must have the permission to view and edit incidents, and view and modify administrator and customization settings. The API key account is not available with the Resilient MSSP add-on deployment.
- You have created the components you need, as described in Chapter 3, "Creating Resilient components," on page 7.

Some action processors need to use the Resilient REST API to access or modify additional Resilient data. This same user account can be used to authenticate with the REST API.

NOTE: When making API calls, you need the JSESSIONID and csrf_token to create a session, as shown in "Example: creating an incident" on page 32.

Message Destination and Org prefix

On the Resilient platform, message destinations have a display name and a generated programmatic name. When connecting to message destinations from your code, use the programmatic name and include the organization ID as a prefix. The following figure illustrates how to locate your organization ID in the Resilient platform.



If you created a message destination in the Resilient platform with a programmatic name of “ticket” and your organization ID is 202 as it is in the previous figure, the name you would use in your action processor code to read messages would be “actions.202.ticket”.

Some client libraries have you connect to the destination using a “/queue” or “/topic” prefix. For example, if you were connecting to the ticket queue, you would use a name of “/queue/actions.202.ticket”. Consult the documentation for your client library for more information.

Action data

Messages contain JSON data. The structure of the JSON data is described in the Resilient REST API documentation in the ActionDataDTO type. This structure contains much of the data that you need to implement with your action processors. However, if there is additional Resilient data that you require, you can access it using the Resilient REST API. See “Resilient REST API with action processors” on page 28 for considerations when doing this.

The following table describes the top-level properties in the ActionDataDTO type. For specific information about this type, consult the Resilient REST API documentation.

Field Name	Description
action_id	ID of the rule that caused the message.
type_id	Type of object that caused the message. The types and their associated IDs are available through the REST API with the “/rest/orgs/{orgId}/types” endpoint.
incident	Incident object to which the invocation applies. Note that this value is set for items that are subordinate to incidents. It is currently the case that all messages contain an incident.
task	Task to which the invocation applies (if any). Note that this value is set for items that are subordinate to tasks, such as task notes and task attachments.
artifact	Artifact to which the invocation applies (if any).
note	Note to which the invocation applies (if any).
milestone	Milestone to which the invocation applies (if any).
attachment	Attachment to which the invocation applies (if any).

Field Name	Description
type_info	Contains information about types/fields that are referenced by the other data. See “Action data and type information” on page 25 for more information.
properties	Contains the field values the user selected when invoking a menu item rule (if any).
user	Contains information about the user that invoked the action.

Action data and type information

The data specified in the incident, task, artifact, note, milestone and attachment fields generally contains only the ID values of objects they reference. For example, the incident “severity_code” field is a select list. The “incident.severity_code” value specified in the message data contains an integer (the severity ID). If your processor needs the severity text that was actually selected, you can get it from the type_info field.

```
# Python example of retrieving severity text from type_info

# Convert message text into a dictionary object
json_obj = json.loads(message)

# Get severity_code from the incident
sev_id = json_obj['incident']['severity_code']

# Use type_info to get the severity's text value
sev_field = json_obj['type_info'] \
  ['incident'] \
  ['fields'] \
  ['severity_code']

text = sev_field['values'][str(sev_id)]['label']

print "Severity text is %s" % text
```

Message headers

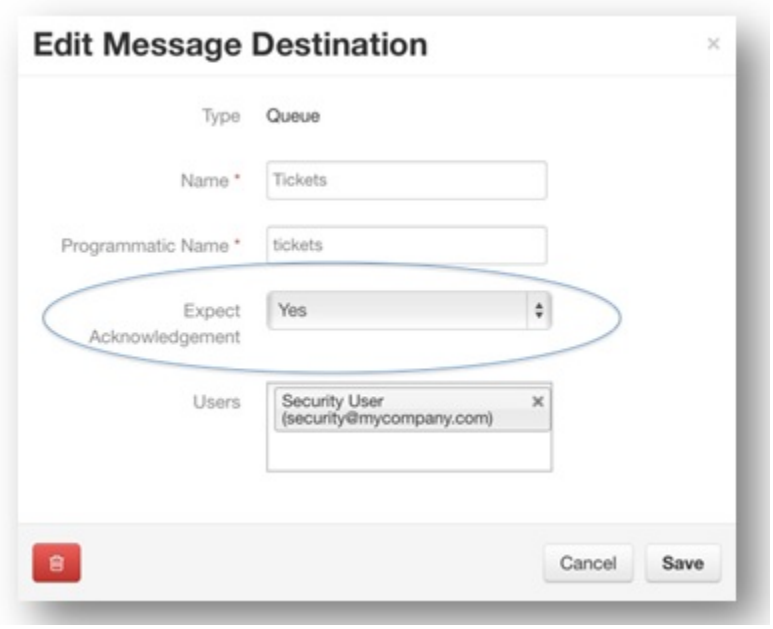
The Resilient platform includes various message headers that are needed (or in some cases just helpful) in processing messages.

Header Name	Request/Reply	Description
Co3ContextToken	Request	A token value that must be specified if the action processor calls back into the Resilient REST API. The primary purpose of this token is to ensure that actions processing does not result in an infinite loop. See “Resilient REST API with action processors” on page 28 for additional information.

Header Name	Request/Reply	Description
correlation-id	Request and Reply	Identifies the rule invocation to which this message applies. It must be included in acknowledgement messages sent back to the Resilient platform. See “Acknowledgements” on page 26 for additional information. If you are using a JMS client, this value can be retrieved with the getJMSCorrelationID method.
reply-to	Request	Identifies a server-controlled message queue that must be used when acknowledging (replying to) this message. See “Acknowledgements” on page 26 for additional information. If using a JMS client, this value can be retrieved with the getJMSReplyTo method.
Co3InvocationComplete	Reply	A boolean header that tells the Resilient platform whether processing is complete. The default is true, so you need to include it only if you are sending an informational message and it is not complete. This header is ignored if the reply message is JSON.

Acknowledgements

Some action processors consume messages and silently process them without returning any indication of progress or status to the Resilient platform (“fire and forget”). Other action processors return an acknowledgement when they have completed the processing of a message (“request/response”). The Resilient platform supports either mode of operation through the Expect Acknowledgement setting of the message destination, as shown in the following figure.



When a message destination is configured with an Expect Acknowledgement value of Yes, the list of executed actions in the Resilient UI shows messages/invocations as Pending until the expected acknowledgement is received. If an acknowledgement is not received within 24 hours, the Resilient platform displays it as an error. Users can see the list of actions invoked on an incident by selecting the **Actions > Action Status** option from the incident view.

If the message destination is configured with an Expect Acknowledgement value of No, the action immediately displays with a status of Completed.

The following is a partial example of how to explicitly send a reply using the stomp.py Python library:

```
# Simple reply using Python
class MyListener(object):
  def __init__(self, conn):
    self.conn = conn

  def on_message(self, headers, message):
    reply_headers = {'correlation-id': headers['correlation_id']}
    reply_to = headers['reply-to']
    reply_msg = "Processing complete"

    conn.send(reply_to, reply_msg, reply_headers)
```

The Resilient platform accepts either JSON or just a simple text string for reply messages. Simple plain text reply messages are a way to provide a success acknowledgement with minimal effort. You can also use a more descriptive JSON string value, which is parsed by the server.

The format for the JSON messages is included in the Resilient REST API documentation (see the ActionAcknowledgementDTO type). For convenience, the following table illustrates sample values for error and informational reply messages.

Reply Type	**Example JSON**
Error	{"message_type": 1, "message": "Some error occurred ...", "complete": true}
Information	{"message_type": 0, "message": "Started processing", "complete": false}

Completed	{"message_type": 0, "message": "Completed processing", "complete": true}

Processors can send reply messages, even if they are not expected. This allows informational or error messages to be returned even if no reply is expected. You may choose to utilize this behavior if you expect that the processors will rarely fail. Unexpected replies are displayed in the Action Status screen just as they are for expected ones.

TLS

Action processors must connect to the Resilient message broker using TLS v1.1 or higher. This applies to both the connection to the message destination (STOMP over TLS and Active MQ/OpenWire over TLS) and the Resilient REST API (HTTPS).

To ensure the security of the connection, action processors must properly validate the server certificate. The exact mechanisms for doing this varies by programming environment and is beyond the scope of this document. However, the following must be considered:

- Is the certificate chain presented by the server *trusted*?
- Is the certificate signature correct?
- Has the certificate *expired*?
- Was the certificate issued to the site to which the connection was made? That is, does the certificate's common name or subjectAltName match the connected server's name?

Some of the common JMS libraries for Java do not perform checking on the certificate name (last bullet above). There is a workaround for this, which is used in the Java examples.

Resilient REST API with action processors

Resilient action processors can make use of the Resilient REST API to update incidents, retrieve additional information not included in the rule message data, etc.

If using a Resilient email/password account to authenticate, you must specify the X-Co3ContextToken HTTP header when making requests. It is not required if using an API key account to authenticate.

The value to specify in this header is passed as the Co3ContextToken message header. This ensures that any modifications done through the API do not cause an infinite loop of message invocations. For example, if an incident rule has no conditions specified then it triggers every time the incident is saved. If the downstream action processor itself saves the incident, then you might end up in a never-ending loop. The X-Co3ContextToken HTTP header tells the server to skip the rule that generated the original message.

The following code is using the SimpleClient class that is included with the examples. SimpleClient provides post, put and delete methods that take the token as an argument. See the example processor code for additional details.

```
# Use Resilient REST API from Python processor
class MyListener(object):
  def __init__(self, conn):
    self.conn = conn
    self.client = co3.SimpleClient(...)

  def on_message(self, headers, message):
    # Get the token from the message header, set into client object
    self.client.context_header = headers['Co3ContextToken']
    message_obj = json.loads(message)

    inc_id = message_obj['incident']['id']
    url = "/incidents/{}/comments".format(inc_id)
```

```
    comment_data = {'text': 'Some comment for the incident'}

    # Create the comment
    self.client.post(url, comment_data)
```

HTTP Conflict (409) errors

It is possible for the Resilient platform to return an HTTP Conflict (409) status when updating (performing an HTTP PUT on) incidents using the REST API. This status code indicates that the incident you are modifying has changed since you last read it. Your processor must be written to handle this situation, generally by re-reading the incident object (using an HTTP GET), re-applying your changes and re-issuing the PUT.

The Resilient examples have accounted for this issue where necessary.

Using a framework

There are frameworks that may simplify the development of Resilient action processors. You may want to investigate tools that may simplify the creation of action processors, which generally follows typical "Enterprise Integration Patterns". See the following table for ESB and ESB-like frameworks worthy of investigation.

NOTE: See http://www.enterpriseintegrationpatterns.com for more information about Enterprise Integration patterns.

IBM Resilient provides a set of action processor components for Python, built with the Circuits framework. Refer to IBM Resilient Python API for a list of Python library modules.

Product	Language	Description
Apache Camel **http://camel.apache.org/**	Java	An open source framework for creating processing routes. For example, a route might: • Read a message from a queue (Resilient message destination). • Convert the message payload from a JSON string to an object. • Invoke an HTTP POST on some external service. • Send a reply to the Resilient platform. Most of this can be done through XML- or DSL-based configurations. There is an example of how you can use Apache Camel in the Resilient API examples distribution.
Apache ServiceMix ESB **http://servicemix.apache.org/**	Java	ServiceMix is an OSGi-based Enterprise Service Bus (ESB). ServiceMix can work seamlessly with Apache Camel to simplify route creation.

Mulesoft ESB **http://www.mulesoft.com**	Java	A commercial Enterprise Service Bus (ESB) that allows you to create graphically action processors using a number of built-in connectors. There is an example of how you can use Mulesoft in the Resilient API examples distribution.
Spring Integration **http://projects.spring.io/spring-integration/**	Java	Extends the Spring programming model to support Enterprise Integration Patterns.
Zato ESB **https://zato.io**	Python	An open source Python-based Enterprise Service Bus (ESB).

Always running

It is easy to write an action processor script that uses the Action Module to read rule messages and perform an operation. When you are developing the script, you can run it from the command line. However, when you exit the shell or log out of your desktop session, the program exits.

You should consider in advance how you are going to ensure that the program remains running when the action processor is deployed in the production environment.

If using Python on Unix, consider using the systemd daemon.

If using Java, consider using the Apache Commons Daemon project.

Retry

You should consider how your action processor handles situations where external systems (including the Resilient platform itself) are inaccessible.

It is generally desirable for action processors to retry their connection to the message destination indefinitely. Indefinitely retrying to reconnect every 30-60 seconds is reasonable.

If other downstream operations fail, you need to decide how to proceed. It may be sufficient to simply fail the operation and send a response message to the Resilient platform indicating the failure, where these messages appear in the Action Status page.

This is one area where an integration framework can help. They generally have built-in support for error handling. For an example, see the Resilient Action Module Apache Camel example in the Resilient API distribution.

Processor installation

Action processors are frequently written to assume the existence of certain message destinations and rules. You can create these dependencies using one of the following methods:

- Create them manually using the Resilient platform.
- Write a program that uses the Resilient REST API to create them.

Testing considerations

Many of the design considerations discussed in the previous section lead to useful test cases. For example, the discussion on "Retry" on page 30 leads a tester to a number of test cases dealing with how the action processor handles situations where other systems are not running or return errors.

The following table contains test cases that serve as a starting point for testing an action processor.

Test	Description
TLS: Certificate Trust	When you invoke the action processor, you must confirm that a "man-in-the-middle certificate attack" causes an error and that no data is sent over the connection. Otherwise, it would be possible for bad code to establish a connection, send passwords *then* check for certificate trust. Sending the password over an untrusted channel would be a security vulnerability. The simplest way to test this is to configure the client (action processor in this case) to NOT trust the Resilient platform certificate and confirm that the operation fails due to a TLS error.
TLS: Certificate Common Name	If an action processor thinks it is connecting to a host named "Resilient.mycompany.com" then it is important that the TLS certificate is issued to "Resilient.mycompany.com". If not and you proceed sending data, it is possible for a man-in-the-middle to present a certificate that was issued by a trusted source, but issued to a different entity (such as www.someothercompany.com). The accepted best practice for a TLS client to guard against this attack is to check that the certificate's common name (or subjectAltName) matches the host to which the connection is being made. The simplest way to test is to change your local hosts file to make "testhost" point to the Resilient platform's IP address, and then try to connect using testhost. The connection should fail. Note that this should be performed against both the Resilient REST API (port 443) and the Action Module server (port 65000 and/or 65001).
Retry/Error Reporting	It is important for the action processor to continue operating in the face of exceptions. The following should be tested: • Does the action processor have a log file? • Does the action processor report errors from external systems? • Does the action processor continue running when the Resilient platform is down?

Test	Description
Always Running	The action processor should generally be running. • Does the action processor start automatically when the host on which it runs is restart? • Does the action processor process survive a user log out?
Conflicting Edits	If the action processor updates the Resilient platform using the REST API, it must be written to handle situations where another user edits the same object. • Has this situation been accounted for by the developer? • If you invoke a rule when the action processor is stopped, then make another change to the object (say an incident), then start the action processor. Does the action processor properly update the incident? Note: If it is not handled by the developer, then you would likely get an error when the processor attempts to do the PUT operation.
Action Status Sent	Does the action processor send a status or error message to the Resilient platform as appropriate? These status messages appear in the Actions > Action Status dialog.
Infinite Loops	Is it possible for the action processor to get into an infinite loop? See the Co3ContextToken discussion in “Message headers” on page 25.

Example: authentication script

The following script is an example of using an API key account to authenticate to the Resilient platform.

```
#!/bin/bash
api_key_id=“<your_api_key>“
apikey_secret=“<your_api_key_secret>”
server_url = "http://localhost:8080/rest/orgs/<org_id>/incidents
curl -kv -u "$api_key_id:$apikey_secret" $server_url
```

Example: creating an incident

The following is an example of how to create an incident using the API. The example uses the RESTclient extension for Firefox which provides a curl alternative. It authenticates using a Resilient email/password account.

```
$ curl -X POST -k -H 'Content-Type: application/json' -i 'https://resilient.example.com/rest/
session' --data '{"email" : "resilient.admin@example.com","password":"*****"}' -v
```

In the response, you want the JSESSIONID and csrf_token.

```
< Set-Cookie: JSESSIONID=22EDC0CB8A2ECDE8C15A92C05ABC1F90; Path=/; Secure; HttpOnly
Set-Cookie: JSESSIONID=22EDC0CB8A2ECDE8C15A92C05ABC1F90; Path=/; Secure; HttpOnly
```

```
{"orgs":
[{"id":201,"name":"Collaborationben","addr":null,"addr2":null,"city":null,"state":null,"zip":null,"perms":
{"administrator":false,"observer":false,"master_administrator":true,"create_incs":true,"create_shared_layout"
[49,50,51,52,53,85,54,55,56,58,59,60,61,62,63,101,102,103,104,105],"role_handles":
[1001],"twofactor_cookie_lifetime_secs":0},
{"id":202,"name":"NYC","addr":null,"addr2":null,"city":null,"state":null,"zip":null,"perms":
{"administrator":false,"observer":false,"master_administrator":true,"create_incs":true,"create_shared_layout"
[49,50,51,52,85,53,54,55,56,58,59,60,61,62,63,101,102,103,104,105],"role_handles":
[1013],"twofactor_cookie_lifetime_secs":0}],"user_id":15,"user_fname":"Resilient","user_lname":"Admin","user_
```

Both the JSESSIONID and csrf_token need to be added as cookies to the next request, shown below. You may not need to add the csrf_token to the cookies.

```
$ curl 'https://resilient.example.com/rest/orgs/201/incidents' -H 'Cookie:
 CSRF_TOKEN=be8bc69380b686782b441e9790eef812; JSESSIONID=22EDC0CB8A2ECDE8C15A92C05ABC1F90'
 -H 'Content-Type: application/json' -H 'Accept-Language: en-US,en;q=0.9' -H 'Accept: */*'
 -H 'text_content_output_format: objects_convert' -H 'X-Requested-With: XMLHttpRequest' -
H 'X-sess-id: be8bc69380b686782b441e9790eef812' -H 'Connection: keep-alive' --data-binary
 '{"name":"test","discovered_date":1524236957000,"due_date":null}' -k -v
```

If you do not add discovered_date, it fails as follows.

```
{"success":false,"title":null,"message":"Field discovered date is required.","hints":
["field_defs"],"error_code":"generic"}
```

Chapter 6. JSON structures in the Resilient API

JavaScript Object Notation (JSON) is the native format for messages in the Resilient REST API and in the Actions Module. The following sections provide an outline of the JSON structures in the Resilient platform, including incidents, tasks, and other objects, without going into specific details of the programming involved.

The Resilient REST API guide includes schemas for all of the JSON sent and received by the API. The guide is accessible from the Resilient platform **Help/Contact** page.

Basics

An incident is represented as a JSON document, with its various properties (fields), such as the following example:

```
{
    "discovered_date": 1434029747498,
    "name": "Phishing emails"
}
```

The order of the fields in a JSON document is not important. However, some values are **lists**, and the order of items in a list is important.

Formatting and whitespace between the fields is not important. Quotation marks can be single-quote or double-quote, but must not be the “curly quotes” that word-processors like to use.

When you receive incident data from the Resilient platform, it is in the form of a JSON document with all the incident’s properties. Some of these properties are for internal use and have no meaning in your application; however, you should retain these properties when sending an updated JSON document back to the platform.

When you create a new incident, you need only to specify the fields that are required. In a standard installation without any field customizations, the only two required fields are **name** and **discovered_date**.

In the REST API documentation, these JSON documents are referred to as Data Transfer Objects (DTO) elements. The Java API includes a set of DTO classes that represent the same data structures. There are a number of these DTOs. An incident might be represented as an `incidentDTO`, or as a `fullIncidentDataDTO` (which includes more fields), or as a `partialIncidentDTO` (which includes fewer fields), depending on the context.

Data types

The basic data types include text, numbers, dates and times, lists, and more complex values.

Data Type	Description
Text	There are two types of text field: plain text, such as the incident name; and text areas. Text areas can be multi-line and also have rich-text values, as described in “Rich text” on page 36. JSON text values are quoted. To include a quote character within JSON text, it must be escaped with a backslash. To include a literal backslash, it too must be escaped with a backslash.

Data Type	Description
Numbers	Numeric fields in the Resilient platform are integers. They do not support fractional values. Fields such as dates and times, and object references are not numeric fields.
Date and Times	All dates and times in the Resilient platform are represented with a numeric value. This encodes the number of milliseconds since January 1st 1970, UTC. Depending on how you access the REST APIs, you often need to convert these into a different representation for display, storage or processing. For example, the value 1439993716000 might be represented as “2015-08-19 14:15:16 UTC”, “08/19/15 09:15:16 -0400” or “Wednesday”.
Boolean	Boolean values are either **true** or **false**.
Null	Null values are **null**.
Lists	A list is a sequence of values. JSON lists are defined with square brackets. A list of numbers would be represented as [1,2,3]. A list of strings would be represented as ["one", "two", "three"].

Other values

Other types of value include object handles (references), which refer to a value that is defined elsewhere; and structured values, where the value has several components.

Rich text

Rich text can include a subset of HTML to describe the text formatting. A rich-text value includes HTML tags such as and .

If you update incident or other data (round-trip), it is best to keep text-area fields in their original format to avoid accidental updates that might cause loss of formatting or unnecessarily notify users of updates.

Object handles

You will often encounter object handles in the Resilient REST API and Actions Module. The handle is a reference to an object that is defined elsewhere and can be referenced by ID. This allows the text of the object to change without having to go back and update every occurrence in the system. In database terms, object handles correspond to foreign keys.

For example, the resolution_id field has several valid values: “Unresolved”, “Duplicate”, “Not an Issue” and “Resolved”. Each value has an ID, for example: 53, 54, 55, 56.

Internally, the incident stores a reference to the value using its ID. This value may appear in the incident JSON as the ID,

```
{ "resolution_id": 53 }
```

or as the string value,

```
{ "resolution_id": "Unresolved" }
```

or as the full object:

```
{ "resolution_id": {"id": 55, "name": "Unresolved"}}
```

Data is returned from the server with the ID values by default. If you wish to receive name values back for object handle fields from the server instead, you can set `handle_format=names` either in an HTTP header or a query string; for example:

```
https://example.mycompany.com/rest/orgs/:orgId/incidents?handle_format=names
```

For a description of the possible values of the handle_format query string parameter/HTTP header, refer to **objectHandleFormat** in the REST API documentation.

When setting object handle values, the server works with either format. Numeric values (not quoted) are interpreted as IDs, and string values (surrounded by quotes) are resolved to the underlying IDs by the server. You can also specify the format by setting `handle_format=ids` in the query string or HTTP header.

Structured values and custom fields

Some values have multiple parts. For example, the full incident DTO produced by the server includes information about the creator, which is represented as a field with a structured value containing the creator's name, id, and so on.

Custom fields in an incident are represented in a similar way, as values within a group "properties".

For example, if you have defined custom fields with API names "source_types" (a multi-select field), "external_case_id" (a text field) and "cmdb_info" (a hidden text field), the incident JSON might show:

```
{
   "id": 2269,
   "properties": {
        "source_types": [],
        "external_case_id": "INC00020478",
        "cmdb_info": null
    },
    "phase_id": 1005
    /* etc */
}
```