IBM Resilient

Incident Response Platform

FUNCTION DEVELOPER'S GUIDE v1.0

Resilient Incident Response Platform Function Developer's Guide

Platform Version	Publication	Notes
1.0	June 2018	Initial release.

Table of Contents

1. Objective

This guide provides the information to integrate the Resilient Incident Response 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 integrations 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.

2. Overview

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

2.1. Resilient Incident Response Platform

The Resilient Incident Response 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 an integration. 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.
- 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.

- Function. A Resilient object that sends data to a remote function processor through a message destination. The *function processor* is the remote code component that performs an activity and returns the results. The results are acted upon by scripts, rules, and workflow decision points to dynamically orchestrate the security incident response activities. Functions simplify development of integrations by wrapping each activity into an individual workflow component. A function consists of the following components:

    - Inputs. Data that is acted on by the function processor. The inputs can be provided by a Resilient user or by a pre-process script.

    - Pre-process script. A script that is used to dynamically set the value of one or more of the function's input fields. You can use the script to retrieve the current value of a property then provide that value to the function as an input. A pre-process script cannot perform write activities on objects, such as changing incident values or adding artifacts.

    - Output. Result of the function processor. A post-process script can act upon this result. If saved, objects within the same workflow instance and executed after the function can also access the data.

    - Post-process script. A script that performs an activity in response to the result provided by the function. The script can change incident values, add artifacts, add data table rows, and more.

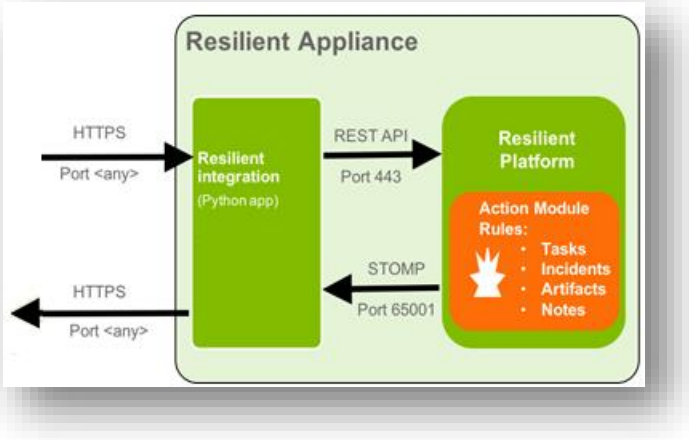
For more information about the Resilient platform and dynamic playbooks, refer to the *Resilient Incident Response Platform Playbook Designer Guide*.

2.2. 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 access the API Reference guide, including schemas for all of the JSON sent and received by the API, log into the Resilient platform, click on your account name at the top right and select **Help/Contact**.

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.



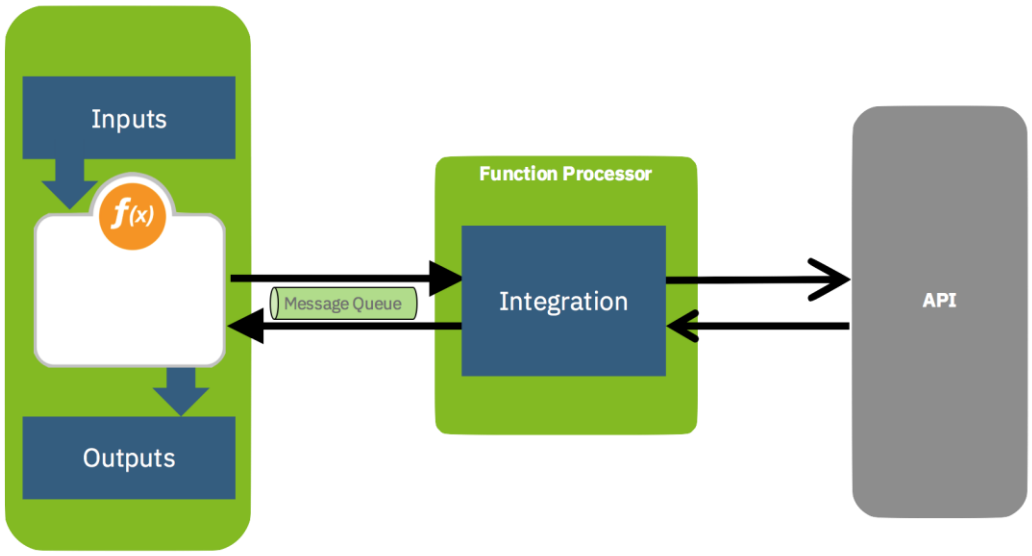
The Resilient platform contains an interactive Rest API browser that allows you to access the Resilient REST API and try out any endpoint on the system. When logged into the Resilient platform, click on your account name at the top right and select **Help/Contact**. Here you can access the complete API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

2.3. Function Processor

Resilient functions send data to external programs called function processors. The function processor can then perform integration work, for example:

- Performing a lookup, such as for information about a user or machine in an asset database, and returning the data values found.
- Searching SIEM logs, such as for an IP address, a URL or a server name, and returning a list of events.
- Sending a file attachment to a sandbox for analysis, producing a report and a collection of observables.
- Opening a ticket in an ITSM system with a type, name and description, and returning the ticket-ID.
- Triggering an external action and then returning results for use in workflows, tasks and other decisions.

In the Resilient platform, a rule is configured on an incident, an artifact or other object. When the rule fires, it runs a workflow that could have multiple steps. Those steps can include functions that send input parameters to the function processor using a message destination, receive the results, and use the results to update the Resilient incident, to decide the direction of subsequent workflow steps or in a variety of other ways.



When a rule fires in the Resilient platform, it executes a workflow that might have multiple steps. Those steps can include functions that send **input parameters** to the function processor using a message destination, receive the result, and use the result to update the workflow, to decide the direction of subsequent workflow steps, or in a variety of other ways.

The Resilient Circuits framework is required and makes it extremely simple to develop and deploy functions using Python. The function processor component is a Python class that implements *function methods*. These functions are called by the framework when the Resilient platform invokes the function from a workflow.

2.4. 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.

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 config file. This file stores information about the Resilient platform, such as user credentials, as well as variables for your functions.

Resilient Circuits use the Resilient helper module, which is a Python library to facilitate easy use of the Rest API.

2.5. Development Overview

Before you write function, you must understand the purpose of the function, the inputs the function needs to perform its activity, the expected results, and the actions or decisions to be made based on the results.

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

1. Install and configure Resilient Circuits and the Resilient helper module on your integration system. You do not need to repeat this step if you have previously installed Resilient Circuits.
2. Log into the Resilient platform and create one or more functions and associated components, such as message destination and workflow.
3. Export the Resilient platform settings to an export file.
4. Using Resilient Circuits and codegen, write the function processor, which is the code for the integration.

   TIP: If you have access to functions that are similar to the one you wish to create, use that function as a template to save time.
5. Use the pip install command to deploy the function to the Resilient platform and test the function by triggering the workflow and checking the results.
6. If you make any changes to any of the function's elements, export the Resilient platform settings and repackage your function processor using codegen.
7. When satisfied with the test results, compress the files into a tar.gz file then distribute that file to other Resilient administrators. You can deploy the package to any Resilient platform at the same version as your test platform.

2.6. Developer Website

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.

- 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.
- Releases. Lists the apps by Resilient Incident Response Platform release. You can also download from this page.

3. Prerequisites

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

- Resilient platform V30 (preferably in a test environment) with the Resilient Action Module.
- A dedicated Resilient account to use as the API user. In most integrations, the account must have the permission to view and edit incidents, and view and modify administrator and customization settings. You need to know the account username and password.
- Resilient Circuits and Resilient helper module V30. You should be familiar with the Resilient Circuits framework.
- A text editor.
- A basic knowledge of Python.
- If using a system for integration other than the one used by the Resilient platform, the system must have access to the Resilient platform and have one of the following configurations:
    - Red Hat Enterprise Linux 7.4 with Python 2.7 or later, or Python 3.4 or later.
    - Windows with Python 2.7 or later, or Python 3.4 or later.

IBM Resilient recommends that you use a Resilient platform in a test environment to create the Resilient elements then later on to test the function. Once tested, you can deploy the function into any Resilient platform that is at the same version as your test platform.

4. Configuring Resilient Circuits

Typically, you install the Resilient helper module and Resilient Circuits framework on the same system as your Resilient platform; however, you can install and manage your integration from a different system. Using a different system is useful if you have multiple Resilient integration packages in your environment.

4.1. Install Resilient Circuits

The installation procedures assume that all the packages are to be installed on the same system as the Resilient platform and that you have downloaded the Resilient Circuits and Resilient helper module packages from IBM Resilient Github.

Install the Resilient helper module and Resilient Circuits framework as follows:

1. Use ssh to connect to your Resilient appliance.
2. Go to the folder where the installers are located.
3. Update your pip version using this command:

   ```
   sudo pip install --upgrade pip
   ```

4. Update your setup tools using this command:

   ```
   sudo pip install --upgrade setuptools
   ```

5. Log out of your system and log in with a new session to ensure your environment is correct.
6. Install the Resilient helper module using the instructions in the readme file.
7. Install the Resilient Circuits package using the instructions in the readme file.

You should see a “successfully installed” message for Resilient helper module and Resilient Circuits.

4.2. Install Integrations

Once you install the Resilient Circuits and helper module, you can optionally install components, such as functions or other types of integration packages. Some are available for download from GitHub and you can write your own as well. Use the following procedure to install and configure a component.

1. Use ssh to connect to your Resilient appliance.
2. Go to the folder where the installers are located.
3. Install your chosen component using the following command:

   ```
   pip install <package_name>-x.x.x.tar.gz
   ```

4. Verify that the component installed using the resilient-circuits list command.

   ```
   resilient-circuits list
   ```

5. Follow the instructions in the component’s readme file to configure the component.

4.3. Create the Configuration File and Log Directory

The Resilient Circuits package requires a configuration file and logging directory.

The configuration file defines essential configuration settings for all resilient-circuits components running on the system. If you have multiple Resilient integration packages, they use the same configuration file.

Other integration components may have additional requirements.

Note to Windows Users: To run integration commands on a Windows system, use resilient-circuits.exe. For example, “resilient-circuits.exe run” rather than “resilient-circuits run”.

The examples in this guide use the default name, app.config, as the name of the Resilient Circuits configuration file.

Perform the following to create the configuration file:

1. Create a directory on the system where Resilient Circuits can write log files.
2. Use one of the following commands to generate a base configuration file.

   - Option 1: Create a directory ‘.resilient’ in your home directory with a file in it called app.config, which is the default and preferred option.

     ```
     resilient-circuits config -c
     ```

   - Option 2: Sometimes it is necessary to create a configuration file in a different location or give it a different name. You need to store the full path to the environment variable, APP_CONFIG_FILE.

     ```
     resilient-circuits config -c /path/to/<filename>.config
     ```

If APP_CONFIG_FILE is not set, then the application looks for a file called “app.config” in the local directory where the run command is launched from. This can be useful during development of a new component.

4.4. Edit the Configuration File

The [resilient] section of the configuration file controls how the core resilient_circuits and Resilient packages access the Resilient platform.

Open the configuration file in the text-editor of your choice and update the [resilient] section with your Resilient appliance hostname/IP and credentials and the absolute path to the logs directory you created. The following table describes all the required and optional values that can be included in this section.

NOTE: If on a Windows system and you edit the 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.

Parameter	Required?	Description
logfile	N	Name of rotating logfile that is written to logdir. Default is app.log.
logdir	N	Path to directory to write log files. If not specified, program checks environment variable DEFAULT_LOG_DIR for path. If that is not set, then defaults to a directory called “log” located wherever Resilient Circuits is launched.
log_level	N	Level of log messages written to stdout and the logfile. Levels are: CRITICAL, ERROR, WARN, INFO (default), and DEBUG.
host	Y	IP or hostname for the Resilient platform.

Parameter	Required?	Description
org	Y, if multiple orgs	Name of the Resilient organization. This is required only if the user account is used with more than one Resilient organization.
email	Y	User account for authenticating to the Resilient platform. It is recommended that this account is dedicated to integrations.
password	Y	Password for the Resilient user account.
no_prompt_password	N	If set to False (default) and the "password" value is missing from this config file, the user is prompted for a password. If set to True, the user is not prompted.
stomp_port	N	Port number for STOMP. Default is 65001.
componentsdir	N	Path to directory containing additional Python modules. Resilient Circuits load the components from this directory.
noload	N	Comma-separated list of: a. Installed components that should not be loaded. b. Module names in the componentsdir that should not be loaded. Example: my_module, my_other_module, InstalledComponentX
proxy_host	N	IP or Host for Proxy to use for STOMP connection. By default, no proxy is used.
proxy_port	N	Port number for Proxy to use for STOMP connection. By default, no proxy is used.
proxy_user	N	Username for authentication to Proxy to use for STOMP connection. If a proxy_host is specified and no proxy_user specified, then assumed no authentication is required.
proxy_password	N	Password for authentication to Proxy to use for STOMP connection. Used in conjunction with proxy_user.
cafile	N	Path and file name of the PEM file to use as the list of trusted Certificate Authorities for SSL verification when the Resilient platform is using untrusted self-signed certificates. If there is a PEM file, use a second instance of **cafile** to set to True or False. If set to False, certificate verification is not performed and the PEM file is used. If set to True (default), allow only trusted certs.

Whenever you install a new component package for Resilient Circuits, you need to update your app.config file to include any required section(s) for the new component(s). After installing the package, run:

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

If using an alternate file location for your app.config file, you need to specify it when you update.

```
resilient-circuits config -u /path/to/app.config
```

This adds a new section to your existing config file with default values. Depending on the requirements of the component, you may need to modify those defaults to fit your environment, such as credentials to a 3rd party system.

4.5. Pull Configuration Values

Values in the config file can be pulled from a compatible keystore system on your OS. This is useful for values like password that you would prefer not to store in plain text. To retrieve a value from a keystore, set it to ^<*key*>. For example:

```
[resilient]
password=^resilient_password
```

Values in your config file can also be pulled from environment variables. To retrieve a value from the environment, set it to $<*key*>. For example:

```
[resilient]
password=$resilient_password
```

4.6. Add Values to Keystore

The Resilient package includes a utility to add all of the keystore-based values from your app.config file to your system's compatible keystore system. Once you have created the keys in your app.config file, run res-keyring and you are prompted to create the secure values to store.

```
res-keyring
Configuration file: /Users/kexample/.resilient/app.config
Secrets are stored with 'keyring.backends.OS_X'
[resilient] password: <not set>
Enter new value (or <ENTER> to leave unchanged):
```

4.7. Run Resilient Circuits

Once configuration is complete, you can run Resilient Circuits with the following command:

```
resilient-circuits run
```

If everything has been successful, you should see lots of output to your shell, including a components loaded message. For example:

```
<load_all_success[loader] ( )>
      2017-03-06 11:04:35,525 INFO [app] Components loaded
```

You can stop the application running with ctrl+c.

4.7.1. Run Multiple Instances

Running the application creates a hidden file called “resilient_circuits_lockfile” in a “.resilient” directory in your home directory. This is to prevent multiple copies of the application from running at once. If your particular situation requires running multiple instances of Resilient Circuits, you can override this behavior by specifying an alternate location for the lockfile via an “APP_LOCK_FILE” environment variable.

4.7.2. Monitor Config File for Changes

You can configure Resilient Circuits to monitor the app.config file for changes. When it detects a change has been saved, it updates its connection to the Resilient appliance and notifies all components of the change. To enable this option, install the “watchdog” package.

```
pip install watchdog
```

Now you can run with:

```
resilient-circuits run -r
```

Without the –r option, changes to the app.config file have no impact on a running instance of Resilient Circuits. Note that not all components currently handle the reload event and may continue using the previous configuration until Resilient Circuits is restarted.

4.7.3. Override Configuration Values

Sometimes it is 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
```

4.8. Run as a Service

You can configure Resilient Circuits to run as a service on a Red Hat Enterprise Linux or Windows system.

4.8.1. Systemd on RHEL

Systemd is a process control program available on a variety of modern Linux systems. It is available on the RHEL-based Resilient appliance. You need to create an OS user for the service. On RHEL Linux:

```
sudo adduser integration --home /home/integration
```

Systemd uses unit configuration files to define services. Copy the configuration file provided below to your integration machine and edit as necessary. The configuration file defines the following properties:

- OS user account to use.
- Directory from where it should run.
- Any required environment variables.
- Command to run the integrations, such as resilient-circuits run.
- Dependencies.

Here is an example of a configuration file. Copy this text to a file called resilient_circuits.service and edit the content to match your setup. If you are not running on the Resilient appliance, then the “After” and “Requires” lines in the [Unit] section should be removed.

```
[Unit]
Description=Resilient-Circuits Service
After=resilient.service
Requires=resilient.service

[Service]
Type=simple
User=integration
WorkingDirectory=/home/integration
ExecStart=/usr/local/bin/resilient-circuits run
Restart=always
TimeoutSec=10
Environment=APP_CONFIG_FILE=/home/integration/.resilient/app.config
Environment=APP_LOCK_FILE=/home/integration/.resilient/resilient_circuits.
lock

[Install]
WantedBy=multi-user.target
```

Copy this to the configuration directory and tell systemd to reload and enable the new service:

```
cp resilient_circuits.service
/etc/systemd/system/resilient_circuits.service
sudo chmod 664 /etc/systemd/system/ resilient_circuits.service
sudo systemctl daemon-reload
sudo systemctl enable resilient_circuits.service
```

To start or stop the resilient_circuits service, run:

```
sudo systemctl start resilient_circuits.service
sudo systemctl stop resilient_circuits.service
```

4.8.2. Windows

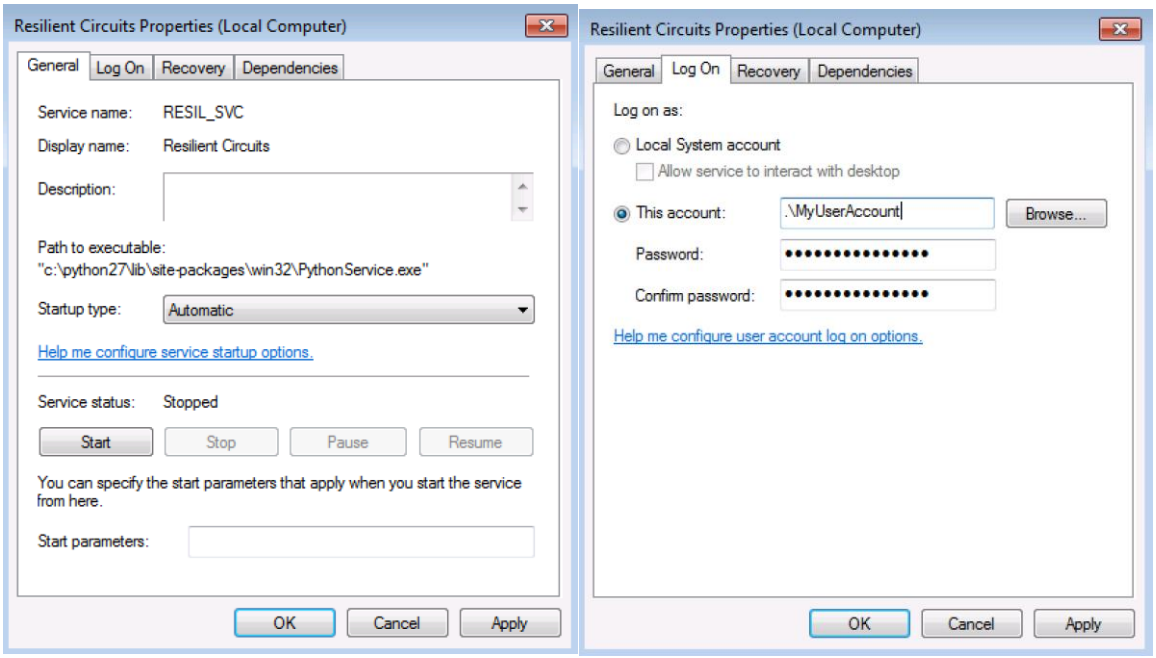
Resilient Circuits can be configured to run as a service. It requires the pywin32 library, which should be downloaded from sourceforge. Instructions for downloading and installing the correct package are at the bottom of the screen and must be followed carefully. Do not use the pypi/pip version of pywin32.

Installation of the wrong version of the pywin32 library will likely result in a Resilient service that installs successfully but is unable to start.

Now run:

```
resilient-circuits.exe service install
```

Once installed, you can update the service to start up automatically and run as a user account.



It is recommended that you log in as whichever user account the service will run as to generate the config file and confirm that the integration runs successfully with “resilient-circuits.exe run” before starting the service.

Commands to start, stop, and restart the service are provided as well.

```
resilient-circuits.exe service start
resilient-circuits.exe service stop
resilient-circuits.exe service restart
```

5. Developing Functions

Before you write a function, you must understand the purpose of the function, the inputs the function needs to perform its activity, the expected results, and the actions or decisions to be made based on the results.

If you wish, you can create multiple functions related to the same integration and include all of them in one package.

TIP: If you have access to functions that are similar to the one you wish to create, use that function as a template to save time.

5.1. Create the Resilient Platform Components

Before you write the function processor code, you need to create the function and associated components as follows. For detailed procedures on creating each component, see the *Resilient Incident Response Platform 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 for the function. Set the Type to **Queue** and Expect Acknowledgement to **Yes**.
3. Go to the Functions tab and create the function then define its properties:
    - **Name**: Enter a unique name that describes the purpose of the function.
    - **API Name**: Generated by the system and based on the Name field. This is the name you use when you write your function processor.
    - **Message destination**: Select the message destination you created in the previous step.
    - **Description**: Enter a description of the function's purpose (what it does), overview of the inputs (and guides to logic needed in pre-process script), and overview of outputs (and guides to logic needed in post-process script).
    - **Inputs**: One or more fields whose values are inputs for the function. See Input Field Considerations for guidelines.
4. 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.
5. If you require a data table to receive data from the function, 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.
6. Go to the Workflows tab and create or modify one or more workflows to include the function.

7. Add the function then enter the following:
    a. Determine whether to enter the input values manually in the Input tab or use a pre-process script to determine the input values.
    b. Choose whether to save the output of a function for use in the pre- or post-process scripts for functions further in the workflow.

Enter a post-process script to act upon the result. See You should test all required input parameters for a valid entry. You can also enforce this when defining the input field (Requirement: Always). Pre-process scripts are needed for field assignment.

```
incident_id = kwargs.get('incident_id')
 if not incident_id:
     yield FunctionError('incident_id is required')
```

    c. Post-Process Script Considerations for guidelines.

8. At the Rules tab, create the rule to execute the workflow you created. Make sure that the rule does not include a message destination. The function defines the message destination.
9. Go to Administrator Settings then click the Organization tab. Use the Migrate Settings – Export feature to export the Resilient platform settings to an export file. See the *Resilient Incident Response Platform System Administrator Guide* for details.

TIP: To avoid conflicts with other functions, use a prefix for your message destinations, functions, input fields, data tables, and so on. Use the name of the integration as the prefix, such as 'jira_', 'ldap_', and 'qradar_'.

5.1.1. Input Field Considerations

Consider the following when defining input fields.

- Input fields are referenced via the inputs object, for example inputs.id.
- For any 'id' input field such as incident.id or artifact.id, you must define the corresponding input field as Number.
- When sending an ID field, such as incident.id, to an input field, you must define the input field as Numeric.
- Text Areas, such as incident.description, must to be passed to input fields using the 'content' property. For example: inputs.fn_description = incident.description.content
- In the Resilient platform, use the Tooltip field to provide extra information about the input. Use an example if helpful.
- Use care when processing Textarea fields, which can contain rich text. Rich text is passed to your function as HTML tags. For example, a bold word is passed as 'word'. This can be confusing to integration applications that do not anticipate this tagging. In most cases, it makes sense to strip out these tags.
- An alternative to naming individual input fields for each parameter to pass to an integration, use a single input field containing a JSON string:

```
inputs.test_details = """{{ "incidentId": "{0}",
"name": "{1}",
"description": "{2}" }}""".format(str(incident.id), incident.name,
incident.description.content)
```

- Decoding this input field may require the removal of control characters:

```
test_details = kwargs.get("test_details")  # text
```

```
mpa = {}.fromkeys(range(32))
dict = json.loads(test_details.translate(mpa))
log.info("incidentId: "+dict['incidentId'])
```

- Binary format, such as an attachment, is not supported. If a function needs the content of an attachment, do not send it through input fields. Instead, the integration needs to call the resilient_client of its super class to get the file content. For example:

```
resilientClient = self.rest_client();
"""
     Example of call:
    /incidents/2095/artifacts/13/contents
"""
api = "/incidents/{}/artifacts/{}/contents".format(incidentID, artifactID)
response = resilientClient.get_content(api)
```

- Input fields can be a composite of multiple Resilient fields, for example:

```
inputs.jira_description =
"Incident types: {}\nNIST Attack Vectors: {}\n\nAdditional Information:
{}"
   .format(incident.incident_type_ids, incident.nist_attack_vectors,
            incident.description)
```

You should test all required input parameters for a valid entry. You can also enforce this when defining the input field (Requirement: Always). Pre-process scripts are needed for field assignment.

```
incident_id = kwargs.get('incident_id')
 if not incident_id:
     yield FunctionError('incident_id is required')
```

5.1.2. Post-Process Script Considerations

Consider the following when defining post-process scripts.

- Use the results object to access the data returned from an integration. Due to limitations in the way Python scripts can be written, the syntax to access the JSON data can vary:
  - results.matched_list should be used rather than results['matched_list']
  - results.matched_list['file'] should be used for the next level item
- Failed post-process scripts may cause a workflow to remain in the running state. Reference the Action Status to verify the successful completion of a function. Reference the Workflow status to terminate any workflow with failed actions.

5.2. Write the Function Processor

A function processor component, in this framework, is a Python class that implements *function methods*. These functions are called by the framework when the Resilient platform invokes the function from a workflow.

To write the Python code that performs the function's integration logic, start by using codegen to generate a Python package with a boilerplate implementation. This package includes everything needed to make your function **installable**. Besides your code, it can include the function definition, custom fields, data tables, workflows and rules, which you created on the Resilient platform.

```
resilient-circuits codegen --package pckg_name --function func_name
```

NOTE: To see additional options on the use of codegen, such as packaging multiple functions, see Codegen Uses.

The following example packages a function called lookup_mode_by_id and a workflow called lookup_model into a package called fn_model.

```
$ resilient-circuits codegen -p fn_model -f lookup_model_by_id --workflow lookup_model
Codegen is based on the organization export from 2018-04-12 14:46:34.355000.
Writing ./fn_model/MANIFEST.in
Writing ./fn_model/README.md
Writing ./fn_model/fn_model/LICENSE
Writing ./fn_model/fn_model/__init__.py
Writing ./fn_model/fn_model/components/__init__.py
Writing ./fn_model/fn_model/components/lookup_model_by_id.py
Writing ./fn_model/fn_model/util/__init__.py
Writing ./fn_model/fn_model/util/config.py
Writing ./fn_model/fn_model/util/customize.py
Writing ./fn_model/setup.py
Writing ./fn_model/tests/test_lookup_model_by_id.py
Writing ./fn_model/tox.ini
$
```

The result is a directory containing the essential files for an installable Python package that implements the function or functions specified. Within this package, the function code itself is a simple script, such as the one below, that you can edit to add your integration logic. This script is a *component* with a method, *decorated* with **`@function()`** that tells the Resilient Circuits framework how to call it.

```
"""Function implementation"""

import logging
from resilient_circuits import ResilientComponent, function,
StatusMessage, FunctionResult, FunctionError


class FunctionComponent(ResilientComponent):
    """Component that implements Resilient function 'lookup_model_by_id"""


    @function("lookup_model_by_id")
    def _lookup_model_by_id_function(self, event, *args, **kwargs):
        """Function: Lookup more information about the specified ID"""
        try:
            # Get the function parameters:
            model_id = kwargs.get("model_id")  # text

            log = logging.getLogger(__name__)
            log.info("model_id: %s", model_id)
```

```
            # PUT YOUR FUNCTION IMPLEMENTATION CODE HERE
            #  yield StatusMessage("starting...")
            #  yield StatusMessage("done...")

            results = {
                "value": "xyz"
            }

            # Produce a FunctionResult with the results
            yield FunctionResult(results)
        except Exception:
            yield FunctionError()
```

First, the function gets its parameters (`model_id`, in this case). The boilerplate implementation logs the values for ease of debugging, although you may want to change that if it's too noisy.

At any stage during the function's processing, you can enter `yield StatusMessage("...")`, which provides a status message that will display to the Resilient user in the Action Status dialog. If your function might run for several seconds or minutes, this can be a useful way to show progress.

The `results` are a Python dictionary, containing named values that will be available in the workflow *output* and *post-processing script*. Typically, functions should return a small 'results' dictionary with one or a few named values, and you should include enough documentation to allow your users to understand how to find and use these results in a custom workflow.

5.3. Run the Function

Before running the function, you must install this package so that Resilient Circuits can load it. The most convenient way to install during development is with pip's *editable* flag (or `-e`). Using this, you can edit your source files directly without needing to reinstall after any changes.

```
pip install --editable ./pckg_name/
```

Run the integration code from the command-line, as follows. The framework reads your configuration file, connects to the Resilient platform, finds and loads all the installed components, then subscribes to the message destination for each function processor component. Leave it running; when a function is invoked, the code handles it.

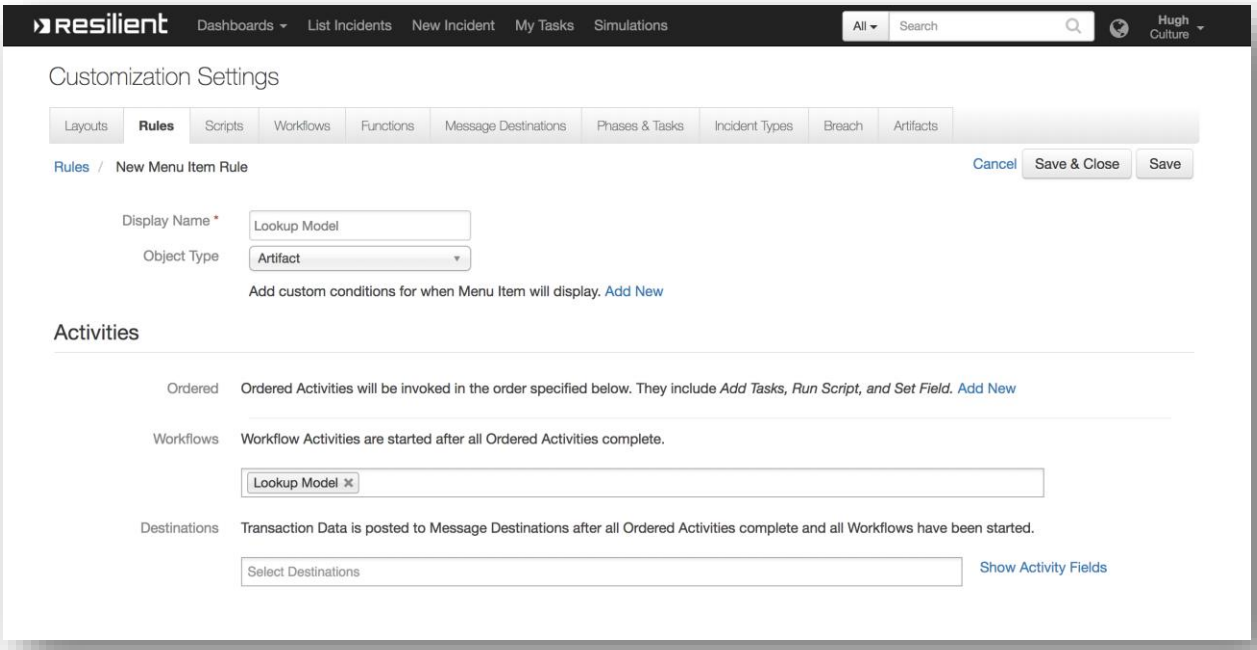
```
resilient-circuits run

2018-04-12 16:10:31,814 INFO [app] Configuration file:
/home/integration/.resilient/app.config
2018-04-12 16:10:31,816 INFO [app] Resilient server:
myserver.resilientsystems.com
2018-04-12 16:10:31,817 INFO [app] Resilient user: api@example.com
2018-04-12 16:10:31,818 INFO [app] Resilient org: PartnerLab
2018-04-12 16:10:31,818 INFO [app] Logging Level: INFO
2018-04-12 16:10:38,075 INFO [component_loader] Loading 1 components
2018-04-12 16:10:38,076 INFO [component_loader]
'fn_model.components.lookup_model_by_id.FunctionComponent' loading
2018-04-12 16:10:38,089 INFO [stomp_component] Connect to
myserver.resilientsystems.com:65001
2018-04-12 16:10:38,090 INFO [actions_component]
'fn_model.components.lookup_model_by_id.FunctionComponent' function
'lookup_model_by_id' registered to 'function_example'
2018-04-12 16:10:38,091 INFO [app] Components loaded
2018-04-12 16:10:38,094 INFO [app] App Started
2018-04-12 16:10:38,196 INFO [actions_component] STOMP attempting to
connect
2018-04-12 16:10:38,198 INFO [stomp_component] Connect to Stomp...
```

```
2018-04-12 16:10:38,199 INFO [client] Connecting to
myserver.resilientsystems.com:65001 ...
2018-04-12 16:10:38,825 INFO [client] Connection established
2018-04-12 16:10:39,090 INFO [client] Connected to stomp broker
[session=ID:ip-1-2-3-252.srv.resilientsystems.com-35733-1523282148180-
5:243, version=1.2]
2018-04-12 16:10:39,092 INFO [stomp_component] Connected to
failover:(ssl://myserver.resilientsystems.com:65001)?maxReconnectAttempts=
1,startupMaxReconnectAttempts=1
2018-04-12 16:10:39,093 INFO [stomp_component] Client HB: 0  Server HB:
15000
2018-04-12 16:10:39,094 INFO [stomp_component] No Client heartbeats will
be sent
2018-04-12 16:10:39,095 INFO [stomp_component] Requested heartbeats from
server.
2018-04-12 16:10:39,098 INFO [actions_component] STOMP connected.
2018-04-12 16:10:39,205 INFO [actions_component] Subscribe to message
destination 'function_example'
2018-04-12 16:10:39,206 INFO [stomp_component] Subscribe to message
destination actions.201.function_example
```

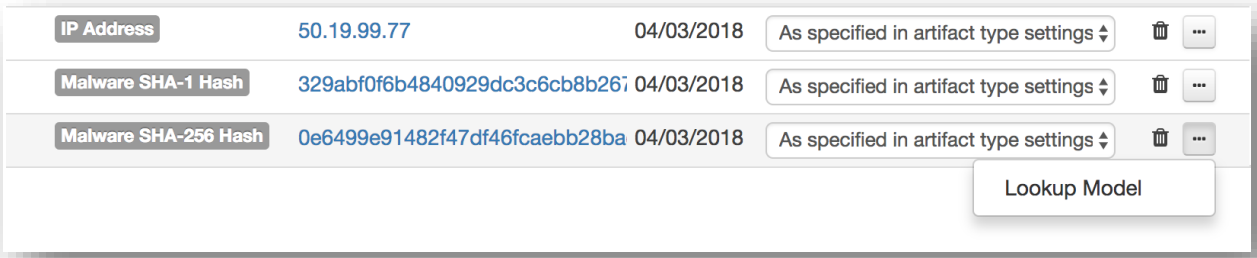
The framework is running and waiting for the function to be called.

5.4. Test the Function

You test the function by using it from the Resilient platform. Use a rule to trigger the workflow that contains the function. The rule can be a menu-item rule, which shows on its object (incident, artifact, task, etc.) when the conditions are met, or an automatic rule that runs the workflow when an object (incident, artifact, task, etc.) is created or modified and meets the rule's pre-defined conditions.



For this example, the workflow is meant to run on an artifact, so the rule is a menu-item rule that appears as an action in each incident's artifact action menu, accessible from the [...] button.

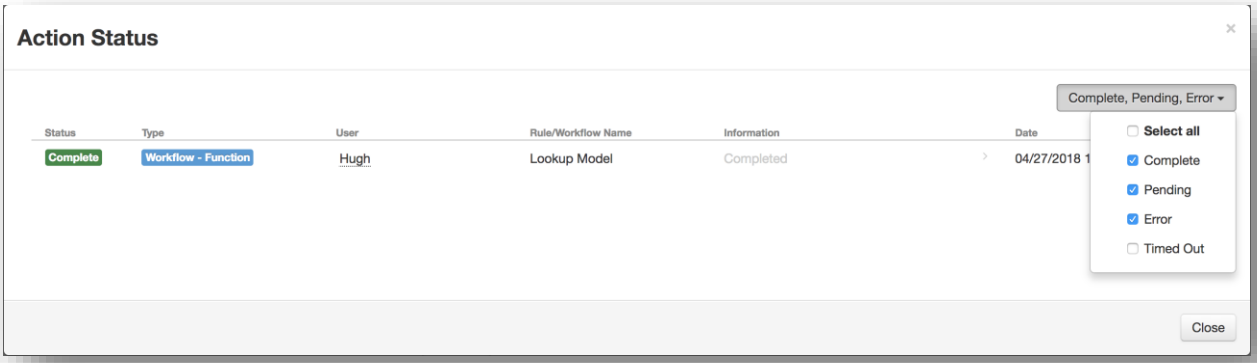


Select your action to start the workflow and call your function.

At the integration console, you see the function message arrive, including the logging message to print the function's parameters as part of the boilerplate code.

```
2018-04-12 16:24:23,235 INFO [actions_component] Event:
<lookup_model_by_id[] (id=219, workflow=lookup_model,
user=who@example.com) 2018-04-12 16:23:20.221000> Channel:
functions.function_example
2018-04-12 16:24:23,237 INFO [lookup_model_by_id] model_id:
0e6499e91482f47df46fcaebb28bac985193aab331beeb5bc553162b422c1f21
```

The incident's Action Status menu shows that status of each rule, which can be pending (queued for delivery to the function processor), processed successfully, or with an error. In the following example, the action completed with success and included a status message.



If you need to make any changes to any of the function's elements, export the Resilient platform settings and repackage your function processor using codegen.

If your package deploys data tables or custom fields, the changes in the Resilient platform layouts may not be preserved. Therefore, you should document those changes in your submission and advise the reader to make those changes manually in the platform.

5.5. Deploy to a Different Resilient Platform

When satisfied with the test results, you can deploy the function to any Resilient platform at the same version as your test platform. You do this by compressing the .py files into a tar.gz file then distributing that file. Your package should include the functions and all the custom fields, data tables, workflows and rules.

Consider creating a document that describes the function and all its components, as how to install it. Note that importing a function's custom fields and custom data tables does not update the tab layouts. Ensure that you specify in the documentation associated with your function how to create the layout changes intended.

A Resilient administrator must have a Resilient Circuits installation that points to their Resilient platform. The administrator simply unzips the package and uses the following command to import the function and its components directly into the platform.

```
pip install ./pckg_name/
```

Alternately, they can unzip the package and use the Resilient Circuits customize utility to update their Resilient platform with any missing message destinations, function definitions, and other design elements. The command is:

```
resilient-circuits customize
```

The optional parameter `-y` applies all the customizations without asking for confirmation.

Each installed package that includes a "customize" entry-point is called and returns a collection of customizations described in Resilient JSON format. These are used to update the Resilient platform.

The customize command only rebuilds those components that were referenced from the codegen command line. For example, if you indicated --workflow, then workflows are rebuilt during the customize process.

6. Coding Considerations

The following sections provide advice and recommendations to consider when creating your function.

6.1. Codegen Uses

When you package your function, you use the following command:

```
resilient-circuits codegen --p pckg_name --f func_name
```

Optionally, you can specify multiple functions to codegen into the package by specifying each function name with the --f parameter.

Use the following command to see a list of functions on your integration system. Codegen returns the file path to the function, which you can open using a text editor, such as vi or nano. The following is an example of listing functions.

```
resilient-circuits codegen
Available functions:
    ldap_search
    splunk_search
```

You can output your function to a py file using the -o argument as follows:

```
resilient-circuits codegen --function <function_name> -o
<function_name>.py
```

When packaging your function, you can add specific elements, such as workflows, rules, and data tables, into the output file by using arguments such as --w, --rule, and --datatable. Use --h to list all the arguments. This is handy for functions that rely on complex data handling in pre-process or post-process scripts, workflows and rules. You need to have the export file, which contains the elements, on the system where you are running Resilient Circuits. The following command includes a workflow and rule, and specifies the export file:

```
resilient-circuits codegen --function <function_name> --workflow <name> --
rule <name> --exportfile <filename> -o <function_name>.py
```

6.2. Data Flow

Functions are blocking until results are returned. Returning `FunctionError()` will abort the result of a workflow.

All functions are stateless. No persistence of data is retained between function calls.

6.3. Error Handling

The high level code should be covered with try / except / finally blocks. Any exception should describe the issue for Resilient Action Status log.

The finally block should be used to perform any connection closing, temporary file cleanup, and so on.

6.4. Logging

Log information for debugging purposes. Sensitive information should never be logged, but can be obscured. For example, user.email@example.com can become us***@example.com.

6.5. Data Results

Use the function, FunctionResult(), to return the JSON for post-process script processing. A sample result should be included in the function's description field.

As much as possible, return top-level items or lists of items (or combinations):

```
{
   'item1': 'result1',
   'item2': 'result2'
 }
```

or

```
{ 'entries':
   [
      {'item1': 'result1', 'item2': 'result2'},
      {'item1': 'result3', 'item2': 'result4'}
   ]
 }
```

6.6. Metrics

Consider logging metrics, which can be used to track usage:

```
{
     'thread': '',
     'function': '',
     'execution_time': 1000 (in milliseconds)
     'timestamp': date time when function completed
}
```

6.7. Packaging Considerations

Consider implementing the actual integration code to the 3rd party system in a separate module from that which manages the Resilient framework. This separation of logic allows you to test the 3rd party integration separate from the code as part of Resilient Circuits.

```
components/
   <function_code>.py
   <3rd_party_integration_code>.py

tests/
   test_<function_code>.py
   test_<3rd_party_integration_code>.py
```

6.8. Linking to 3rd Party Integration Objects

Perform the following to create a live URL link back to the integrated system, such as a ticketing system.

1. Create a new incident custom field as a text area with Rich Text enabled.
2. Add it to the summary section of an incident's layout.
3. In the function's post-process script, build the URL as an HTML anchor:

```
incident.properties.pd_incident_url = "<a href='{}'
target='blank'>PagerDuty
Link</a>".format(results.pd['incident']['html_url'])
```

6.9. Rich Text

Rich text fields may contain html markup. This format may be undesirable for the target integration system. A method such as the one below can strip off the html elements, preserving some of the new line format:

```
def _cleanHtml(htmlFragment):
    '''
    Resilient textarea fields return html fragments. This routine will
remove
      the html and insert any code within <div></div> with a linefeed
    :param htmlFragment:
    :return: cleaned up code
    '''

    tmp = re.sub(r'</div>', '\n', htmlFragment)
    tmp = re.sub(r'</ol>', '\n', tmp)        # numbered lists
    tmp = re.sub(r'</li>', '\n', tmp)        # unnumbered lists
    return re.sub(r'<([^>]+)>', '', tmp)       # removes all remaining
html
```