IBM Security

IBM

IBM Resilient SOAR Platform
App Developer's Guide
V37

Date: June 2020

Resilient SOAR Platform
App Developer's Guide

Version	Publication	Notes
37.0	June 2020	Initial publication.

Contents

Chapter 1. Introduction

Using the information provided in this guide, you can write your own apps that integrate the Resilient SOAR Platform with your organization's existing security and IT investments. Apps make 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 apps 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.

What is a Resilient app?

A *Resilient app*, formerly called extension or integration, is a collection of playbook components, code executables or both that represent an end-to-end function and is deployed to the Resilient platform.

A Resilient app is available in two formats to support App Host Kubernetes-based container environments or the previously available extension format that requires an integration server.

The container-based format uses containers for improved usability, manageability, and security. A Resilient app in this format is installed directly to the Resilient platform. A Resilient app in the extension format requires that the extension is loaded on to an integration server then deployed to the Resilient platform.

An app can contain one or more of the following elements:

- **Resilient playbook components**. These are the rules, workflows, functions, scripts, tasks, message destinations, fields, and data tables that you initially create in the Resilient platform. The message destinations make data accessible to external code. Fields and data tables can be populated by external data. The other components process and make data accessible to users.
- **Function processor**. Python code you supply that performs an activity and returns the results to the workflow with the function that triggered the process. Activities include access and return external data, interact or integrate with other security systems, or be a utility that performs a specific action. Typically, an app with a function processor would also provide Resilient functions, workflows, rules and a message destination.

Resilient SDK

The Resilient SDK allows you to create apps in the container and extension formats.

The Resilient SDK provides the template files to develop your app and specify how to run your app in a container. Containers run Python 3.6.8 and a framework that automatically manages authenticating and connecting to the STOMP connection and REST API in the Resilient platform. It simplifies creating apps by allowing you to focus on writing the behavior logic.

The Resilient SDK provides the following features:

- The Python environment and template files with which you write your code. Your code can take the form of a function processor, action processor, or a utility that acts upon data received from a Resilient organization but does not return data to the Resilient organization.
- A dockerfile that provides all the information needed to build a container for the app. You can modify the file in those situations where your app requires additional operating system or Python components. The dockerfile is compatible with Docker or other container management solutions, such as Red Hat Enterprise Linux Podman.
- An apikey_permissions.txt file with a list of permissions to choose from. You can easily select which Resilient permissions you need for your app. When your completed app is installed in a Resilient organization, the system automatically generates an API key account with the permissions you specified.

- A documentation template that prompts you for the information need to generate installation and user instructions.
- The ability to generate your app in the Resilient extension format for earlier versions of the Resilient platform that do not support the app format.
- The ability to convert existing Resilient extensions to the app format.

If you are familiar with the Resilient Circuits framework used with Resilient extensions, Resilient Circuits is also used with apps but within the container. You do not interact with Resilient Circuits directly.

The Resilient SDK has the ability to create the app in the extension format with the same functionality but for earlier versions of the Resilient platform that do not support the container-based apps. The Resilient SDK also has the ability to repackage existing extension apps into a container-based format.

Resilient REST API

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. Custom behaviors are triggered by adding a message destination to a rule, workflow or function defined in the Resilient platform and subscribing your code to that message destination.

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.

Playbook components

A Resilient dynamic playbook is the set of rules, conditions, business logic, workflows and tasks used to respond to an incident.

You should be familiar with your organization's customized Resilient playbook when designing an app. 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.
- 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.

The function processor, which is the code you write, 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. The function processor performs an activity then sends the results to the workflow that invoked the function, which allows the workflow to act upon the results.

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

Development overview

Before you write an app, 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.

Before you start developing your app, you need to make a number of decisions. These include:

- Does the app facilitate communication between the Resilient platform and a security program?
- What data does the app require from the Resilient platform?
- What data will the app send to the Resilient platform? If sending data to the Resilient incident, you need to write a function processor or action processor.
- Is the data used to facilitate automated decision-making? If so, should the data be presented to a function, workflow or rule? You use a function processor to send data to a function; otherwise, you use an action processor.
- Is the data presented to be made available directly to a user, such as a security analyst, for analysis and manual action? If so, the data can be presented to the user such as in a data table or custom field. You can populate a data table or custom field directly from an action processor or indirectly through a function and workflow with a function processor.

These decisions assist in determining which Resilient playbook components to create, type of processor to write, and the Resilient permissions needed for the app to accomplish its objective.

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

1. Create your development environment, which includes installing and configuring the Resilient SDK on your system.
2. Log into the Resilient platform and create those playbook components required for your app. You then export the components for use in your development environment. Optionally, test the components before packing them in your app.
3. Use the Resilient SDK codegen to write the function processor. Alternately, you can write a utility that receives data but does not send data to the Resilient platform.
4. Modify the apikey_permissions.txt to select those permissions required by your app. When you app is installed, the Resilient platform automatically generates a Resilient API key account with the specified permissions for exclusive use by the app.
5. Modify the dockerfile only if your app requires additional operating system or Python components.
6. Connect to a Resilient platform to test the app.

7. Use the Resilient SDK docgen to generate a document that provides information on deploying and using the app.
8. Package your app then distribute it to other Resilient administrators. They can deploy the package to any Resilient platform at the same or later version as your test platform. Optionally, share your app with other developers in the IBM Resilient developer community.

Developer website

The Resilient developer web site contains example apps, and documentation. The links are provided below.

- 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 apps, example scripts, and developer documentation. It also contains the Resilient SDK. This is also accessible from the developer website reference page.
- You can also obtain the Resilient SDK and other Python packages from Pypi.
- 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.

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 2. Development environment

Make sure you have the proper environment for developing apps with the Resilient SDK.

You install the Resilient SDK on a system different than the one hosting the Resilient platform.

The following lists the requirements for the system hosting the Resilient SDK.

- Container tool, such as Docker or Podman
- Operating system:
  - Apple Mac
  - Red Hat Enterprise Linux 7.4 to 7.7, Centos Linux 7.*x* (where *x* is 4 or later) and CentOS 8.
  - Windows (Windows Enterprise Server 2016 or later is recommended)

    If using a Windows server, you also need to install the pywin32 library, which should be downloaded from here. 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.
- Minimum of 5GB free disk space.
- Minimum of 8 GB RAM.
- Python enabled IDE, such as PyCharm for Microsoft Visual Studio, to edit files.
- Access to the PyPI web site or a local Python environment such as Github.

In addition, you need access to a Resilient platform V37.1 or later. You create those playbook components you need for you app on the Resilient platform. IBM Resilient recommends that you use a Resilient platform in a test environment to create the components needed for your app. This can be a separate Resilient platform or a Resilient organization within your platform dedicated for testing. Once tested, you can deploy the app into any Resilient platform that is at the same or later version as your test platform.

If the Resilient platform is beyond a firewall, such as in a cloud configuration, configure the firewall to allow the Resilient SDK host access to the following ports:

- 443. Required for the SDK to connect to Resilient data using the REST API. This an “inbound-only” connection from the Resilient SDK host to the Resilient platform.
- 65001. Required to communicate with the platform using ActiveMQ OpenWire. The connection is bidirectional.

Download the Resilient SDK from Pypi as a Python command line package.

Chapter 3. Playbook components

Before you write your app code, create the Resilient playbook components on a Resilient platform.

You create playbook components at a Resilient organization. For detailed procedures on creating each component, see the *Resilient SOAR Platform Playbook Designer Guide*.

Also consider the following conventions for naming your app and its associated components:

- The name of your app should be meaningful as well as be unique. For example, LDAP Query, Machine Learning for Resilient, and ElasticSearch Functions for Resilient.
- The components you create should reflect the app name as well as be unique. Using LDAP Query as an example, here are some component names:
  - Function name: ldap_query
  - Message destination name: ldap_query
  - Input field: ldap_query_inputname
  - Data table: ldap_query_datatable
  - Custom field: ldap_query_fieldname
  - If you includes rules or workflows that are meant as examples, you should use the prefix, Example, such as “Example: Query LDAP”. If using a function, you use the API name of the workflow, so an example of a workflow name would be "example_ldap_query_workflow_name".

  You do not need to add the app name prefix to commonly used fields, such as incident_id, task_id, attachment_id, and artifact_id.

  NOTE: IBM Resilient has a common library of these fields which you can use.

Optionally, test each function individually on the Resilient platform before packaging the app to ensure that the Resilient components are working as expected. The *Resilient SOAR Platform Playbook Designer Guide* includes an example of how to test functions.

If using functions, also see “Function input field considerations” on page 7 and “Function post-process script considerations” on page 8.

Function 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, you must define the corresponding input field type as Number. Frequently used ID input fields include:
  - incident_id
  - artifact_id
  - task_id
  - attachment_id
- When sending an ID field to an input field, you must define the input field as Numeric.
- Text Areas, such as incident.description, must 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 apps 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 app, 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 app 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)
```

  NOTE: IBM Resilient provides a helper library of common functions such as reading attachments from the Resilient platform. The library is resilient-lib and is found in Github. It is also pip installable.

- 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. Another common function available in resilient-lib: validate_fields()

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

NOTE: Another common function available in resilient-lib: validate_fields()

Function post-process script considerations

Consider the following when defining post-process scripts.

- Use the results object to access the data returned from an app. 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. For a given incident, click the **Actions** button then select **Action Status** to verify the successful completion of a function. Click the **Actions** button then select **Workflow status** to terminate any workflow with failed actions.
- Use dict.get("key") when testing whether data exists for that key. The dict["key"] may cause the script to fail if "key" does not exist.

Chapter 4. Write the function processor

You create an app based on Resilient functions.

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

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, 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-sdk codegen --package pckg_name --function func_name
```

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

```
$ resilient-sdk codegen -p fn_model -f lookup_model_by_id --workflow lookup_model
Generating codegen package...
Connecting to Resilient Appliance...
'codegen' complete for 'fn_model'
```

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 app logic. This script is a *component* with a method, decorated with **`@function()`**.

```
"""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-process script. If possible, your functions should return a small 'results' dictionary with one or a few named values. In this case, you should include enough documentation to help your users understand how to find and use these results in a custom workflow.

Coding considerations

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

Data flow

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

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

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.

Do not use `yield FunctionError()` within try/except. It causes the message destination message to remain stuck. Each time Resilient Circuits is restarted, the message is attempted again. The better solution is call `raise FunctionError()` and then the except block can perform the `yield`. The `yield FunctionError()` also interferes with `finally` in try/except/finally. The fix is to change `yield` to `raise`.

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.

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'}
   ]
 }
```

Links back to the app are supported by creating a Resilient custom field, such as a TextArea with rich text enabled. Add the custom field to an incident's Summary section for easy access. In the workflow's post-processing script, add code similar to this:

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

Data tables are a convenient way to return and display row results. There are a number of considerations to bear in mind with their use:

- Data tables are global to an incident (as are all custom fields). This means that the table rows are not unique to a specific object, such as an artifact.
- In order to deal with the shared use of tables, add two columns to a table to identify the search argument (such as artifact.value) and a timestamp when the results were returned.

Linking to 3rd party app 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'>reference Link</
a>".format(results.pd['incident']['html_url'])
```

Rich text

Rich text fields may contain HTML markup. This format may be undesirable for the target app 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
```

Temporary files

For integrations that read from or write to files, one method to create temporary files is to use Python's tempfile:

```
import tempfile
...
with tempfile.NamedTemporaryFile() as temp_file:
```

In situations when filehandles are used, consider using StringIO instead. This avoids any OS interaction with files:

```
import StringIO
...
with StringIO(filedata) as temp_file:
```

For more information, see https://docs.python.org/2.7/library/stringio.html.

Avoid using self.

Using `self.` in functions is not thread-safe. This means that `self.` is shared by all functions (even though they are running in different threads). Therefore, a statement such as, `self.connection =`

`Connection` will likely be overwritten when more than one function is invoked and running at the same time.

The better solution is to use Class level variables and only use `self.` for class function and Resilient static functions.

Example: Making an API call

If using Python to access the Resilient directly, the following script shows how to authenticate using an API key account and make an API call to get a list of constants from the Resilient REST API.

```
"""
    This script uses API Key Authentication with Resilient platform to
    get list of constants from a server. Modify as needed.

    Prerequsites:
    * Python 2.7 or higher

    It is recommended that you run this in a virtualenv as follows:
    * pip install virtualenv
    * virtualenv myvirtualenvdir
    * . myvirtualenvdir/bin/activate

    Once activated, verify that the requests module is installed
    * pip install requests
"""
from __future__ import print_function
from requests.auth import HTTPBasicAuth
import requests
import json


def main():
    """
        * Resilient version must be 33.0 or higher
        * API Key ID and Secret are known. Sample values shown in script.
        * API Key has the permissions needed for the operation
    """

    key_id = "c5b0d84c-73d6-4dca-a9fb-e98870d62826"
    key_secret = "e2-5k4IT3WlVH9mfSl_g5ALBdtmVEaRsh4EEoFUs2HI"
    server = "9.70.195.31"
    resource = "rest/const"
    url = "https://{0}/{1}".format(server, resource)
    headers = {"Content-Type": "application/json; charset=UTF-8"}
    auth = HTTPBasicAuth(key_id, key_secret)

    req = requests.get(url, headers=headers, auth=auth, verify=False)
    print(json.dumps(req.json(), indent=4, sort_keys=True))


if __name__ == "__main__":
    main()
```

Chapter 5. Test the app

You can test the Resilient functions separately and test the app for proper permissions and connectivity.

You can test your functions individually on the Resilient platform before creating the app to ensure that the Resilient components are working as expected. The *Resilient SOAR Platform Playbook Designer Guide* includes an example of how to test functions.

Once you build the app, you can create a container on your local system then connect with a Resilient organization. To do this, you must have access to a Resilient platform with an account that has the proper permissions. This test is to ensure that the permissions have been set correctly and there is communication with the message destination. If your app uses API calls, you should attempt to make that call to exercise.

Before running the app, you must install this package. 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/
```

Create your docker image. From within your development folder, run:

```
docker build . -t resilient/<fn_example>:1.0.0
```

Run the app 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.

```
docker run -v /path/to/your/local/app.config:/etc/rescircuits/app.config resilient/
fn_example:1.0.0
```

The -v flag in the command mounts your local app.config file into the container, so that Resilient Circuits is aware of the settings that you have configured locally. When you run this command, you should see Resilient Circuits initialize, load the relevant functions for your app, subscribe to the message destination, and run successfully.

The app.config file contains parameters for your app that are subject to change or provide tuning of application settings such as timeout values. You may need to adjust those settings.

You can test connectivity by installing your app on the Resilient platform. You do not need an App Host. Log in to the Resilient platform as an administrator, go to **Administrator Settings** and click the **Apps** tab. Install the app. When done, click the app and click it's **Configuration** tab. Click **app.config**. At the bottom of the **App Settings** page, click the **Test Configuration** button. For details about navigating the **Apps** tab, see Apps.

NOTE: If you have a Resilient integration server, you can use this environment to test your app using the Resilient Circuits selftest command.

If there are any changes to your app, update the app as follows:

```
resilient-sdk codegen --reload -p <app_name>
```

Chapter 6. Document your app

IBM Resilient strongly recommends that you create an installation guide and a user guide, where the latter describes the function and all its components.

The Resilient SDK provides a docgen command that walks you through the process of creating the installation and user guides. Once you have finalized your app, enter the following command to start the process:

```
resilient-sdk docgen
```

The document files are placed in a subfolder called Docs.

If your package contains custom fields and data tables, make sure to document that the Resilient playbook designer should include these in the incident layouts, since the fields and data tables cannot be automatically added to layouts. When possible, provide details on the intended incident layouts, such as a New Incident Wizard or a specific incident tab.

You should also document that re-importing a function's custom fields and custom data tables does not restore any layout changes made. Therefore, the layout changes may need to be recreated.

Chapter 7. Package your app

When you have finished development and testing, you package your app for use with other Resilient platforms.

You can deploy the package to a Resilient platform in your environment different than the one you were using for testing, or make it available to the Resilient community.

To package your command, perform the following:

1. Enter the command to create the Dockerfile, apikey_permissions.txt and entrypoint.sh files:

   ```
   resilient-sdk codegen --reload
   ```

   The apikeys_permissions.txt file is used with other key files, such as setup.py to capture metadata about your application. Dockerfile and entrypoint.sh are essential for containerizing your app. In almost all cases, the entrypoint.sh file should not be edited. It serves as a means to execute Resilient Circuits and monitor changes to the app.config file. Any changes to app.config causes Kubernetes to restart the container.
2. Edit Dockerfile as follows:
   - Change <app_name> in `ARG APPLICATION=<app_name>` to the name of your app. It should match the `name=` parameter in your setup.py file.
   - Change the version number in `ARG RES_CIRCUITS_VERSION=xx.x` to match the latest version of Resilient Circuits found on PyPI.
   - In advanced cases where you need to install additional operating system level packages, edit the following section:

     ```
     # uncomment and replicate if additional os libraries are needed
     #RUN yum -y update && yum clean all
     #RUN yum -y install <package>
     ```

   - In advanced cases where you need to install additional Python packages, edit the following section:

     ```
     # uncomment and replicate if additional pypi packages are needed
     #RUN pip install <package>

     # uncomment and replicate if additional local packages are needed
     #COPY /path/to/extra_package /tmp/packages/.
     #RUN pip install /tmp/packages/<extra_package>*.tar.gz
     ```

   - In advanced cases where you need to expose one or more ports for your app, edit the following section:

     ```
     # uncomment to expose port only if a custom threat feed
     #EXPOSE 9000
     ```

   The build process uses `dist/<app_name>-version.tar.gz` source distribution produced by the Python `setup.py dist`.
3. Save the dockerfile.
4. Use your container tool to build a container image. For example, the following Docker command assumes the docker file is in the current directory and assigns a tags with a version number of 1.0.0. The container version must match the version specific in your setup.py file.

   ```
   docker build  -t your_company/your_custom_app:1.0.0
   ```

5. If your app needs more than read_data and read_function permissions, edit the apikey_permissions.txt file and uncomment those permissions you need.
6. Package your app:

```
resilient-sdk package
```

This creates three files: Dockerfile, apikey_permissions.txt and entrypoint.sh. The apikeys_permissions.txt file is used with other key files, such as setup.py to capture metadata about your application. Dockerfile and entrypoint.sh are essential for containerizing your app. In almost all cases, the entrypoint.sh file should not be edited. It serves as a means to execute Resilient Circuits and monitor changes to the app.config file. Any changes to app.config causes Kubernetes to restart the container.

7. Update the two template icon files, app_logo.png and company_logo.pg, in the icons folder with your company's app and company logo. By default, these icons are the standard IBM Shield logos.
8. Run the following command to create the source distribution of your application:

```
python setup.py sdist
```

9. Package the app as follows:

```
resilient-sdk package -p /path/to/your/app
```

The path to your app is where the setup.py file exists. If the file is in your local directory, you can use:

```
resilient-sdk package -p .
```

The resulting zip file is placed in the app's folder. Within this file are:

- *app_name*.tar.gz file, which is the app in the Resilient extension format.
- app.json file containing the app metadata and list of permissions used to create API key account. The information displays in the Apps Details tab in the Resilient platform is obtained from this file .
- export.resz file containing the playbook components, which you use to import these components when using the app as a Resilient extension.

You deploy the zip file to a Resilient platform, as described in the *Resilient SOAR Platform System administrator Guide*.

If you wish to use your app with releases of the Resilient platform earlier than V37.1, deploy the *app_name*.tar.gz file using the Integration Server, as described in the *Integration Server Guide*.

Chapter 8. Publish your app

In addition to deploying your functions to other Resilient platforms in your environment, you can share your functions with the Resilient community by publishing to the IBM Security X-Force App Exchange.

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

IBM Resilient recommends using the tag, apphost, for searches to easily find App Host enabled publications.

Chapter 9. Convert an extension into an app

You can convert an existing extension into an app format using the Resilient SDK.

In extension's directory, run the following command to generate the Dockerfile, apikey_permissions.txt and entrypoint.sh files.

```
resilient-sdk codegen --reload -p app_name
```

Perform the procedure in Chapter 7, “Package your app,” on page 17 to edit the files and package the app.