IBM Resilient

Incident Response Platform

PYTHON INTEGRATION DEVELOPMENT GUIDE v28.0

Resilient Incident Response Platform Python Integration Development Guide

Version	Publication	Notes
28.0	August 2017	Initial release.

Table of Contents

1. Overview

Based on a knowledgebase of incident response best practices, industry standard frameworks, and regulatory requirements, the Resilient Incident Response Platform helps make incident response efficient and compliant.

You can quickly and easily integrate the Resilient platform with your organization's existing security and IT investments. It makes security alerts instantly actionable, provides valuable intelligence and incident context, and enables adaptive response to complex cyber threats.

1.1. Architecture

There are a number of elements involved with an integration:

- 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.

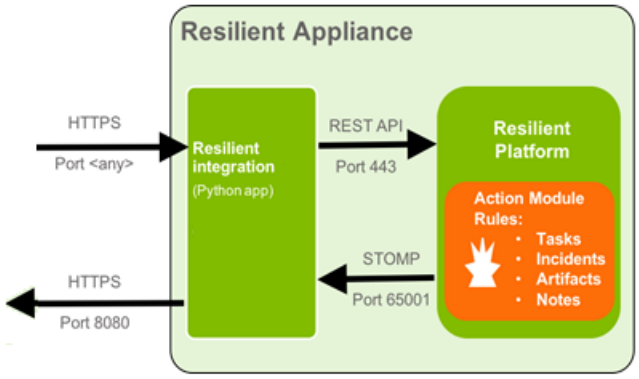
  The Resilient appliance has a fully-functional Rest API browser that lets you try out any endpoint on the system. When logged into the Resilient platform, click on your account name at the top right and select **Help/Contact**. Here you can access the complete Reference guide, including schemas for all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

- Co3 helper module. A Python library to facilitate easy use of the Rest API for several popular programming languages.

- Action Module. 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. For additional information, see the Action Module Programmer's Guide on the Customer Success Hub.

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

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



1.2. Download Locations

The Resilient Github repositories on the Resilient Success Hub contain the core Co3 and Resilient Circuit packages, additional integration packages, documentation and examples. The links are provided below.

- If not already done, request access to the Resilient Success Hub.
- Co3 helper module.
  - Co3 package for download.
  - Resilient Python API Readme file with instructions for installing the package.
  - Examples for using the Rest API, via the co3 package, to authenticate the Resilient platform and perform a variety of actions.
  - Additional examples.
- Resilient Circuits.
  - Resilient Circuits package for download. It requires that the co3 package be installed as well. Typically, the Resilient Circuits and Co3 packages are installed on the same system as the Resilient appliance, but you can install them on a remote Linux or Windows environment as described in the Installation section.
  - Resilient Python API Readme file with instructions for installing the package.
- AddTaskAction. Creates a rule for ease of adding ad-hoc tasks to an incident. This is used as an example of an integration installation and configuration.
  - rc-taskadd component for download.
  - Instructions for creating the message destination, rule, etc., to run this example
- Other Python examples.
  - https://github.com/Co3Systems/co3-api/tree/master/python/examples/action-module
  - https://github.com/Co3Systems/resilient-api-examples/tree/master/examples/resilientsystems.com/python/Circuits
- Integration Packages
  - https://github.com/Co3Systems/resilient-api-examples/releases
  - Source code and documentation
- pytest framework fixtures. Used for writing tests.
  - pytest plugin download
  - pytest documentation
- Resilient Circuits template, which you can use to create your own Python module.
- rc-webserver. Package and usage example
  - rc-webserver package
  - Web example

2. Installation

Typically, you would install everything on your Resilient appliance; however, you can install the co3 helper module and Resilient circuits framework, and manage your integration from a different system. Using a different system is useful if you have multiple Resilient integration packages in your environment.

If using a different system, it must be a Debian Linux or Windows system with Python 2.7 or later, or Python 3.4 or later, and have access to the Resilient appliance.

The installation procedures in this guide assume that all the packages are to be installed on the Resilient appliance.

2.1. Core Packages

Perform the following procedure to install the co3 helper module and Resilient Circuits framework.

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 co3 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 each component, co3 and, Resilient Circuits.

2.2. Component Packages

Once the Resilient Circuits and co3 helper module are installed, you can install components to perform customized behaviors in the Resilient platform. Some are available for download from GitHub and you can write your own as well. This guide provides an example installation and configuration of the rc-taskadd component, which creates a rule for ease of adding ad-hoc tasks to an incident.

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

```
pip install rc-taskadd-x.x.x.tar.gz
```

4. Verify that the component installed using the resilient-circuits list command. In the following example, you have version 27.0.0 of the rc-taskadd package installed in your environment and it provided the AddTaskAction component to Resilient Circuits.

```
bash-3.2$ resilient-circuits list

The following packages and components are installed:

rc-taskadd (27.0.0) installed components:

  AddTaskAction
```

5. Follow the instructions in the rc-task add readme file to create the message destination, rule, etc., in the Resilient platform, which are required for the AddTaskAction component to work.

3. Configuration

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 will 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”.

3.1. Create the Configuration File and Log Directory

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 This 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.

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

3.2. Edit the Configuration File

The [resilient] section of the configuration file controls how the core resilient_circuits and co3 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.

Parameter	Required?	Description
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 appliance.
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: • Installed components that should not be loaded. • 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 components 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 (e.g.; credentials to a 3rd party system).

3.3. 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
```

3.4. Add Values to Keystore

The co3 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.

```
bash-3.2$ 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. 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.

4.1. 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.

4.2. 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.

4.3. 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
```

5. Create a Resilient Circuits Component

You can create your own python module that contains your integration code.

5.1. Get Started

To develop a new custom component for Resilient Circuits to load, perform the following:

1. Create a directory to load it from. Create a directory called something like “components” and then add a values for “componentsdir” to your app.config file set to the absolute path of this components directory.
2. 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, or click here to use the template.
3. 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.

5.2. Start Building

Assuming you are working from the template component, perform the following steps to start building your integration:

1. Create a queue message_destination in your Resilient Organization and add your integrations account as an authorized user of that message destination.
2. In the __init__ method of your component class, set the “channel” member to “actions.<message_destination>” with <message_destination> being swapped out for the programmatic name of the queue you just created.
3. Create a new rule in the Resilient platform, either Automatic or Menu Item, and add your queue as a message destination for that rule.
4. Rename the _framework_function method of your class to something descriptive for what the action should do, and then update the @handler decorator above it to match the programmatic name of the rule you created in the Resilient platform. For example, if you created a new rule called “Do Something” then your decorator should look like: @handler(“do_something”)
5. Add a section to your app.config file with a name that is reflective of your component to store configuration values, such as [do_something]. Put any configuration values you need here.
6. Update the CONFIG_DATA_SECTION variable in your module with the name of the section you created. For example: CONFIG_DATA_SECTION = “do_something”
7. 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. There are many examples of API usage on GitHub. 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 "Task added successfully"
   ```

8. Run your integration with “resilient-circuits run” and your component should be loaded and run whenever your rule is triggered.

5.3. Run During Development

During development, it would be very 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, etc) so you can use tools like a debugger.

In lieu of the “resilient-circuits run” 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 haven’t 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.

5.4. Add Functionality with Decorators

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

5.4.1. required_field

This 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, then 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"
```

5.4.2. 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.

5.4.3. 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 only works 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 don't 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"
```

5.4.4. debounce

There are times when an action handler is likely to be triggered multiple times in quick succession, but you don't want to handle the events until they are all 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 only works with handlers that specify the action they are handling. Methods that are being used as a default handler, with @handler(), can't 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"
```

5.5. 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 up the handler to take care of the next incoming event, you can user 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 acks 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"
```

5.6. Web UI and Restful Components

The 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.

Documentation for circuits.web can be found here.

5.6.1. Build a Web Component

The first step in building a web component for Resilient Circuits is to install the rc-webserver package, which can be downloaded from GitHub. 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.

An example is available from Simple Web Example.

6. Package a Resilient Circuits 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
```

Your setup.py file should look very similar to the one in taskadd example on GitHub mentioned previously, so you can use it as an example. Swap out “taskadd” for the name of your integration. 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, you can share it with other Resilient users on the Resilient community examples GitHub repo.

.

7. Test a Resilient Circuits 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.

7.1. 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".

```
bash-3.2$ ls logged_responses/ActionMessage*
logged_responses/ActionMessage_AddTask_2017-03-07T09:24:41.822231
```

Now, 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 example below, 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.

```
bash-3.2$ 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}
```

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>".

7.2. 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.

The pytest plugin can be downloaded from the Resilient GitHub repository here and installed with:

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

7.2.1. 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 the Run Tests section.
    - 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.

The task_add example mentioned previously has a single test included that shows usage of each of these fixtures.

7.2.2. 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 as shown in the task_add example, 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 don’t 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
```

7.3. 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 run 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. The example in the task_add project is a good starting point. 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 “co3”, “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 now 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
```

7.4. Mock Resilient API

It is not always practical or possible to run tests against a live Resilient instance. The Co3 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 co3 and resilient-circuits packages use them.

The add_task example contains a very basic mock (click here for the example) that returns just the data logged from a saved session for the minimum number of endpoints to support the AddTask component. A mock to support actual tests would need to alter the responses based on the contents of POST/PUT requests.

Create a class derived from co3.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 co3.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)
```

8. Run as a Service on Windows

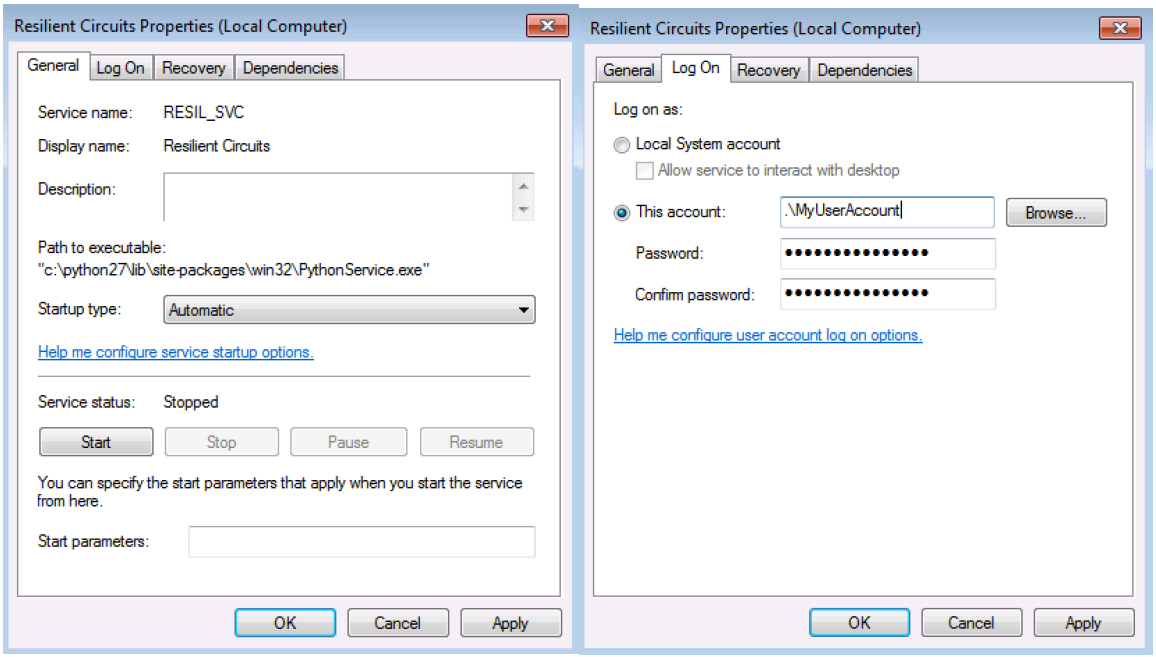
Resilient Circuits can be configured to run as a service. It requires the pywin32 library, which should be downloaded from sourceforge. Instructions for downloading and installing the correct package are at the bottom of the screen and must be followed carefully. Do not use the pypi/pip version of pywin32.

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
```

9. Run as a Service on Linux

Resilient Circuits can be configured to run as a service on Linux with supervisord.

If you do not have supervisord on your Debian Linux platform, you can download it using the following command.

```
sudo apt-get install supervisor
```

If you had supervisord on your platform, make sure you have the latest version:

```
sudo apt-get update
```

Install supervisord:

```
sudo apt-get install supervisor
```

Locate the supervisord configuration file then review and edit as necessary. The configuration file defines the following properties:

- A name to identify the program for supervisord.
- 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
- Location for the logfile.

Here is an example of a configuration file:

```
[program:resilient_circuits]
user=integration
directory=/usr/share/integration/
environment=LANG=en_US.UTF-8,LC_ALL=en_US.UTF-8
command=resilient-circuits run
stdout_logfile=/var/log/resilient_circuits.log
redirect_stderr=true
autorestart=true
```

You may need to add additional environment variables to the configuration file, such as APP_CONFIG_FILE.

The program to run is defined in the configuration file. Copy this to the configuration directory and restart the service:

```
sudo cp actions_supervisor.conf /etc/supervisor/conf.d/
sudo service supervisor restart
```

The supervisor service logs its activity to /var/log/supervisor/supervisord.log.

To restart the supervisor service, use:

```
sudo service supervisor restart
```