IBM Resilient

resilient

Incident Response Platform

ACTION MODULE PROGRAMMER'S GUIDE v28

Resilient Incident Response Platform Action Module Programmer's Guide

Platform Version	Publication	Notes
28.0	May 2017	Supports the V28 Resilient platform.
27.1	February 2017	Change to logo. No technical changes.
27.0	December 2016	Removed action-install.py.
27.0	December 2016	Reflect changes to the V27 Resilient platform.
26.0	June 2016	Updated the company logo. No technical changes.
24.0	January 2016	Added pointer to action processor components for Python under *Using Framework*.
24.0	November 2015	Updated with the V24 release of the Resilient platform.
23.0	June 2015	Initial publication

Table of Contents

1. Objective

This document is intended for programmers, testers, architects and technical managers interested in developing and testing integrations with the Resilient Incident Response Platform using the Resilient Action Module. It assumes a general understanding of the Resilient platform and message-oriented middleware (MOM) systems.

2. Introduction

The Action Module is an available extension point to the Resilient platform that enables organizations to define and implement custom behaviors in response to user-defined conditions arising within the Resilient platform. The module is built on an embedded version of Apache ActiveMQ.

The following sections describe rules, which are created on the Resilient platform, and action processors. This document contains the information needed to design, implement and deploy action processors.

2.1. Rules

From the Rules tab in the Resilient platform, master administrators can define the conditions that trigger rules and the activities which are executed when a condition is met. The activities include setting incident field values, inserting tasks into the task list, invoking workflows, running internal scripts to implement business logic, and placing items on message destinations to be acted upon by remote action processors.

There are two types of rules, automatic and menu item. An automatic rule executes without user involvement when its conditions are satisfied. A menu item rule displays in the Resilient platform as an action in the object’s Actions drop-down list and only executes when a user invokes it.

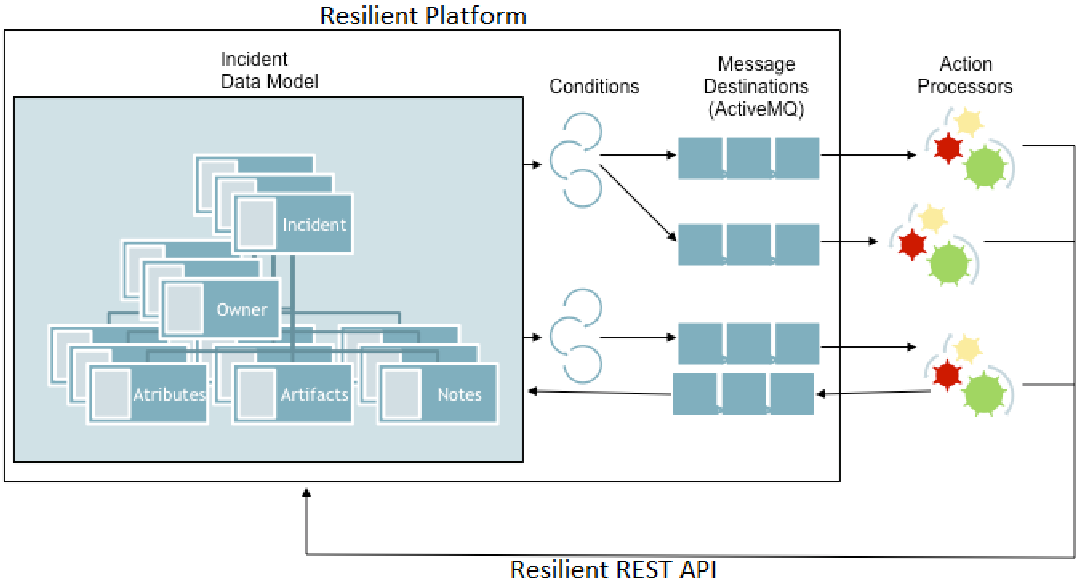
When a rule executes, it can optionally send a message to a *message destination*, which is the location where the message is stored and made accessible to remote action processors. The message includes details about the object and the activity taken. You can configure rules to send messages to one or more message destinations. In addition, you can configure workflows to send messages to one or more message destinations

The Action Module is required for the workflow, script and message destinations functions to be enabled on the Resilient platform. For more information, refer to the *Resilient Incident Response Platform Master Administrator Guide*.

2.2. Action Processors

Action processors allow you to integrate the Resilient platform with existing security systems to create a single hub for incident response.

An action processor can be configured to access specific message destinations in the Resilient platform, perform an operation (possibly using the Resilient REST API) using the information then optionally send a reply back to the platform. The following figure shows the Action Module within the Resilient platform architecture.



Action processors can implement custom business logic and interact with external systems. The following are some example use cases:

- User wants to be able to create a note (e.g., "We need to collect the logs from server xyz") and then be able to generate a ticket in the IT helpdesk. If the rule requires that additional parameters need to be collected (e.g., BU, charge code, contact email), the rule can provide a dialog prompt.

- Customer has an internal asset DB. Every time an IP artifact is added, they want a rule to be invoked automatically to check that IP against their asset DB and add information to the artifact.

- The customer has specific logic they want to use to set a field value, such as severity based on built-in and custom fields. Every time an incident is created or modified, they want to execute their custom logic and then set the value accordingly.

- There is a task in the IR plan that says, "You should disable any compromised user accounts," and we would like to include a "Disable AD Account" button to perform that action. When the user clicks the button, it prompts the user to enter the user ID and then it dispatches the rule that connects to the local AD and disables the account.

- Customer uses Splunk or other SIEM. Every time a DNS name or IP address artifact is added, it automatically performs a SIEM search to find any relevant log entries then add the search results as an incident/task note or amend the artifact description.

- When adding a malware sample as an artifact, send the sample to a sandbox virtual machine to analyze. Include the resulting report in the artifact description or as a note.

3. Design Considerations

This section contains information about design considerations for action processors.

3.1. Choosing a Programming Language and Messaging Protocol

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

The language you choose depends on many factors, including language familiarity. One additional factor to consider is the maturity of the client library support. Although most modern languages have messaging support (generally through libraries implementing the STOMP protocol), Java-based languages (e.g., Java, Groovy, Scala) are generally considered to have the most mature messaging support.

If you use a Java-based language, typically you would use the ActiveMQ client library, which uses the OpenWire protocol. There are libraries that support STOMP and are available for most modern programming languages. The following web page includes many different STOMP client library options: http://stomp.github.io/implementations.html#STOMP_Clients

The examples in this guide are written either in Groovy (which is Java-based) or Python.

3.2. User Authentication/Authorization

The action processors authenticate to the message broker using Resilient credentials. It is recommended that you create dedicated service accounts for this purpose. These accounts can be created with very strong passwords.

Accounts can be created from the Resilient command line using the following commands:

```
$ openssl rand -hex 32
<SOME RANDOME HEX STRING>
$ sudo resutil newuser -email security@mycompany.com -org "My Company" -
first Security -last User
[sudo] password for resilient_admin: <enter Resilient_admin password>
Enter the password for the user: <random hex from above>
Confirm the password for the user: <random hex from above>
```

When prompted for the new user password, enter the random value from the openssl rand command. Because you use sudo to invoke the resutil tool, you may be prompted to enter your current login user password first.

The openssl command generates a random password of 32 bytes of data encoded as a hex string (which results in 64 characters). The second command adds a user using the random password from the previous command.

The users created by the resutil command are administrative users. This may or may not be required by your application. If it is not required, you can use the Resilient platform to remove the administrative rights before you use it to connect for the first time.

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

Once created, you can grant the user access to the message destinations to which it needs access. This is done through the Resilient platform (Customization Settings > Message Destinations). You grant access to message destinations by adding the allowed users in the Users section. Refer to the *Resilient Incident Response Platform Master Administrator Guide* for details.

3.3. Queue vs. Topic

Resilient Message Destinations can be configured to be either topics or queues. The following table describes the differences between topics and queues in the message broker context.

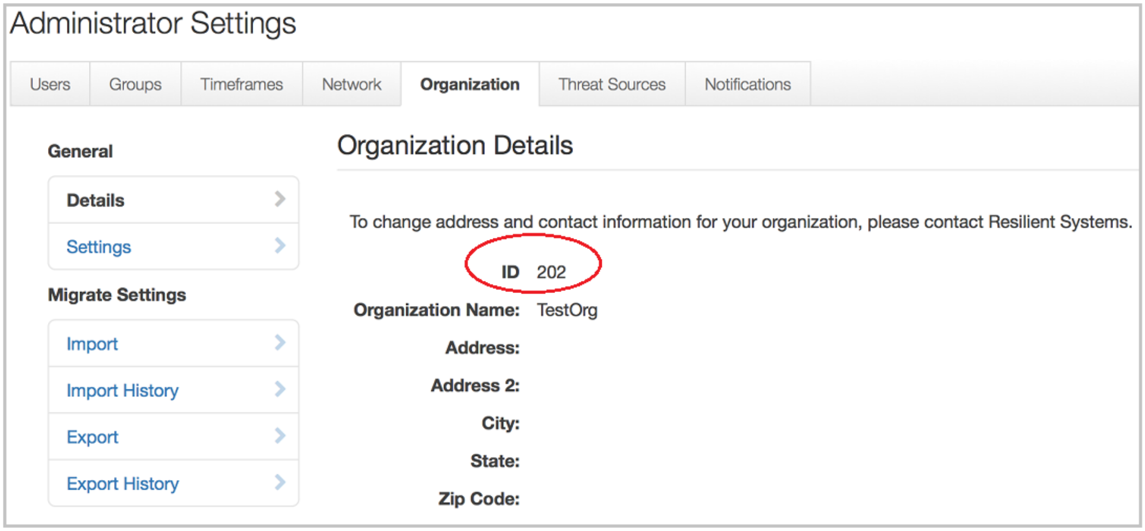
Type	Who receives messages?	No active subscribers?
Queue	One subscriber	Message is stored for later consumption
Topic	All *active* subscribers	Message is dropped

Based on the nature of Resilient integrations, we expect most message destinations to be created as queues. This ensures that messages are not dropped because services are down.

3.4. Message Destination Naming

When you create a message destination, you give it a display name. The system automatically generates a programmatic name that is used when your action processors connect to it. The display name can contain any character. The programmatic name can contain only alphanumeric characters and underscores.

Please note that when you connect to the destination from your code, you have to specify a prefix that includes the organization's numeric ID. This ID is specified in Administrator Settings > Organization. The following figure illustrates how to locate your organization ID.



Administrator Settings

Organization Details

To change address and contact information for your organization, please contact Resilient Systems.

ID 202

Organization Name: TestOrg

Address:

Address 2:

City:

State:

Zip Code:

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

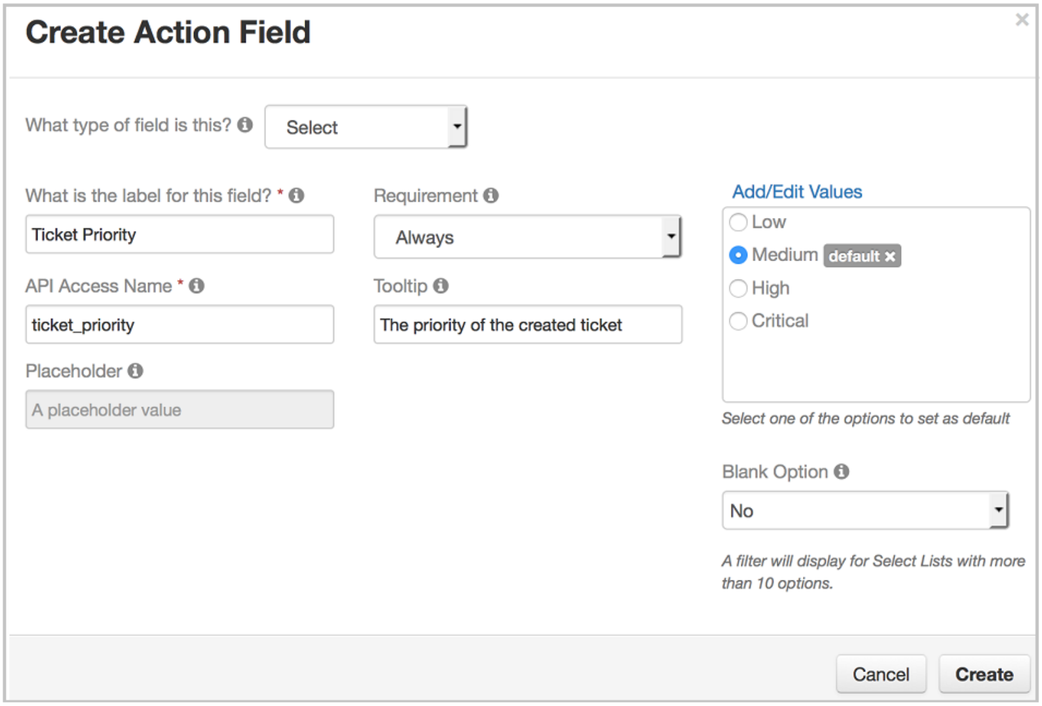
NOTE: Some client libraries have you connect to the destination using a "/queue/" or "/topic" prefix. For example, if you were connecting to the ticket queue, you would use a name of "/queue/actions.202.ticket". Consult the documentation for your client library for more information.

3.5. Menu Item Rules and Activity Fields

As described earlier, there are two types of rules, automatic and menu item.

The menu item rule displays as an action in the object's **Actions** drop-down menu and executes only when a user invokes it. In some cases, it is necessary for the user to enter additional information when selecting an action. For example, if you are developing a "Create ticket" action, you may need to allow the user to select a priority for the ticket that is to be created. You do this by creating an activity field. Activity fields are managed through the Resilient platform as part of the menu item rule. Refer to the *Resilient Incident Response Platform Master Administrator Guide* for details.

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



Create Action Field

What type of field is this? ℹ Select

What is the label for this field? * ℹ Ticket Priority

Requirement ℹ Always

Add/Edit Values

○ Low
● Medium default ×
○ High
○ Critical

Select one of the options to set as default

API Access Name * ℹ ticket_priority

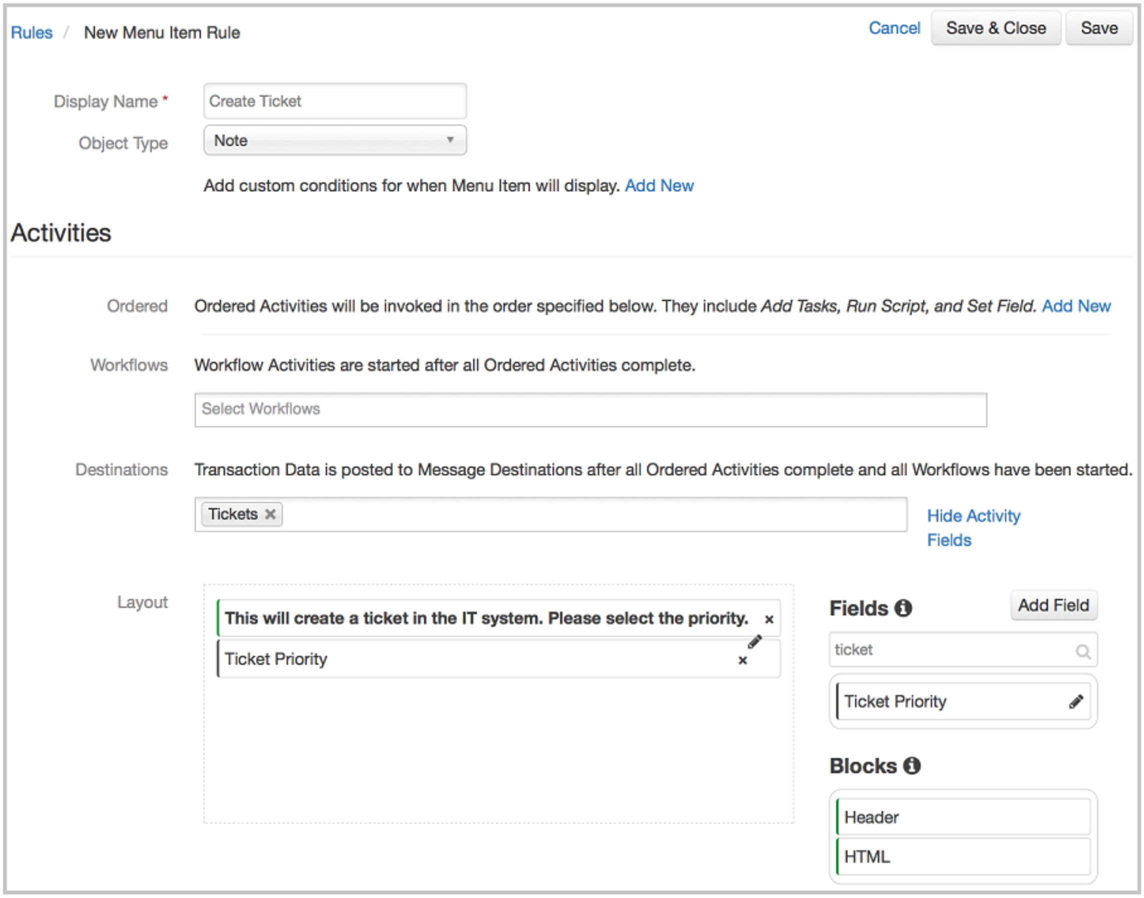
Tooltip ℹ The priority of the created ticket

Placeholder ℹ A placeholder value

Blank Option ℹ No

A filter will display for Select Lists with more than 10 options.

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



Display Name * Create Ticket

Object Type Note

Add custom conditions for when Menu Item will display. Add New

Activities

Ordered: Ordered Activities will be invoked in the order specified below. They include *Add Tasks, Run Script, and Set Field.* Add New

Workflows: Workflow Activities are started after all Ordered Activities complete.

Select Workflows

Destinations: Transaction Data is posted to Message Destinations after all Ordered Activities complete and all Workflows have been started.

Tickets ✕

Hide Activity Fields

Layout:

This will create a ticket in the IT system. Please select the priority.

Ticket Priority

3.6. Action Data

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

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

Field Name	Description
action_id	ID of the rule that caused the message.
type_id	Type of object that caused the message. The types and their associated IDs are available through the REST API with the "/rest/orgs/{orgId}/types" endpoint.
incident	Incident object to which the invocation applies. Note that this value will be set for items that are subordinate to incidents. It is currently the case that all messages will contain an incident.
task	Task to which the invocation applies (if any). Note that this value will be set for items that are subordinate to tasks, such as task notes and task attachments.
artifact	Artifact to which the invocation applies (if any).
note	Note to which the invocation applies (if any).
milestone	Milestone to which the invocation applies (if any).
attachment	Attachment to which the invocation applies (if any).
type_info	Contains information about types/fields that are referenced by the other data. See the Action Data and Type Information section for more information.
properties	Contains the field values the user selected when invoking a menu item rule (if any).
user	Contains information about the user that invoked the action.

3.7. Action Data and Type Information

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

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

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

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

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

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

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

3.8. Message Headers

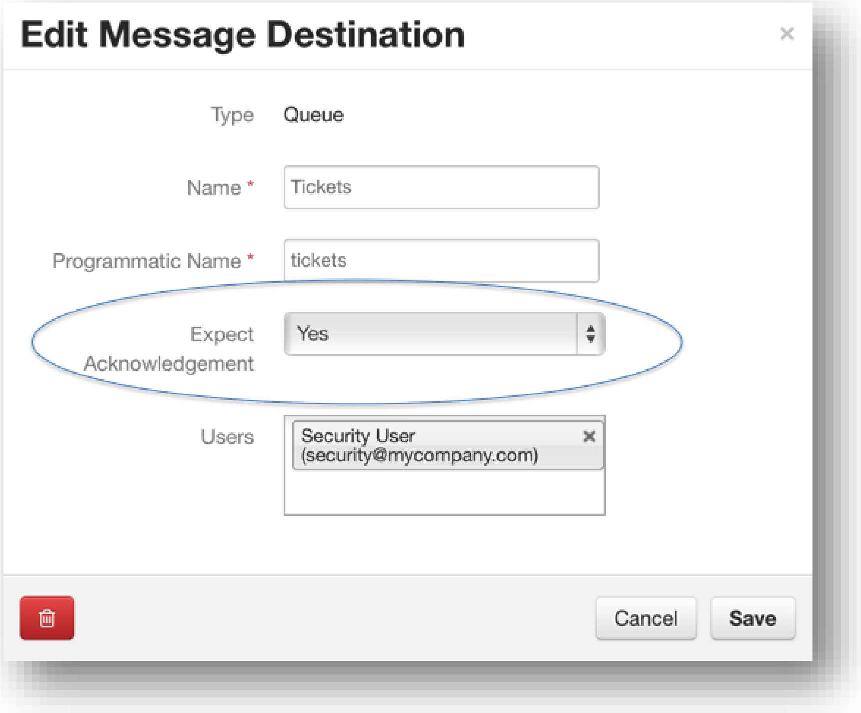
The Resilient platform includes various message headers that are needed (or in some cases just helpful) in processing messages.

Header Name	Request/Reply	Description
Co3ContextToken	Request	A token value that must be specified if the action processor calls back into the Resilient REST API. The primary purpose of this token is to ensure that actions processing does not result in an infinite loop. See the Using the Resilient REST API with Action Processors section for additional information.
correlation-id	Request and Reply	Identifies the rule invocation to which this message applies. It must be included in acknowledgement messages sent back to the Resilient platform. See the Acknowledgements section for additional information. If you are using a JMS client, this value can be retrieved with the getJMSCorrelationID method.
reply-to	Request	Identifies a server-controlled message queue that must be used when acknowledging (replying to) this message. See the Acknowledgements section for additional information. If using a JMS client, this value can be retrieved with the getJMSReplyTo method.

Header Name	Request/Reply	Description
Co3InvocationComplete	Reply	A boolean header that tells the Resilient platform whether processing is complete. The default is true, so you only need to include it if you are sending an informational message and it is not complete. This header is ignored if the reply message is JSON.

3.9. Acknowledgements

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



Edit Message Destination

Type: Queue

Name *: Tickets

Programmatic Name *: tickets

Expect Acknowledgement: Yes

Users: Security User (security@mycompany.com)

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

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

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

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

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

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

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

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

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

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

3.10. TLS

Action processors must connect to the Resilient message broker using TLS v1.0 or higher. To ensure the security of the connection, action processors must properly validate the server certificate. The exact mechanisms for doing this varies by programming environment and is beyond the scope of this document. However, the following must be considered:

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

Some of the common JMS libraries for Java do not perform checking on the certificate name (#4 above). We have developed a workaround for this, which is used in the Java examples.

3.11. Using the Resilient REST API with Action Processors

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

The only restriction is that when making REST API requests you must specify the X-Co3ContextToken HTTP header. The value to specify in this header is passed as the Co3ContextToken message header.

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

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

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

    inc_id = message_obj['incident']['id']
    url = "/incidents/{}/comments".format(inc_id)
    comment_data = {'text': 'Some comment for the incident'}

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

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

3.12. HTTP Conflict (409) Errors

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

The Resilient examples have accounted for this issue where necessary.

3.13. Using a Framework

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

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

IBM Resilient provides a set of action processor components for Python, built with the Circuits framework. Refer to the main API project on GitHub (Co3 / resilient-api) for further details.

Product	Language	Description
Apache Camel **http://camel.apache.org/**	Java	An open source framework for creating processing routes. For example, a route might: • Read a message from a queue (Resilient message destination) • Convert the message payload from a JSON string to an object • Invoke an HTTP POST on some external service • Send a reply back to the Resilient platform Most of this can be done through XML- or DSL-based configurations. There is an example of how you can use Apache Camel in the Resilient API examples distribution.
Apache ServiceMix ESB **http://servicemix.apache.org/**	Java	ServiceMix is an OSGi-based Enterprise Service Bus (ESB). ServiceMix can work seamlessly with Apache Camel to simplify route creation.
Mulesoft ESB **http://www.mulesoft.com**	Java	A commercial Enterprise Service Bus (ESB) that allows you to create graphically action processors using a number of built-in connectors. There is an example of how you can use Mulesoft in the Resilient API examples distribution.
Spring Integration **http://projects.spring.io/spring-integration/**	Java	Extends the Spring programming model to support Enterprise Integration Patterns.
Zato ESB **https://zato.io**	Python	An open source Python-based Enterprise Service Bus (ESB).

3.14. Always Running

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

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

If using Python, you might consider using the python-daemon library (Python 2) or python-daemon-3K (Python 3).

If using Java, you might consider using the Apache Commons Daemon project (http://commons.apache.org/proper/commons-daemon/).

3.15. Retry

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

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

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

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

3.16. Processor Installation

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

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

4. Testing Considerations

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

This section contains test cases that serve as a starting point for testing an action processor.

4.1. TLS

The action processors connect to the Resilient platform using TLS. This applies to both the connection to the message destination (STOMP over TLS and Active MQ/OpenWire over TLS) and the Resilient REST API (HTTPS).

The Design Considerations TLS section discusses this in detail. This section includes some guidance on how you can test that TLS connections are properly established.

4.1.1. Certificate Trust

When you invoke the action processor, you must confirm that a "man-in-the-middle certificate attack" causes an error and that no data is sent over the connection. The second part of that statement is important because it would be possible for bad code to establish a connection, send passwords *then* check for certificate trust. Sending the password over an untrusted channel would be a security vulnerability.

The simplest way to test this is to configure the client (action processor in this case) to NOT trust the Resilient platform certificate and confirm that the operation fails due to a TLS error.

4.1.2. Certificate Common Name

If an action processor thinks it is connecting to a host named "Resilient.mycompany.com" then it is important that the TLS certificate is issued to "Resilient.mycompany.com". If not and you proceed sending data, it is possible for a man-in-the-middle to present a certificate that was issued by a trusted source, but issued to a different entity (e.g., maybe it was issued to www.someothercompany.com). The accepted best practice for a TLS client to guard against this attack is to check that the certificate's common name (or subjectAltName) matches the host to which the connection is being made.

The simplest way to test this is to change your local hosts file to make "testhost" point to the Resilient platform's IP address, and then try to connect using testhost. The connection should fail. Note that this should be performed against both the Resilient REST API (port 443) and the Action Module server (port 65000 and/or 65001).

4.2. Retry/Error Reporting

It is important for the action processor to continue operating in the face of exceptions. The following should be tested:

- Does the action processor have a log file?
- Does the action processor report errors from external systems?
- Does the action processor continue running when the Resilient platform is down?

4.3. Always Running

The action processor should generally be running.

- Does the action processor start automatically when the host on which it runs is restart?
- Does the action processor process survive a user log out?

4.4. Conflicting Edits

If the action processor updates the Resilient platform using the REST API, it must be written to handle situations where another user edits the same object.

- Has this situation been accounted for by the developer?
- If you invoke a rule when the action processor is stopped, then make another change to the object (say an incident), then start the action processor. Does the action processor properly update the incident? Note: If it is not handled by the developer, then you would likely get an error when the processor attempts to do the PUT operation.

4.5. Action Status Sent

Does the action processor send a status or error message to the Resilient platform as appropriate? These status messages appear in the Actions > Action Status dialog.

4.6. Infinite Loops

Is it possible for the action processor to get into an infinite loop? See the Co3ContextToken discussion in the Message Headers section.

4.7. Others?

If there are other elements of the action processor that you think could be mentioned here, we'd love your feedback! Please contact us at success@resilientsystems.com.

5. Customer Success

If you need support in creating action processors, please contact someone in our Customer Success group at success@resilientsystems.com.