Co3 Custom Action Framework

Programmer's Guide v20.0

Co3 Systems

ABSTRACT

This document is intended for programmers, testers, architects and technical managers interested in developing and testing integrations with the Co3 application using the Co3 Custom Action Framework (CAF). It assumes a general understanding of the Co3 application and message-oriented middleware (MOM) systems.

Introduction

CAF is an available extension point to the Co3 platform that enables organizations to define and implement custom behaviors in response to user-defined conditions arising within the Co3 system.

Actions can be configured within Co3 as either automatic or manual. Automatic actions are executed automatically without user interaction when an event occurs meeting the pre-set conditions defined for that action. Manual actions require user invocation by clicking a button that appears in the Co3 user interface.

CAF is built on an embedded version of Apache ActiveMQ. When the incident data model changes match an action's pre-set conditions or the user manually invokes the action, a message describing the event is added to a message destination. An action processor then reads this message, performs some operation (possibly using the Co3 REST API) then optionally sends a reply back to the server.

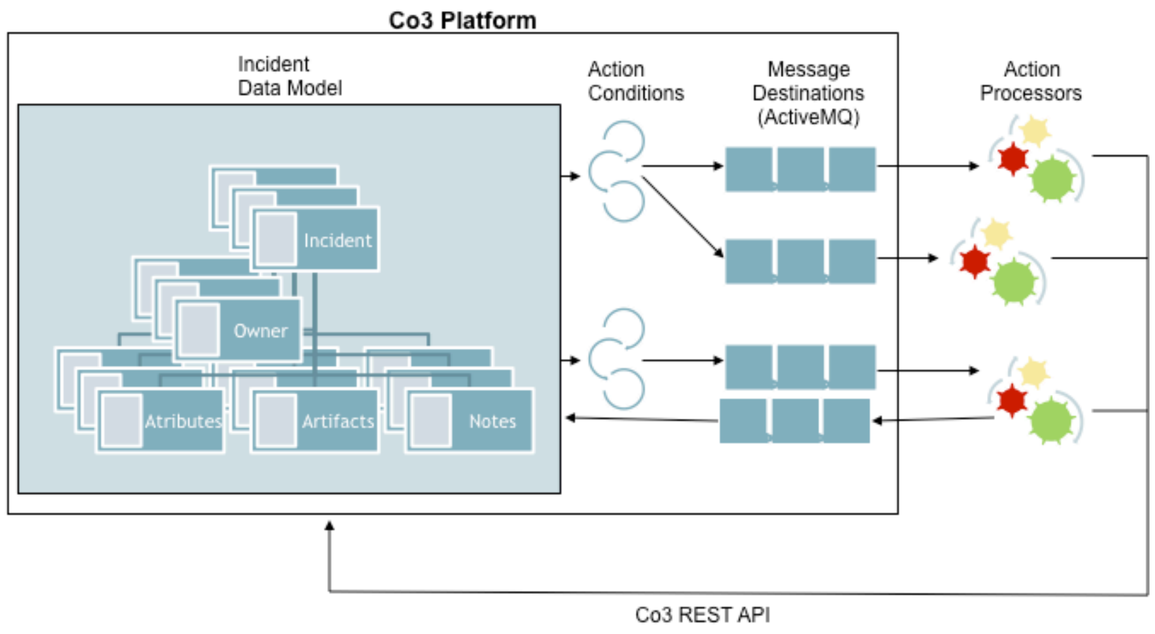


Figure 1 CAF in context

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 invoke an action on that note to generate a ticket in the IT helpdesk. It may be that the action will require additional

parameters that will need to be collected (e.g. BU, charge code, contact email) through a dialog prompt, which is supported through action layouts and fields.
- Customer has an internal asset DB. Every time an IP artifact is added they want a custom action to be automatically invoked 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'd like to include a button "Disable AD Account" to invoke to perform that action. When the user hits the button it prompts them to enter the user ID and then it dispatches the custom action that will connect to the local AD and disable the account.
- Customer uses Splunk or some other SIEM. Every time a DNS name or IP address artifact is added, automatically perform a SIEM search to find any relevant log entries. Add search results as incident/task note or amend the artifact description.
- When a malware sample is added as an artifact, send the malware sample to a sandbox virtual machine to analyze. Include the resulting report in the artifact description or as a note.

This document contains information needed to design, implement and deploy CAF processors.

Design Considerations

This section contains information about design considerations for CAF processors, including:

Choosing a Programming Language and Messaging Protocol
User Authentication/Authorization
Queue vs. Topic
Message Destination Naming
Manual vs. Automatic Actions
Action Fields and Views
Action Data
Action Data and Type Information
Message Headers
Acknowledgements
TLS
Using the Co3 REST API with Action Processors
HTTP Conflict (409) Errors
Using a Framework

Always Running
Retry
Processor Installation

Choosing a Programming Language and Messaging Protocol

CAF 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 will depend 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 you will generally use the ActiveMQ client library, which uses the OpenWire protocol. There are libraries that support STOMP available for most modern programming languages. The following page includes many different STOMP client library options:

http://stomp.github.io/implementations.html#STOMP_Clients

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

User Authentication/Authorization

CAF processors authenticate to the message broker using Co3 credentials. Co3 recommends that you create dedicated service accounts for this purpose. These accounts can be created with very strong passwords.

Accounts can be created from the Co3 appliance command line using the following commands:

```
$ openssl rand -hex 32
<SOME RANDOME HEX STRING>
$ sudo co3util newuser -email security@mycompany.com -org
"My Company" -first Security -last User
[sudo] password for co3admin: <enter co3admin password>
Enter the password for the user: <random hex from above>
Confirm the password for the user: <random hex from above>
```

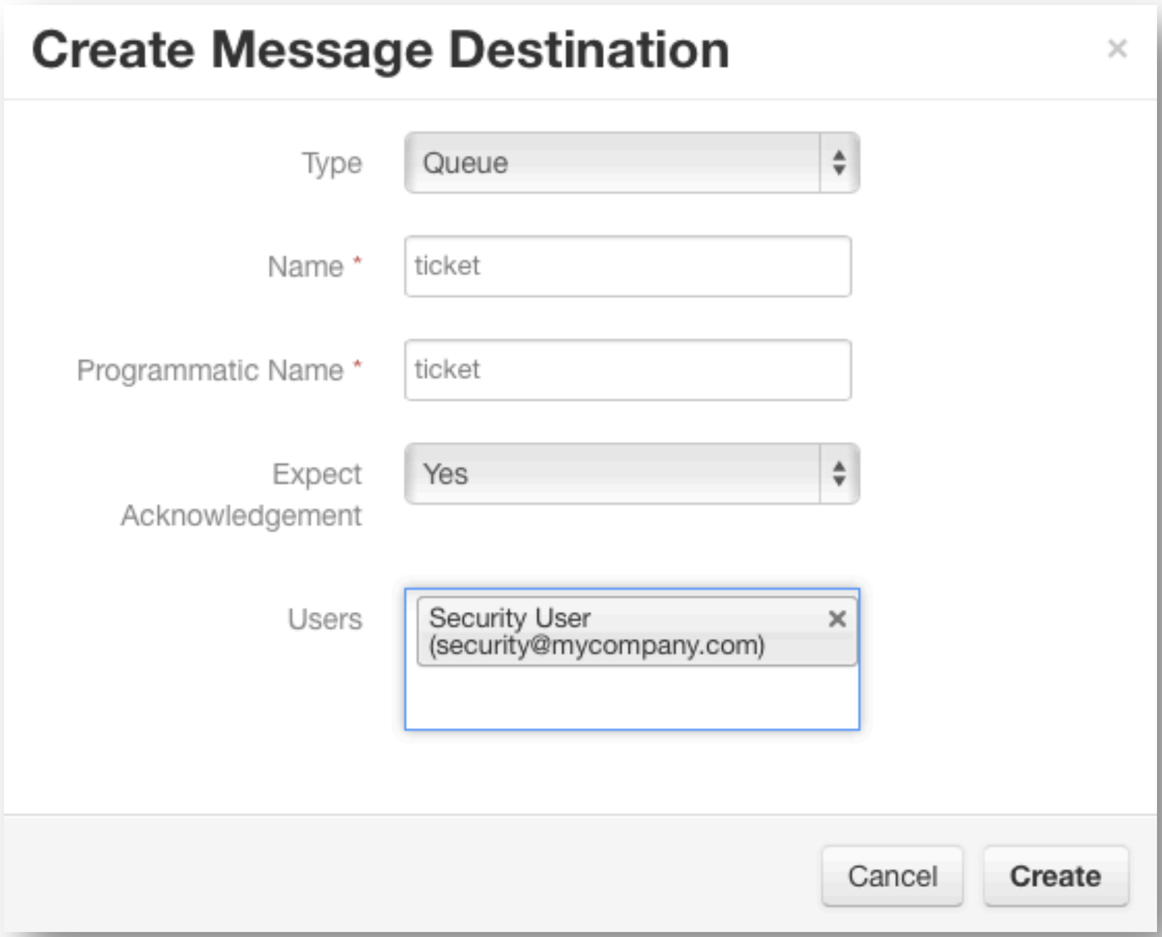
When prompted for the new user password, enter the random value from the “openssl rand” command. Note that because you use sudo to invoke the co3util tool, you may or may not be prompted to enter your current login user password first.

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 co3util command will be administrative users. This may or may not be required by your application. If it is not required you can use the Co3 web application to remove the administrative rights before you use it to connect for the first time.

Some CAF processors need to use the Co3 REST API to access or modify additional Co3 data. This same user account can be used to authenticate with the REST API.

Once the user is created, you can grant it access to the message destinations to which it needs access. This is done through the Co3 web application (Administrator Settings > Actions > Message Destinations). You grant access to message destinations by adding the allowed users in the Users section.


Create Message Destination

Type: Queue

Name *: ticket

Programmatic Name *: ticket

Expect Acknowledgement: Yes

Users: Security User (security@mycompany.com)

Figure 2 Create Message Destination User Authorization

Queue vs. Topic

Co3 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

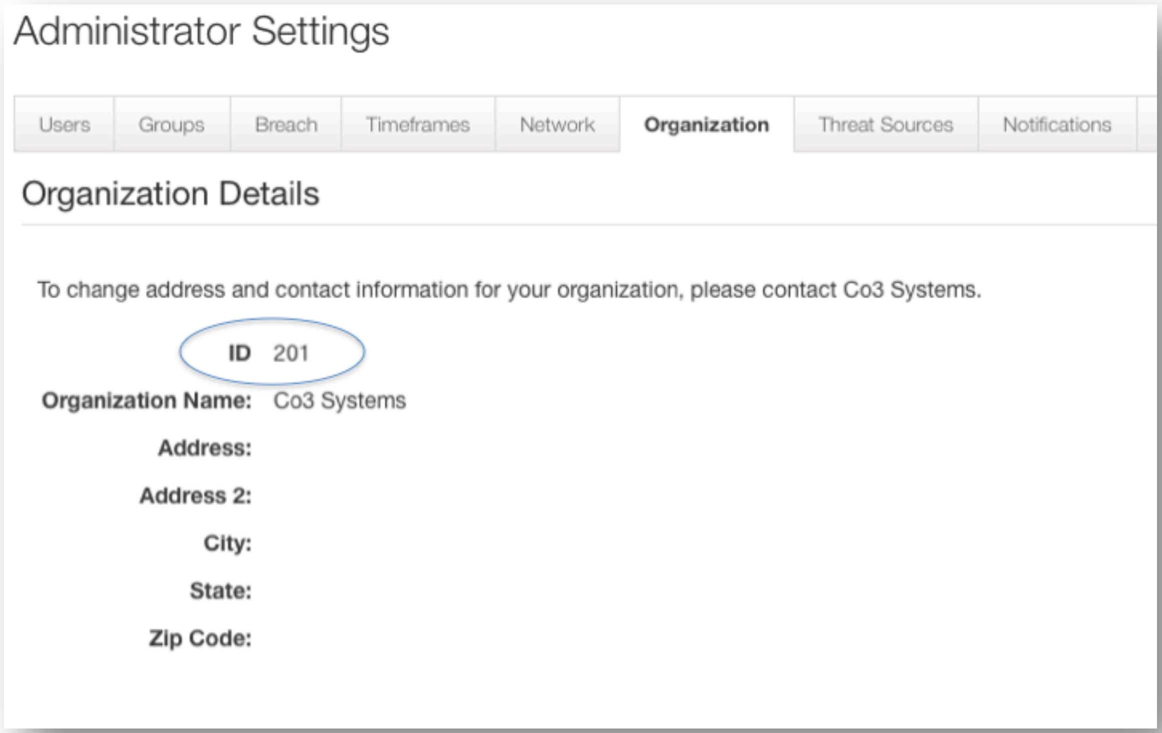
Table 1 Queues vs. Topics

Based on the nature of Co3 integrations, we expect most message destinations to be created as queues. This ensures that messages do not get dropped because services are down.

Message Destination Naming

When you create a Message Destination in Co3 you give it a display name. The system will automatically generate a programmatic name that will be used when your action processors connect to it. The display name can contain any characters. The programmatic name can only contain 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. Figure 3 illustrates how to locate your organization's ID.



Organization Details

To change address and contact information for your organization, please contact Co3 Systems.

ID 201
Organization Name: Co3 Systems
Address:
Address 2:
City:
State:
Zip Code:

Figure 3 Locating the organization ID

If you created a message destination in the Co3 web UI with a programmatic name of "ticket" and your organization ID is 201 as it is in Figure 3, the name you would use in your CAF processor code to read messages would be "actions.201.ticket"[1].

Manual vs. Automatic Actions

As mentioned earlier, Co3 supports two different types of actions: automatic and manual.

Automatic actions are automatically triggered by some pre-configured condition being met. When you define an automatic action you are really just describing the conditions under which a message will be placed on a message destination, which will then be consumed by an action processor to perform the desired processing. Automatic actions are managed through the Co3 web application (Administrator Settings > Actions > Automatic Actions).

Manual actions are those that are executed as a result of a user explicitly invoking them. Configuring a manual action results in an action option being displayed in an

[1] Some client libraries will have you connect to the destination using a "/queue/" or "/topic" prefix. For example, if you are connecting to the "ticket" queue, you would use a name of "/queue/actions.201.ticket". Consult the documentation for your client library for more information.

action menu in the appropriate place in the Co3 user interface. For example, if you configure a manual action on notes for escalating the note to an external ticketing system, then when users are viewing notes they will see an option on the note's action menu to invoke that action. Like automatic actions, invoking a manual action results in a message being placed on a message destination, which will then be consumed by an action processor to perform the desired processing.   Manual actions are managed through the Co3 web application (Administrator Settings > Actions > Manual Actions).

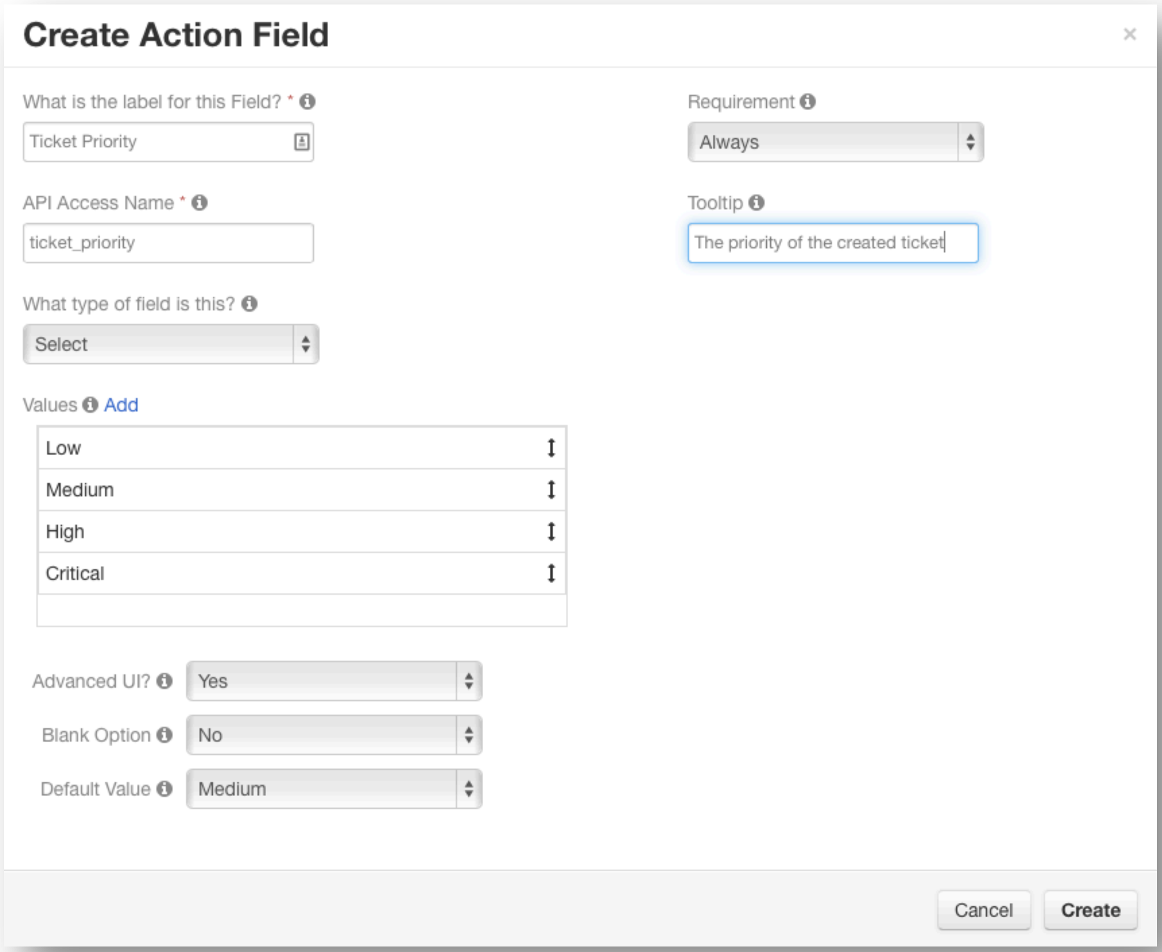
Both manual and automatic actions can be configured with conditions, however the context of the condition evaluation is different between the two. For automatic actions, the conditions dictate when the action results in messages being added to the associated message destinations. For manual actions, the conditions dictate when the action will be enabled in the user interface for a user.

If you do not add any conditions when configuring the action, the action will always apply. For automatic actions, this means that a message will be added to the message destinations whenever the object type is modified or created. For manual actions, this means that the action menu will always be enabled in the UI.

Action Fields and Views

In some cases it is necessary for the user to enter additional information when a manual action is invoked. For example, if you're 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 action field. Action fields are managed through the Co3 web application (Administrator Settings > Actions > Action Fields).

Figure 4 illustrates the creation of a Ticket Priority field that will later be added to a manual action layout. It will be a select field that has 4 values (Low, Medium, High and Critical). The field value will be required.


Create Action Field

What is the label for this Field? *: Ticket Priority

Requirement: Always

API Access Name *: ticket_priority

Tooltip: The priority of the created ticket

What type of field is this?: Select

Values Add

Low
Medium
High
Critical

Advanced UI?: Yes

Blank Option: No

Default Value: Medium

Figure 4 Adding an action field

After the field has been created, it can be added to one or more manual actions. Figure 5 shows how you add the field to the manual action's layout.

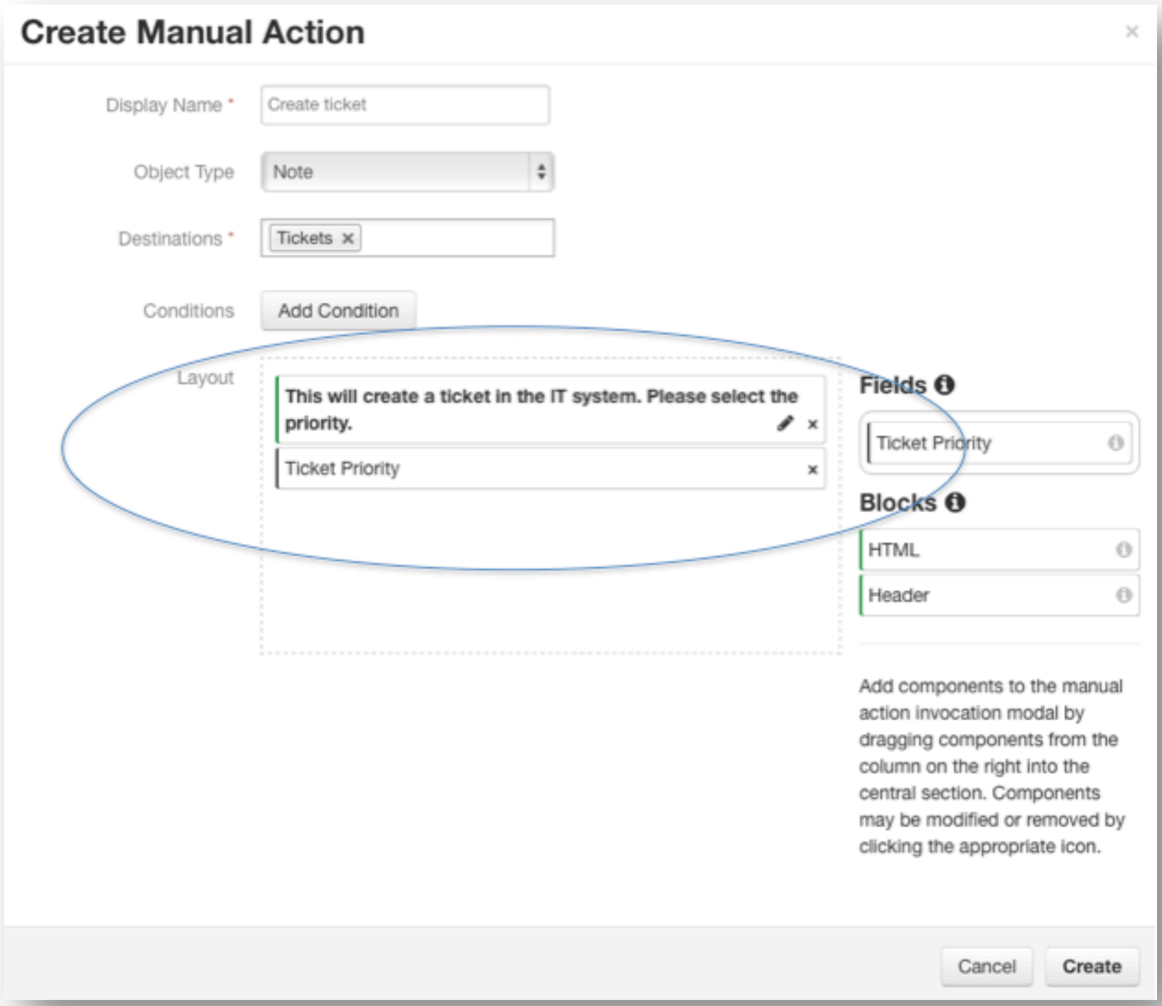

Create Manual Action
Display Name *
Create ticket
Object Type
Note
Destinations *
Tickets
Conditions
Add Condition
Layout
This will create a ticket in the IT system. Please select the priority.
Ticket Priority
Fields
Ticket Priority
Blocks
HTML
Header
Add components to the manual action invocation modal by dragging components from the column on the right into the central section. Components may be modified or removed by clicking the appropriate icon.
Cancel
Create

Figure 5 Creating a manual action with action fields

The layout can also contain a “header” that gives the user some additional information, as shown in Figure 5.

Action Data

Messages contain JSON data. The structure of the JSON data is described in the Co3 REST API documentation in the ActionDataDTO type. This structure contains much of the data that you will need to implement your action processors. However, if there is additional Co3 data that you require you can access it using the Co3 REST API (see the Using the Co3 REST API section for considerations when doing this).

Table 2 describes the top-level properties in the ActionDataDTO type. For specific information about this type consult the Co3 REST API documentation.

Field Name	Description

action_id	The ID of the action that caused the message.
type_id	The 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	The 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	The 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	The artifact to which the invocation applies (if any).
note	The note to which the invocation applies (if any).
milestone	The milestone to which the invocation applies (if any).
attachment	The attachment to which the invocation applies (if any).
type_info	This contains information about types/fields that are referenced by the other data. See the Action Data and Type Information section for more information.
properties	This contains the field values the user selected when invoking a manual action (if any).
user	This contains information about the user that invoked the action.

Table 2 ActionDataDTO top-level objects

Action Data and Type Information

The data specified in the incident, task, artifact, note, milestone and attachment fields will generally only contain 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 will contain an integer (the severity ID). If your processor needs the severity text that was actually selected you can get it from the type_info field.

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

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

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

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

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

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

Message Headers

Co3 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 Co3 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 Co3 REST API with Action Processors section for additional information
correlation-id	Request and Reply	Identifies the action invocation to which this message applies. It must be included in acknowledgement messages sent back to the Co3 server. 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 you are using a JMS client, this value can be retrieved with the getJMSReplyTo method.
Co3InvocationComplete	Reply	A boolean header that tells the Co3 server whether processing is complete. The default for this header is true, so you only need to include it if you are sending an informational message and it is not complete. Also note that this header is ignored if the reply message is JSON.

Acknowledgements

Some action processors will consume messages and silently process them without returning any indication of progress or status to the Co3 server ("fire and forget"). Other action processors will return an acknowledgement when they have completed the processing of a message (“request/response”). Co3 supports either mode of operation through the Expect Acknowledgement setting of the message destination (see Figure 6).

When a message destination is configured with an Expect Acknowledgement value of “Yes” the list of executed actions in the Co3 UI will show messages/invocations as “Pending” until the expected acknowledgement is received. If an acknowledgement is not received within 24 hours, the Co3 UI will display it as an error. You 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 will immediately display 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)
```

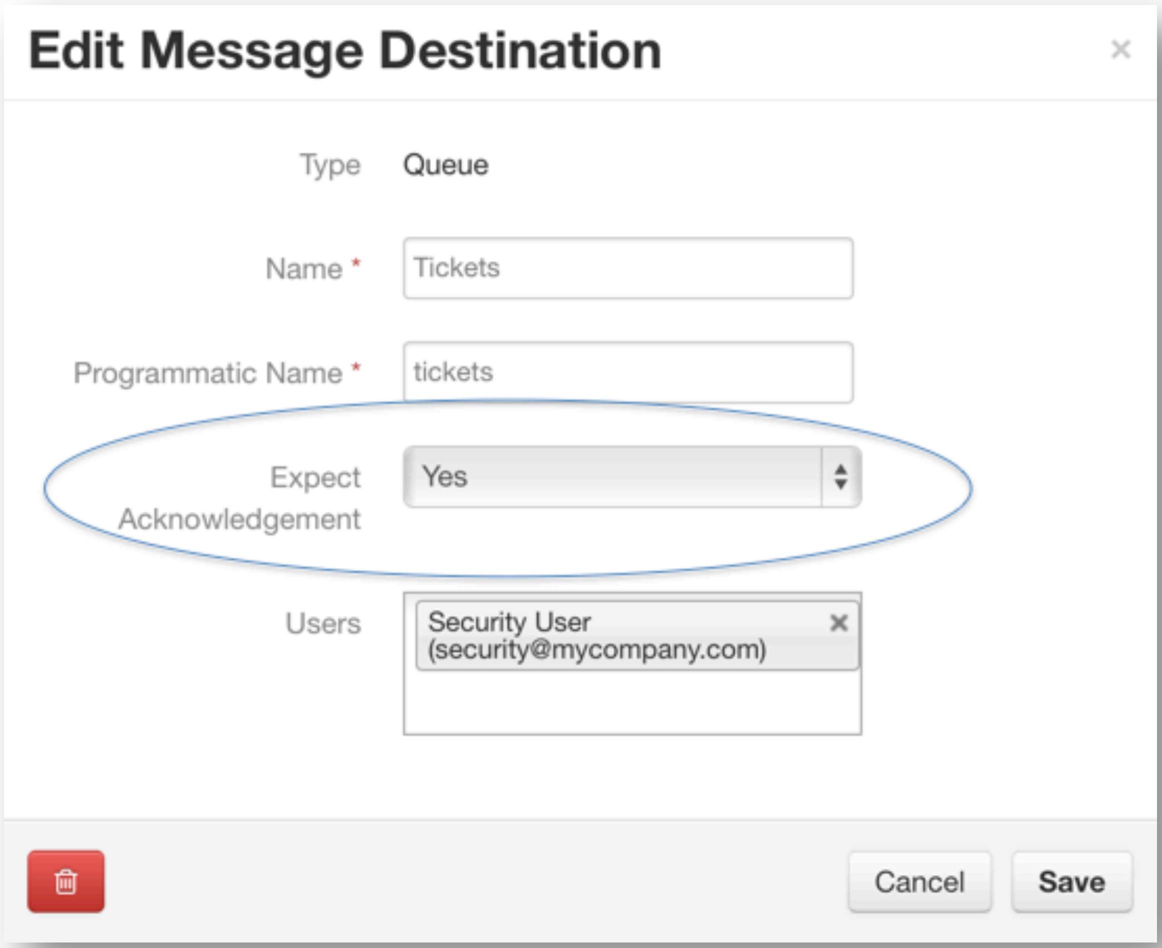
The Co3 server will accept either JSON or just a simple text string for reply messages. Simple plain text reply messages are a way to provide a success acknowledgement with minimal effort. You can also use a more descriptive JSON string value, which will be parsed by the server.

The format for the JSON messages is included in the Co3 REST API documentation (see the ActionAcknowledgementDTO type). For convenience, the following 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}

Table 3 Example ActionAcknowledgementDTO values



Type Queue

Name * Tickets

Programmatic Name * tickets

Expect Acknowledgement Yes

Users Security User (security@mycompany.com)

Figure 6 Message destination Expect Acknowledgement

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 your processors will very rarely fail. Unexpected replies will be displayed in the Action Status screen just as they are for expected ones.

TLS

Action processors must connect to the Co3 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 will vary 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 server name that you connected to?

It is known that some of the common JMS libraries for Java do not perform checking on the certificate name (#4 above). Co3 has developed a workaround for this, which is used in the Java examples.

Using the Co3 REST API with Action Processors

Co3 action processors can make use of the Co3 REST API to update incidents, retrieve additional information not included in the action 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 action has no conditions specified then it will trigger 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 will tell the server to skip the action that generated the original message.

```
# Use Co3 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.

HTTP Conflict (409) Errors

It is possible for the Co3 server 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 Co3 examples have accounted for this issue where necessary.

Using a Framework

There are frameworks that may simplify the development of Co3 action processors. You may want to investigate tools that may simplify the creation of action processors, which will generally follow typical "Enterprise Integration Patterns"[2].

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 (Co3 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 Co3 server Most of this can be done through XML- or DSL-based configurations. There is an example of how you can use Apache Camel with Co3 in the Co3 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 graphically create action processors using a number of built-in connectors.

[2] See http://www.enterpriseintegrationpatterns.com for more information about Enterprise Integration patterns.

		There is an example of how you can use Mulesoft with Co3 in the Co3 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).

Table 4 ESB and ESB-like frameworks worthy of investigation

Always Running

It's easy to write a CAF processor script that uses CAF to read action messages and perform some operation. When you are developing the script, it is fine to simply run it from the command line. However, when you exit the shell or log out of your desktop session, the program will exit.

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

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

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

Retry

You should consider how your CAF processor handles situations where external systems (including the Co3 server itself) are inaccessible.

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

If other downstream operations fail you will need to decide how to proceed. It may be sufficient to simply fail the operation and send a response message to the Co3 server indicating the failure (these messages will 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 Co3 CAF Apache Camel example in the Co3 API distribution.

Processor Installation

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

1. Create them manually using the web application
2. Write a program that uses the Co3 REST API to create them
3. Use the action-install.py Python script that is included in the Co3 examples

#3 is recommended because it reduces the chance of errors during deployment of your CAF processor.

Testing Considerations

Many of the design considerations discussed in the previous section will lead to useful test cases. For example, the discussion on Retry will lead a tester to a number of test cases dealing with how the CAF 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 of a CAF processor.

TLS

CAF processors connect to the Co3 system using TLS. This applies to both the connection to the message destination (STOMP over TLS and Active MQ/OpenWire over TLS) and the Co3 REST API (HTTPS).

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

Certificate Trust

When you invoke the CAF 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 (CAF processor in this case) to NOT trust the Co3 server certificate and confirm that the operation fails due to a TLS error.

Certificate Common Name

If a CAF processor thinks it's connecting to a host named "co3.mycompany.com", then it is important that the TLS certificate is issued to "co3.mycompany.com". If it's 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 way 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 Co3 server's IP address, then try to connect using testhost. The connection should fail. Note that this should be performed against both the Co3 REST API (port 443) and the CAF server (port 65000 and/or 65001).

Retry/Error Reporting

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

- Does the CAF processor have a log file?
- Does the CAF processor report errors from external systems?
- Does the CAF processor continue running when the Co3 server is down?

Always Running

The CAF processor should generally be running.

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

Conflicting Edits

If the CAF processor updates the Co3 server 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 an action when the CAF processor is stopped, then make another change to the object (say an incident), then start the CAF processor, does the CAF processor properly update the incident? Note: If it weren't handled by the developer, then you'd likely get an error when the processor attempts to do the PUT operation.

Action Status Sent

Does the CAF processor send a status or error message to the Co3 server as appropriate? These status messages appear in the Actions > Action Status dialog.

Infinite Loops

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

Others?

If there are other elements of the CAF processor that you think could be mentioned here, we'd love your feedback! Please contact us at success@co3sys.com.

Customer Success

If you need support in creating CAF processors, please contact someone in our Customer Success group at success@co3sys.com.