FORTINET

Playbooks Guide

FortiSOAR 7.6.5

FORTINET DOCUMENT LIBRARY
https://docs.fortinet.com

FORTINET VIDEO LIBRARY
https://video.fortinet.com

FORTINET BLOG
https://blog.fortinet.com

CUSTOMER SERVICE & SUPPORT
https://support.fortinet.com

FORTINET TRAINING & CERTIFICATION PROGRAM
https://www.fortinet.com/training-certification

FORTINET TRAINING INSTITUTE
https://training.fortinet.com

FORTIGUARD LABS
https://www.fortiguard.com

END USER LICENSE AGREEMENT
https://www.fortinet.com/doc/legal/EULA.pdf

FEEDBACK
Email: techdoc@fortinet.com

December, 2025
FortiSOAR 7.6.5 Playbooks Guide
00-400-000000-20201230

TABLE OF CONTENTS

Change Log **6**
Introduction **7**
Content Hub 7
Playbooks and Components **8**
Playbook Collections and Playbooks 8
Ownership, Roles, and Permissions 9
Setting Logging Levels 11
Simplified Expression View 12
Playbook Triggers 15
Trigger Steps 16
Trigger Block 43
Trigger Payloads and Data Structure 45
Playbook Steps 53
Core Steps 55
Evaluate Steps 74
Execute Steps 119
References Steps 131
Email Step 137
Authentication Step 140
Extending Playbook Steps with Actions 141
Tips for Working with Playbook Steps 154
Deprecated Playbook Steps 154
Creating and Designing Playbooks **156**
Playbook Designer 156
Tips on Designing a Playbook 157
Adding Blocks and Notes 160
Live User Implementation in Designer 164
Creating New Playbooks 164
Saving Versions of Playbooks 168
Exporting Versions of Playbooks 170
Editing Playbooks (Solution Packs, System Playbooks) 171
Importing Playbooks 180
Importing BPMN Shareable Workflows 180
Playbook Execution and Debugging **184**
Playbook Execution 184
Changing Prioritization of Playbook Execution 187
Disabling Prioritization of Playbook Execution 188
Playbook Debugging 189
Playbook Recovery 191
Viewing and Editing Blocks 192
Reference Blocks 192
Trigger Blocks 196
Viewing Executed Playbook Logs 197

Auto-Cleanup of Workflow Execution History ... 214
Advanced Playbook Development ... 217
Reserved Keywords ... 217
Dynamic Values ... 218
Jinja Editor ... 218
Dynamic Values Window Usage (I/O, Functions, Global Variables) ... 222
Jinja Template Security Enhancements and Configuration Options ... 237
Dynamic Variables ... 238
Syntax, Implementation, Scope, Functionality ... 238
Dictionary-like Objects ... 240
Built-in Functions & Filters ... 240
Jinja Filters, Functions, Expressions, and Extensions ... 241
Jinja Filters and Functions ... 241
Jinja Expressions ... 265
Jinja Extensions ... 269
YAQL and Custom Filters ... 270
YAQL Filters ... 270
Custom Functions and Filters ... 272
Maintaining Playbooks ... 275
Working with Content ... 275
Designing for Flexibility and Scalability ... 275
Leveraging Templates and Reusable Components ... 276
Documenting Playbooks and their Usage ... 276
Testing and Validating Playbooks ... 276
Optimizing Playbook Performance ... 277
Handling Playbook Errors and Failures ... 277
Security Considerations in Playbook Design ... 277
Supporting Multi-Tenancy ... 278
Monitoring and Continuous Improvement ... 278
Optimizing and Troubleshooting ... 279
Optimizing Playbooks ... 279
Optimized Workflow Runtime for Memory and CPU consumption ... 281
Managing the Day Filter Limit for Global Executed Playbook Logs ... 282
Code Snippet Connector: Usage Tips and Examples ... 283
Common Errors and Fixes ... 285
High CPU/Memory Usage ... 285
'Access Denied' Errors ... 285
Jinja Parsing Error: Integer Too Long ... 285
Playbook failure due to missing Jinja json_query filter ... 286
Picklist Item Errors ... 286
No Permission Errors ... 286
Ingestion Failures ... 287
Issues with Purging Executed Playbook Logs ... 287
Too Many Connections Error ... 287
"Picklist item not found" Error ... 287
Manual Input Email - Incorrect Link Error ... 288

Incorrect Hostname in System Playbooks 288
Database Connection Issues 288
Frequently Asked Questions **289**

Change Log

Date	Change Description
2026-01-22	Added the Playbook failure due to missing Jinja `json_query` filter topic in the *Optimizing and Troubleshooting* chapter.
2026-01-19	Added the Code Snippet Connector: Usage Tips and Examples topic in the *Optimizing and Troubleshooting* chapter.
2025-12-17	Initial release of 7.6.5

Introduction

FortiSOAR playbooks are central to its automation and orchestration capabilities. They enable automated security processes across external systems while aligning with organizational workflows. Playbook templates can be customized to reflect existing procedures and leverage FortiSOAR's powerful automation features.

Highly configurable, playbooks promote consistent and comprehensive execution of incident response plans—streamlining triage and accelerating threat containment.

This guide presents best practices for designing playbooks, identifying common pitfalls and how to avoid them. It is intended for both new users and experienced developers working on advanced workflows. Topics such as designing effective playbook logic, managing content via the Content Hub, configuring and utilizing connectors, and maintaining playbooks over time are included in this guide.

Starting with release 7.6.5, the csadmin user's sudo privileges are restricted to only the commands required to work with FortiSOAR, instead of providing full 'root' access. This enhancement aligns with the principle of least privilege and reduces exposure to sensitive system files. Therefore, commands such as `yum`, `systemctl`, `csadm`, etc, must be prefixed with `sudo`, for example, `sudo csadm --help`.
To open or edit a file, prefix the command with '`sudo`' and specify the file's full path (`sudo vi <full path of file>`).
For example, `sudo vi /opt/cyops-auth/utilities/das.ini`.
Additionally note that for security reasons, '`root`' access is provided via the system console and is not available over SSH.

Content Hub

The Content Hub is a centralized repository for out-of-the-box reference materials and product add-ons such as, connectors, widgets, and solution packs. Accessible directly within FortiSOAR, it allows users to browse, install, and manage these components seamlessly. Public access is also available at: https://fortisoar.contenthub.fortinet.com/, offering a comprehensive view of all available FortiSOAR add-ons and reference materials. This helps users maximize the platform's capabilities and improve overall utilization.

The Content Hub synchronizes with the FortiSOAR repository hourly to ensure all listings remain up to date.

To access the Content Hub, a user must be assigned a role that has a minimum of `Read` permission on the `Content Hub` and `Applications` modules.

For details, see the Access and Install Content from the Content Hub chapter in the "User Guide."

Playbooks and Components

Playbooks in FortiSOAR allow you to automate your security processes across external systems while respecting the business process required for your organization to function. Playbook templates can be customized to follow an organization's current procedures while leveraging the automation capabilities of FortiSOAR.

Playbooks are the key to empowering your organization with the full benefits of orchestration for both the human and machine side.

Playbooks can leverage a number of different FortiSOAR capabilities, such as inserting new data records, sending email notifications, and even referencing specified conditions to determine what path(s) to continue executing. Playbooks are highly configurable and provide consistent and thorough execution of IR response plans, enabling swift triage and containment of any potential cybersecurity threats.

The Playbook Engine runs asynchronously, meaning as an independent service, within the FortiSOAR application. This allows for better scalability and also frees the Application Engine to focus on request execution for better responsiveness to human users.

Playbook Collections and Playbooks

- **Organize with Playbook Collections**: Playbook Collections function like folders, allowing you to group and manage playbooks according to your environment's strategy. Consider these recommendations for grouping:
  - **By Functionality**: Group utility playbooks that serve multiple collections in a common folder, such as "*Utilities*."
  - **By Integration Target**: Organize playbooks by their integration target, for example, *Fortinet FortiAnalyzer*.
  - **By Actions**: Group playbooks by their purpose, such as *Investigation*, *Enrichment*, and *Remediation*. These can leverage integration playbooks..
  - **By Response Plans**. Structure collections to follow standard incident categories, using sequences of action playbooks.
- **Playbooks**: Playbooks are individual workflows consisting of a sequence of steps designed to achieve a specific goal.
  Note:
- **Naming and Descriptions**: Give playbook collections, playbooks, and playbook steps unique and descriptive names. Include a short summary in the description to clearly convey their purpose without opening the playbook or step.
  Example:
  Playbook Name: Phishing Email – User Reported – Triage and Mitigation

Description: Automatically triages user-reported phishing emails by extracting indicators, checking reputation, and isolating affected accounts if necessary.

- **Tagging**: Add tags to playbooks and playbook collections to indicate behavior, categorize, and facilitate filtering.
  Examples:
  '`dataIngestion`': Indicates the playbook ingests data from a third-party integration
  '`Schedule`': Indicates the playbook is triggered on a schedule.
- **Create Modular Playbooks**: Break large playbooks into smaller sub-playbooks for better reusability and maintenance.
  Example:
  A phishing mitigation playbook can have a parent playbook with child playbooks for enrichment, containment (e.g., blocking a domain), and notifications.
- **Leverage Conditions**: Use filters or conditional branches to skip unnecessary steps.
  Example:
  Add a condition to esclate an alert to an incident only if its `Severity is High or Critical`.
- **Document the Flow**: Add comments or notes in complex steps to facilitate knowledge transfer and future updates.

Ownership, Roles, and Permissions

Ownership: Ownership is determined by whether a playbook is created as 'Public' or 'Private '

- **Public Playbooks** (Default): Available to all users who have the necessary permissions. Public playbooks are visible and executable across teams.
- **Private Playbooks**: Intended for restricted workflows, sensitive operations, or team-specific use cases. Visibility and execution are limited to users in the assigned owner team(s). For steps to create 'Private' playbooks, see the Assigning ownership of playbooks topic.
  Visibility and Execution Behavior:
  - **Execution in Modules**: Only users belonging to the assigned teams will see private playbooks in the **Execute** dropdown within modules (e.g., *Alerts*, *Incidents*).
  - **Manual Triggers or Custom API Triggers**: Execution is allowed only if the initiating user belongs to an owning team.
  - **Automatic Triggers (e.g., On Create, On Update)**:RBAC is enforced at runtime. The system verifies whether the team(s) assigned to the record match those assigned to the playbook. Only playbooks with matching team assignments are triggered.

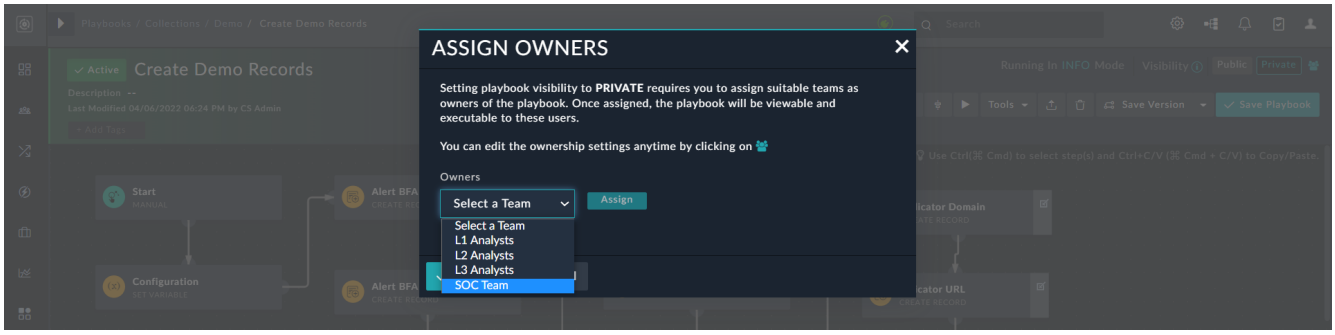
Ownership is lost during export: When a single private playbook or a collection is exported, all playbooks in the export package are converted to 'Public', and ownership assignments are removed.
After import, playbooks must be manually re-marked as 'Private, and ownership must be reassigned to the appropriate teams.

Assigning ownership of playbooks

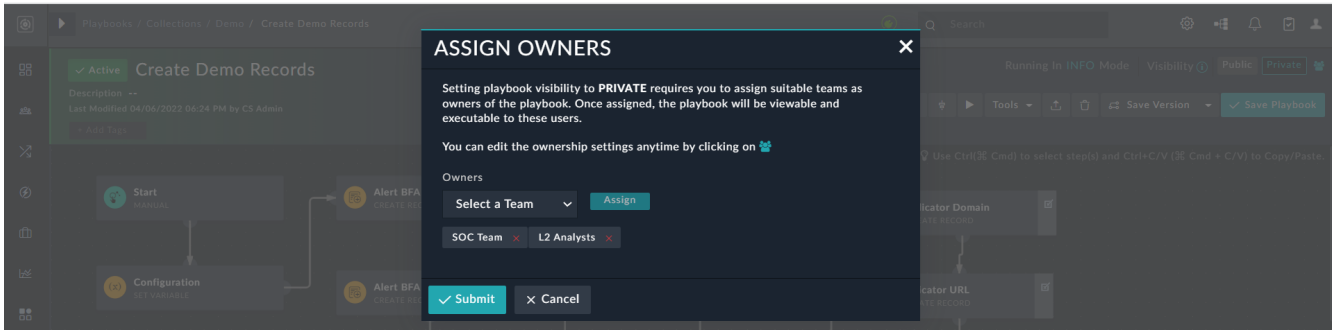
You can assign ownership to playbooks, i.e., if you want certain playbooks to be executed only by certain teams, then you can create a Private playbook and assign the playbook to only those teams.

By default, when you are creating a playbook, the playbook is created as a Public playbook, i.e., the playbook can be executed by all (if they have other appropriate rights). However, you can change this to Private by clicking the **Private** button that is present at the top of the Playbook Designer, as shown in the following image:

To assign the playbook to a particular team, click the **Teams** icon ( ). This opens the `Assign Owners` dialog. In the `Assign Owners` dialog, from the **Owners** drop-down list select the team that will own this playbook and click **Assign**.



You can make multiple teams, owners of this playbook in a similar manner. If you want to remove ownership from a particular team, click the **red cross** that appears besides the team name.



It is important to note that execute actions such as **Escalate**, **Resolve**, or any actions, which are displayed in the **Execute** drop-down list in records of modules such as Alerts, are shown based on ownership. For example, if you have created a **Private** playbook with a Manual Trigger or a Custom API Endpoint trigger on the `Alerts` module, and if you go to the alerts module and select the record, then **Execute** drop-down list will contain only those playbooks that belong to your team(s). In case of On Create or On Update triggers, RBAC is honored by matching the team defined in the playbook with the teams associated with the record.

When you export a playbook collection then all the playbooks within that collection become "Public" playbooks, even if some were marked as "Private playbooks, and the owners of the private playbooks become blank. Therefore, when you import these playbooks back into FortiSOAR, and you want the playbooks to be private, then open the playbook and click "Private" and reassign the owners. Exporting a single private playbook also marks it as public and its owners also become blank, and therefore, after importing this playbook into FortiSOAR, you will have to follow the same steps to make it "Private", if you want this playbook to be a "Private" playbook.

Role-Based Access Control (RBAC)

FortiSOAR uses Role-Based Access Control (RBAC) to manage access to playbook. Administrators require to assign roles with appropriate permissions (Create, Read, Update, Delete, Execute) in the Playbooks module to users who require to work with playbooks. For example, for users who require to run playbooks must be assigned the `Execute` permission on the `Playbooks` module.

Playbooks are executed by default in the context of Playbook Appliance (PBA).

Ensure that when you are creating a playbook that you give the PBA all the necessary privileges on all the modules that will be consumed while executing the playbook. For example, if you want to extract indicators from an incident record using a playbook, then the playbook must have a minimum of **Read** permission on the `Incident` module and the **Create** permission on the `Indicator` module..

Permissions

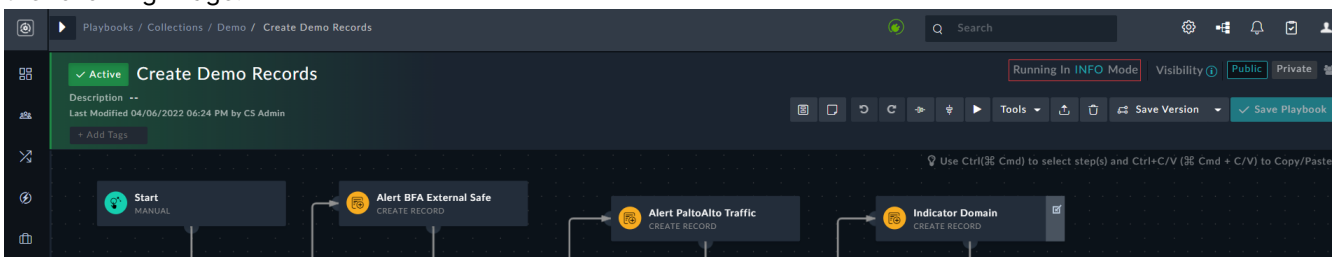
- To create Playbooks; you must be assigned a role with a minimum of `Create`, `Read`, and `Update` permission on the `Playbooks` module.
- To modify steps and to view steps in-depth, you must be assigned a role with a minimum of `Read` and `Update` permission on the `Playbooks` module.
- To view the Playbook Designer (you cannot view the steps in detail), you must be assigned a role with a minimum of `Read` permission on the `Playbooks` module.
- To create and delete Playbooks, you must be assigned a role with a minimum of `Create`, `Read`, `Update`, and `Delete` permission on the `Playbooks` module.
- To run Playbooks, you must be assigned a role with `Execute` permission on the `Playbooks` module.

Setting Logging Levels

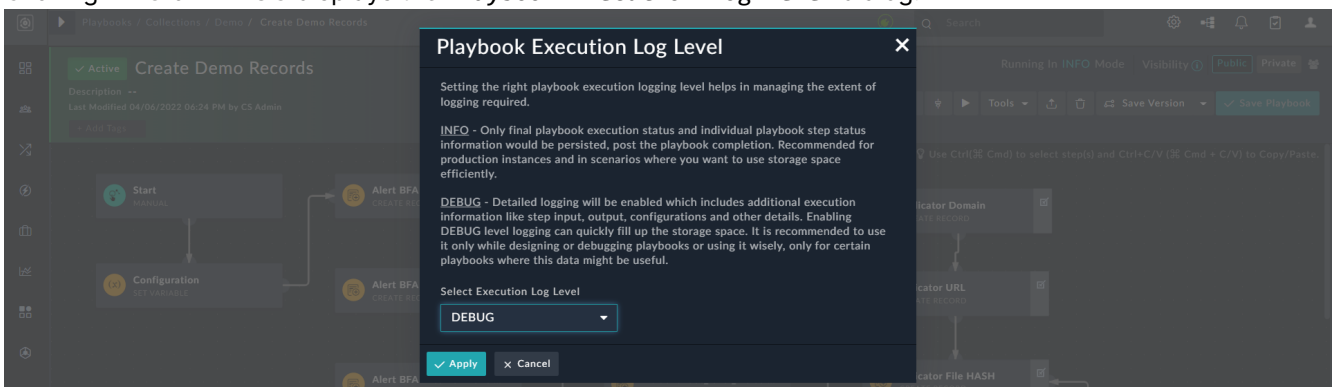
You can choose to set the logging levels for individual playbooks to either `INFO` (default) or `DEBUG`. From release 7.4.0 onwards, the default logging level for failed playbooks is set to `DEBUG` so that users do not need to rerun the playbook to view the exact reason for playbook failures.

Your administrator sets the global playbook logging levels; however, an administrator can also enable a setting that allows users to change the logging level for individual playbooks, which in turn overrides the global playbook logging levels. Set the playbook level logging to `INFO` for production instances and in scenarios where you want to use storage space efficiently; whereas use the `DEBUG` option only while designing or debugging playbooks since this option can quickly fill up the storage space.

To set the logging levels of a playbook, open the playbook in the playbook designer, and then click **INFO** or **DEBUG** (depending on the logging level set), which is present at the top of the Playbook Designer, as shown in the following image:



Clicking **INFO** or **DEBUG** displays the `Playbook Execution Log Level` dialog:



From the **Select Execution Log Level** field, select the logging level that you want to set for this playbook and click **Apply**. For example, if you select **DEBUG**, you will see that text at the top of the Playbook Designer changes to **Running in DEBUG mode**. Click **Save Playbook** to apply this change to the playbook.

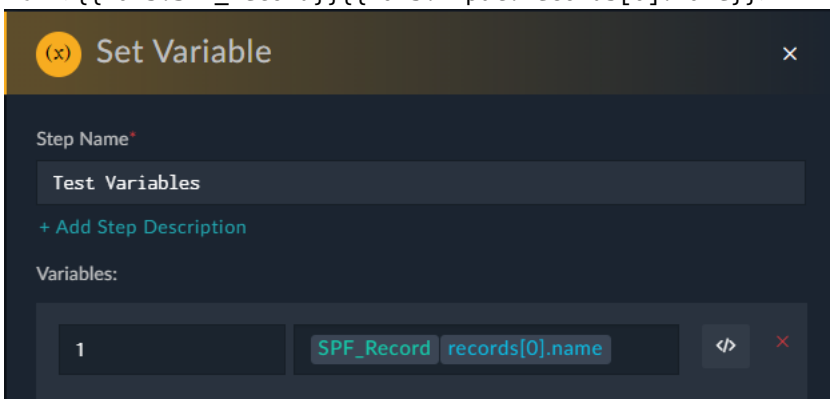
For more information on playbook execution logs, see the Playbook Execution and Debugging chapter.

Simplified Expression View

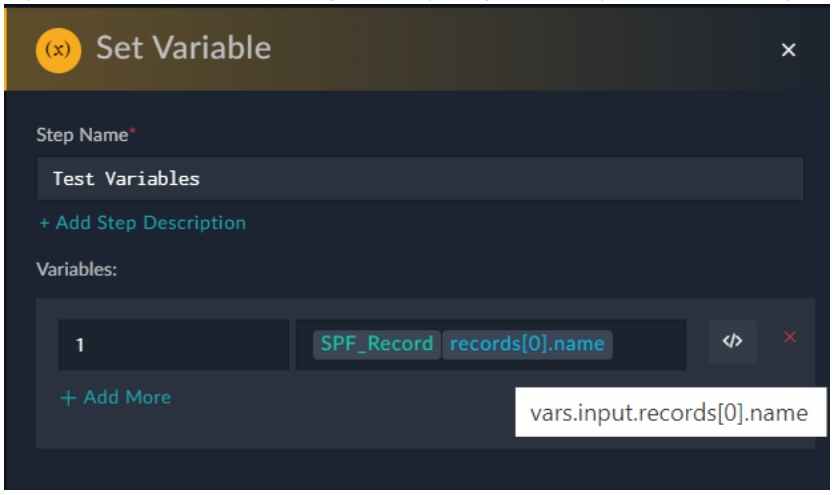
The 'Simplified Expression View' setting, is the default view for rendering expressions in the playbook designer, i.e., the playbook designer renders tag-based simplified expressions instead of the complete Jinja expressions. However, if your administrator disables this setting, then complete Jinja expressions are displayed in the playbook designer. For more information on usage of Jinja expressions, see the Dynamic Values chapter.

The 'Simplified Expression View' setting converts Jinja expressions into tags. The color of the tags are determined on the type of variable, i.e., difference colors are allocated for custom variables, global variables, step result variables and input variables. Following are some examples of Jinja expressions that are converted into tags:

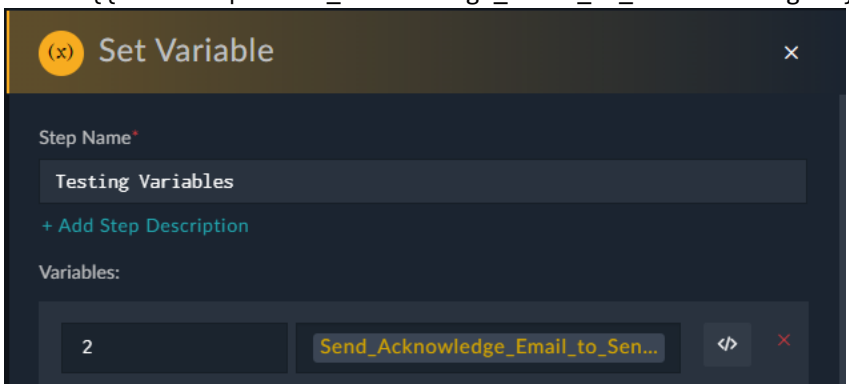
- Var1 : `{{vars.SPF_Record}}{{vars.input.records[0].name}}`:



In this example, `SPF_Record` is a custom variable, and `records[0].name` is a field (name) in the input record (0).
If you want to view the complete Jinja expression, you can do so by hovering on that tag:

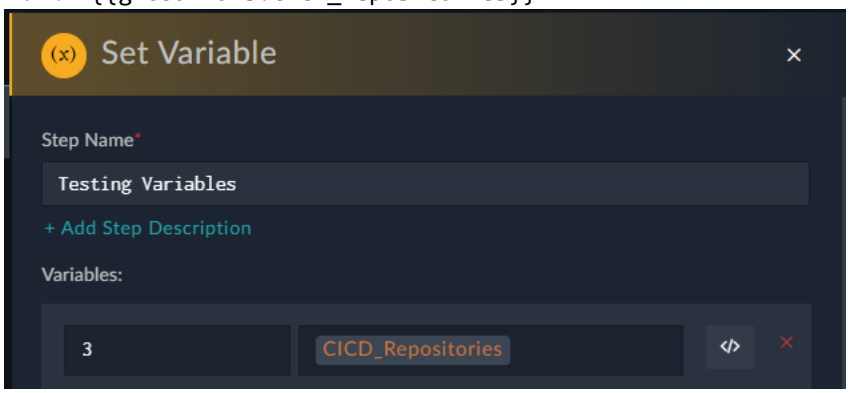


- Var 2 - `{{vars.steps.Send_Acknowledge_Email_to_Sender.assignee}}`



In this example, `assignee` is a field in a previous step named `Send_Acknowledge_Email_to_Sender`.

- Var 3 - `{{globalVars.CICD_Repositories}}`

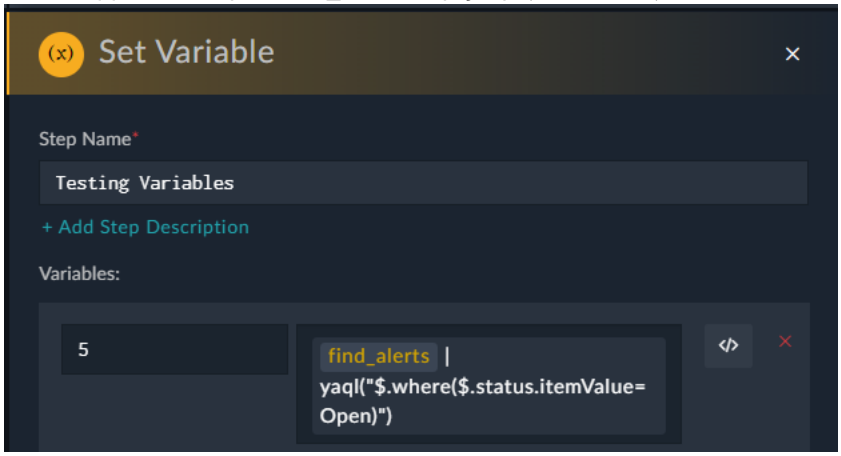


  In this example, `CICD_Repositories` is a global variable.

- Var 4 - `{{vars.foundList | extract_artifacts}}`

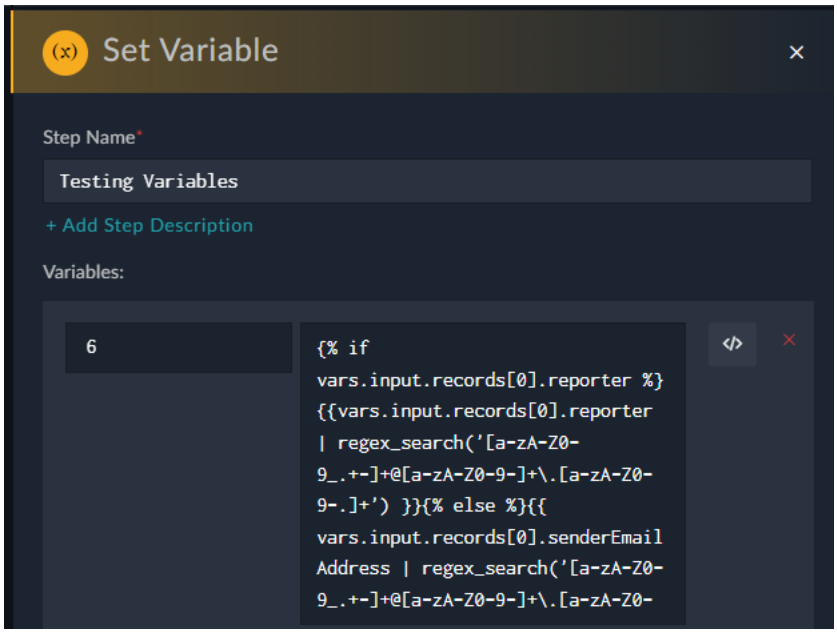
  In this example, `vars.foundList` is a custom variable on which want to run the `extract_artifacts` function.

- Var 5 - `{{vars.steps.find_alerts | yaql("$.where($.status.itemValue=Open)")}}`



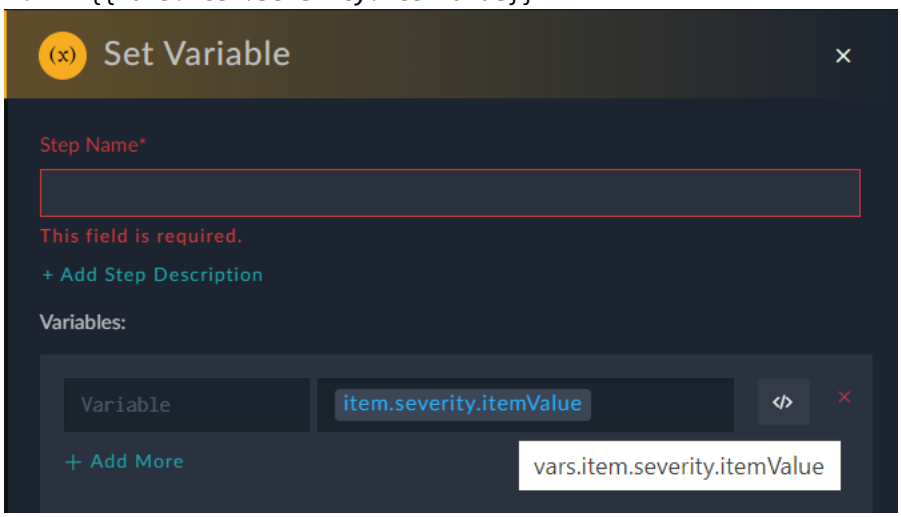
  In this example, a 'yaql' expression is run on the previous step named `find_alerts`.

- Var 6 - `{% if vars.input.records[0].reporter %}{{vars.input.records[0].reporter | regex_search('[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+') }}{% else %}{{ vars.input.records[0].senderEmailAddress | regex_search('[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+') }} {% endif %}`



In this example, we have a code snippet. Currently FortiSOAR does not convert code snippets to tags.

- Var 7 - `{{vars.item.severity.itemValue}}`



In this example, we are using the `itemValue` of a item (i.e. `severity` picklist) that is part of a loop in that step.

Playbook Triggers

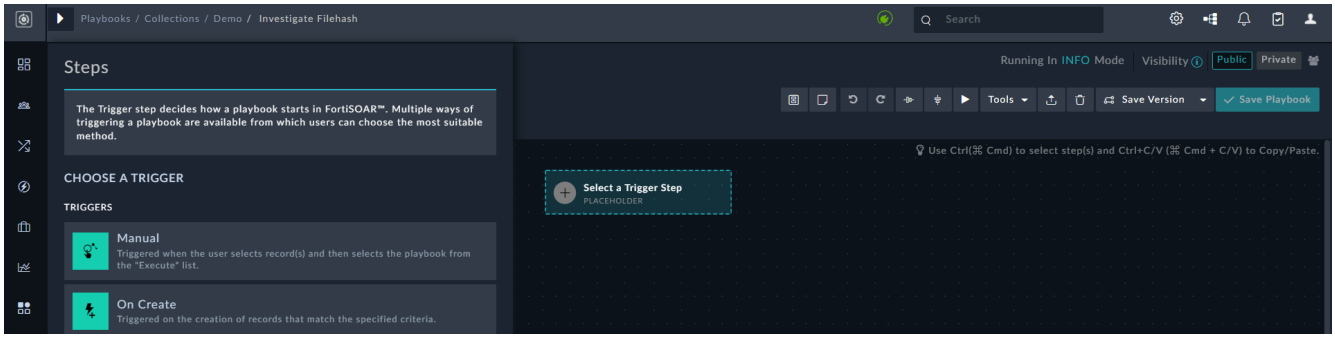
Triggers define when a Playbook is to be executed.

Trigger Steps

Trigger steps are always the *first step* in a playbook and define when the playbook is executed. Once triggered, the playbook flows through the remaining defined steps, following the routes on the canvas with the trigger as the starting point.

There are six different trigger steps defined in the Playbook Engine. Most triggers are based upon actions that you can perform on models in the FortiSOAR database. The parameters of each are defined below.

Once you add a playbook, the playbook gets created with a placeholder **Trigger** step as shown in the following image. Then specify the required parameters for the trigger and then click **Save** to add the first step to the playbook. The procedure for creating playbooks is mentioned in the *Playbooks Overview* section.



You can add **Step Utilities**, i.e., Variables and Messages for all triggers. Add variables for all trigger by clicking the **Variables** link that appears in the playbook step footer to add input variables. Input variables are the inputs that are required to be provided by the user at the time of playbook execution. Required variables are made available in the environment based on the given name. Required variables can be of any standard field format within the UI, including text, picklist values, lookup, and checkboxes. See Variables for more information. You can also add a custom message for each playbook step to describe its behavior. See Message for more information.

Manual

The Manual Trigger allows you to call a specific playbook from within any module in the system, i.e., these are for click-to-start playbooks. You can then execute any desired operations within that playbook on demand.

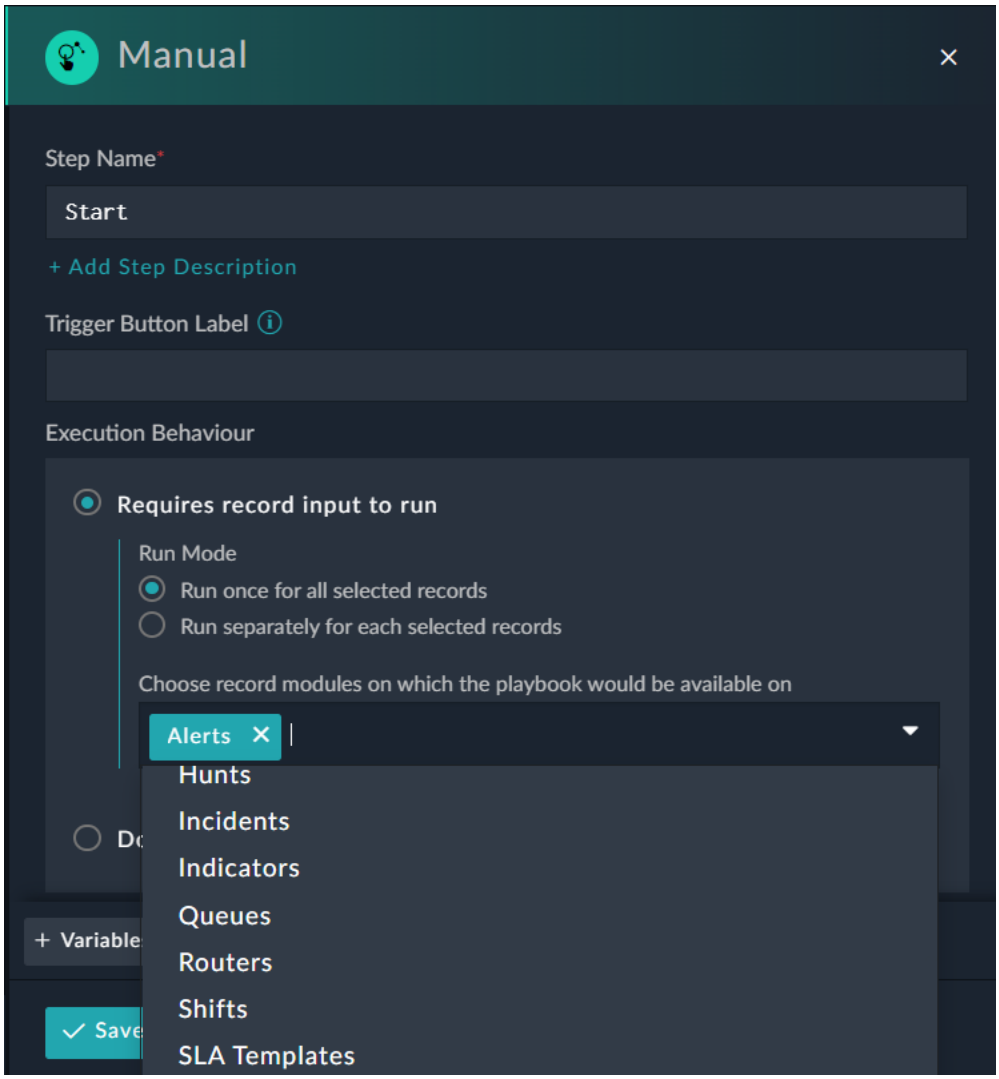
To add a Manual trigger, click **Manual Trigger** in the `Playbook Designer`, type the name of the step in the **Step Name** field. In the **Trigger Label Button** field, type the name that will be displayed in the selected module (s) to trigger this playbook. The name that you specified in this field is what the user will see in the **Execute** drop-down list on the module list.

The Manual Trigger step provides you with options to specify whether the execution behavior of the playbook, i.e., you can decide whether the playbook requires a record to be executed or if it does not require a record to be executed. If the playbook requires a record to be executed, then select the **Requires record input to run** option and then select the run mode, i.e., if the action must be executed once, then select the **Run once for all selected records** option or if the action must be executed separately for each selected record then select the **Run separately for each selected Record**. By default, the **Run once for all selected records** option is selected. This makes it more effective to handle multiple selections since you do not require to write two playbooks and map the second playbook in the first playbook.

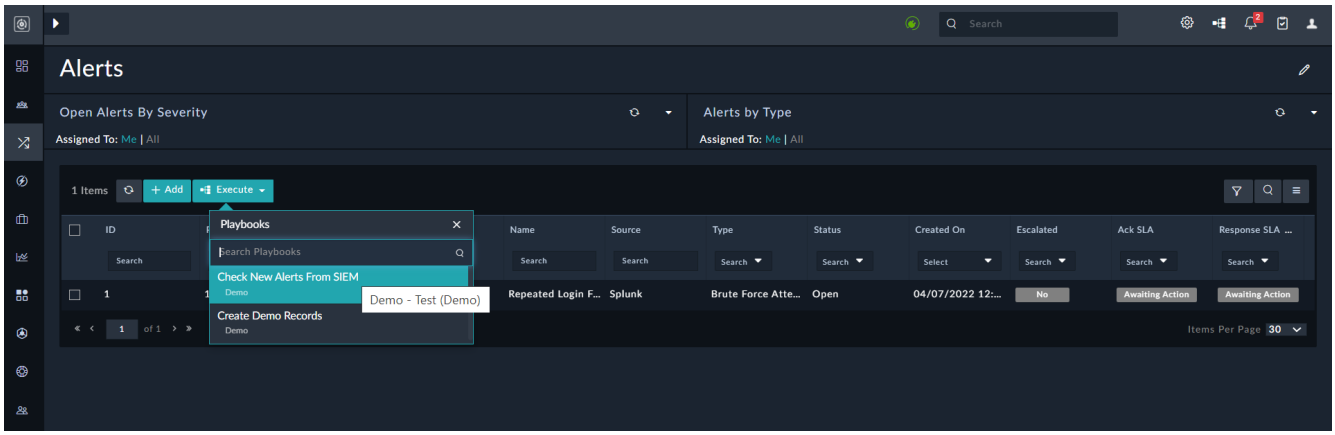
In the case of a "Manual Trigger" step that has the **Run separately for each selected records** option selected, and in which you have selected multiple records and triggered a playbook from the designer for debugging purposes, you will observe that only a single playbook will be triggered on a single record to simulate the output. For information on triggering playbooks from the playbook designer for debugging, see the *Playbook Execution* topic.

If you want the playbook to run without having to select a record, then select the **Does not require a record input to run** option. This acts as a module-based trigger, i.e., you can trigger a playbook based on a selected module without having to select a record in the specified module. An example of this could be a manual trigger to check for new alerts from a SIEM tool could be run globally on the `Alerts` module.

In either of the cases, from the **Choose record modules on which the playbook would be available on** select one or more modules on which you want to register this trigger and execute the playbook. For example, you can choose `Alerts` and `Incidents`. When you select this Manual Trigger from the **Execute** drop-down list, the playbook gets executed, and at the time of execution, the record (s) of the registered module (s) are passed into the playbook environment with the trigger.



The playbook that you create with the **Does not require a record input to run** option will appear in the **Execute** drop-down list in the module, or you can also create a specific button for this action, by updating the module template. In case of our example, when you open the **Alerts** module, you will see the **Check New Alerts From SIEM** option in the **Execute** drop-down list in the module (when you do not select any record), as shown in the following image:



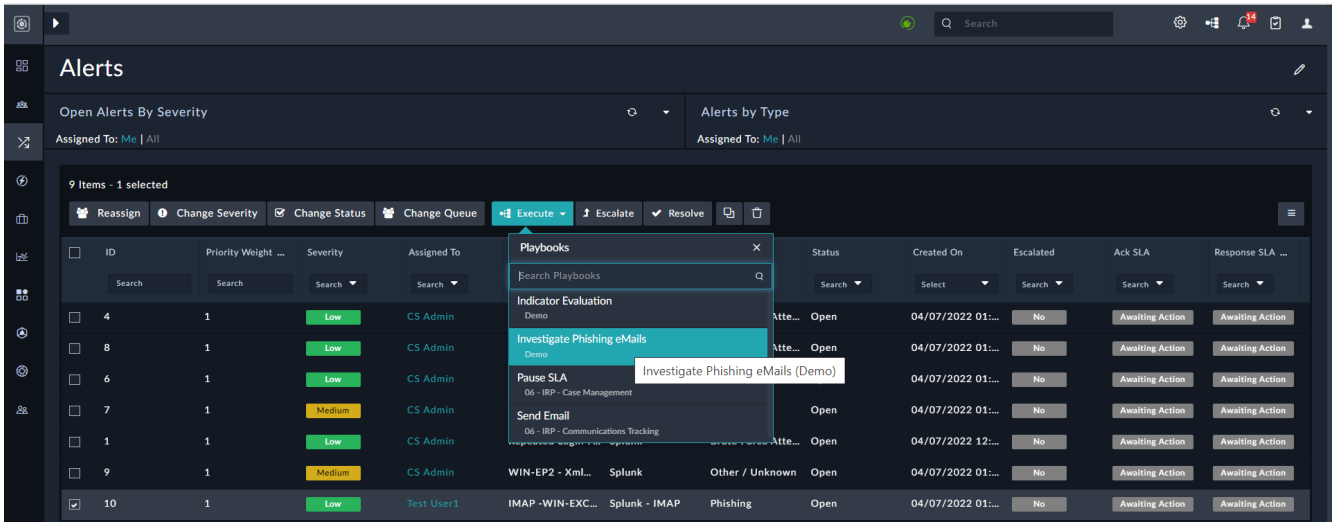
However, if you select a record in the Alerts module, then you will observe that the **Check New Alerts From SIEM** option will not present in the **Execute** drop-down list.

When you choose **Run once for all selected records** in the `Execution Behaviour` section, then a single playbook is run with the input set as `vars.input.params.records`, which is an array containing a list of all selected records that acts as an input to the playbook. When you choose **Run separately for each selected record** in the `Execution Behaviour` section, then one instance of the playbook is run per selected record with the input to the playbook set as `vars.input.params.records`.

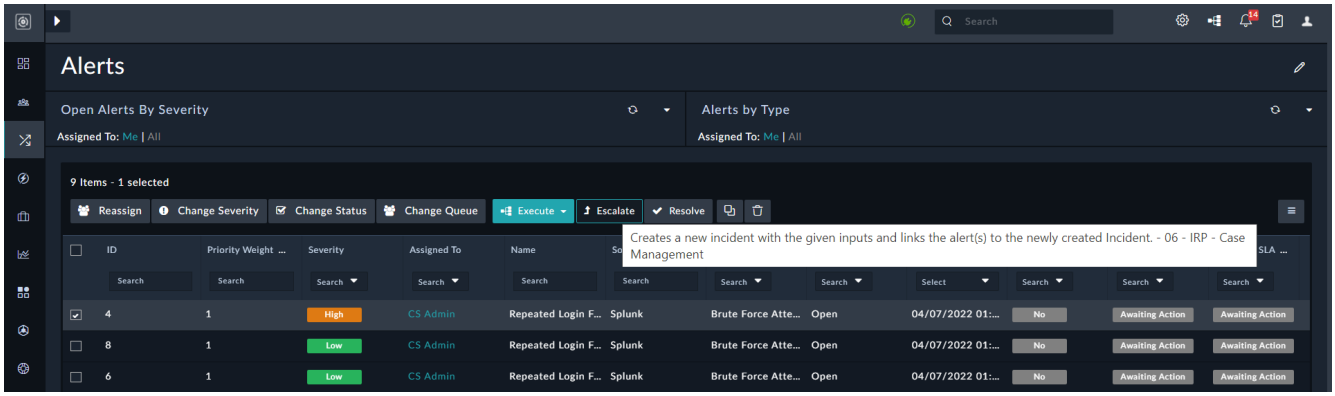
You can also build a customized user prompt form by adding multiple types of input fields of standard field format within the UI such as Text, Picklist, Lookup in the `User Prompt` section. For more information on building user prompts, see the Building a User Prompt topic.

Once you have completed providing all the above parameters, click **Save** to save the Manual Trigger.

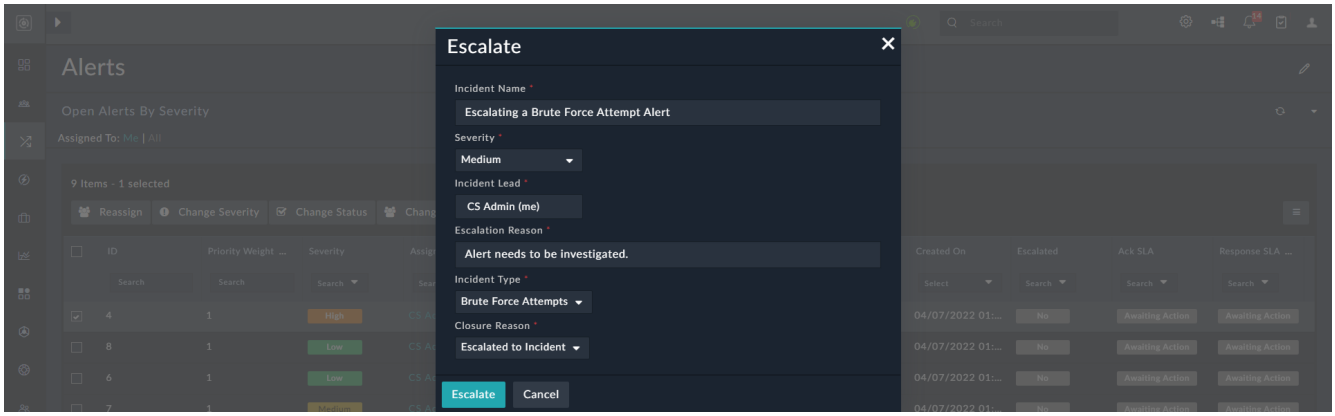
When you want to execute a playbook action, click the module (s) on which you have registered the Manual Trigger. This would be a module you have specified in the **Choose record modules on which the playbook would be available on** field. In the grid view of this module, click one or more records to display the **Execute** drop-down list (if you have selected the **Requires record input to run** option). Pressing the down arrow provides a list of available actions. The name of the action displayed is based on the name that you have specified in the **Trigger Label Button** field, for example, if you have entered `Investigate Phishing eMails` in the **Trigger Label Button** field, then **Investigate Phishing eMail** is an option in the **Execute** drop-down list as shown in the following image:



An example of a Manual Trigger that is included by default in the `Alerts` module in the form of an 'Action' button is the **Escalate** action. Select an alert record or records and click **Escalate** to automatically create a new incident based on the inputs you provide in the `Escalate` dialog, links the alert(s) to this newly created incident as well as links the newly created incident to the alert(s). Note that in the case of MSSP setups, ensure that alerts selected to be escalated to an incident belong to the same tenant, else an error is displayed and the selected alerts are not escalated.



When you initiate an action with an associated required input variable, such as the **Escalate** action, you will be prompted to enter that information in an input dialog as shown in the following image:

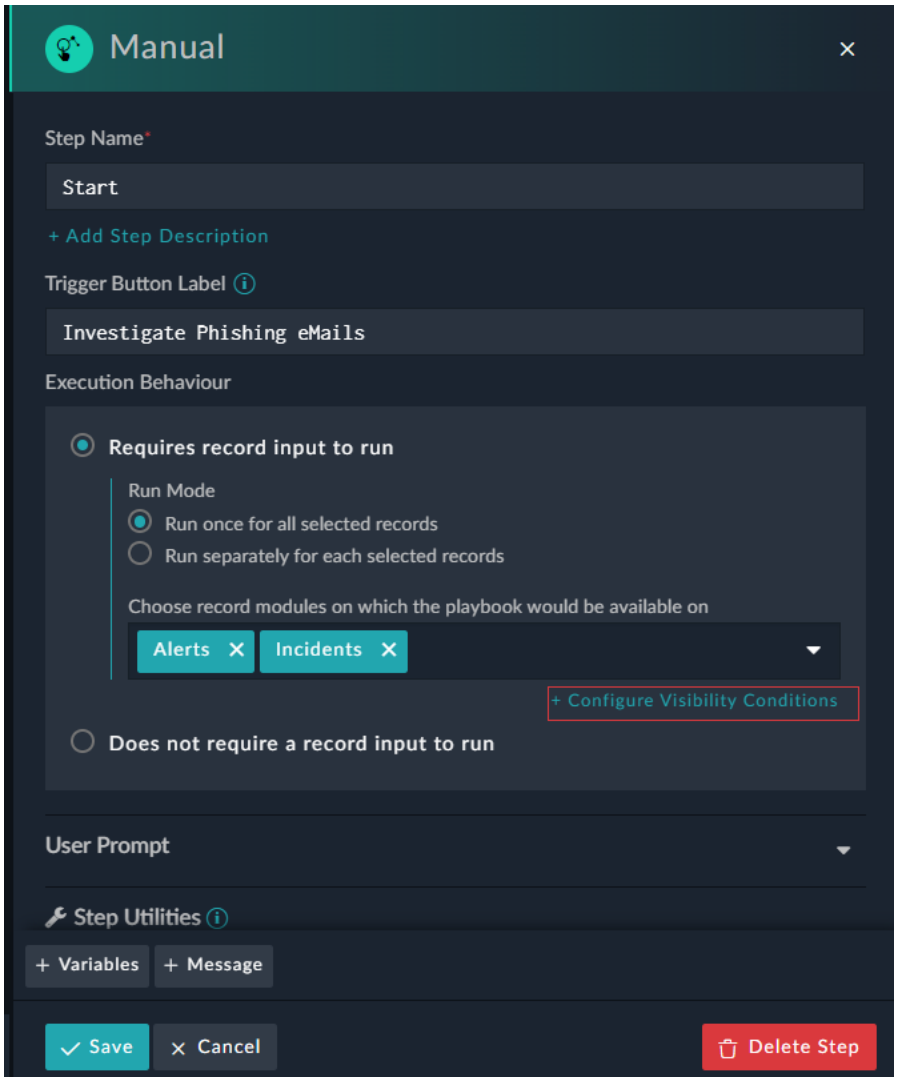


Enter the required information and click **Execute** to execute the Escalate playbook.

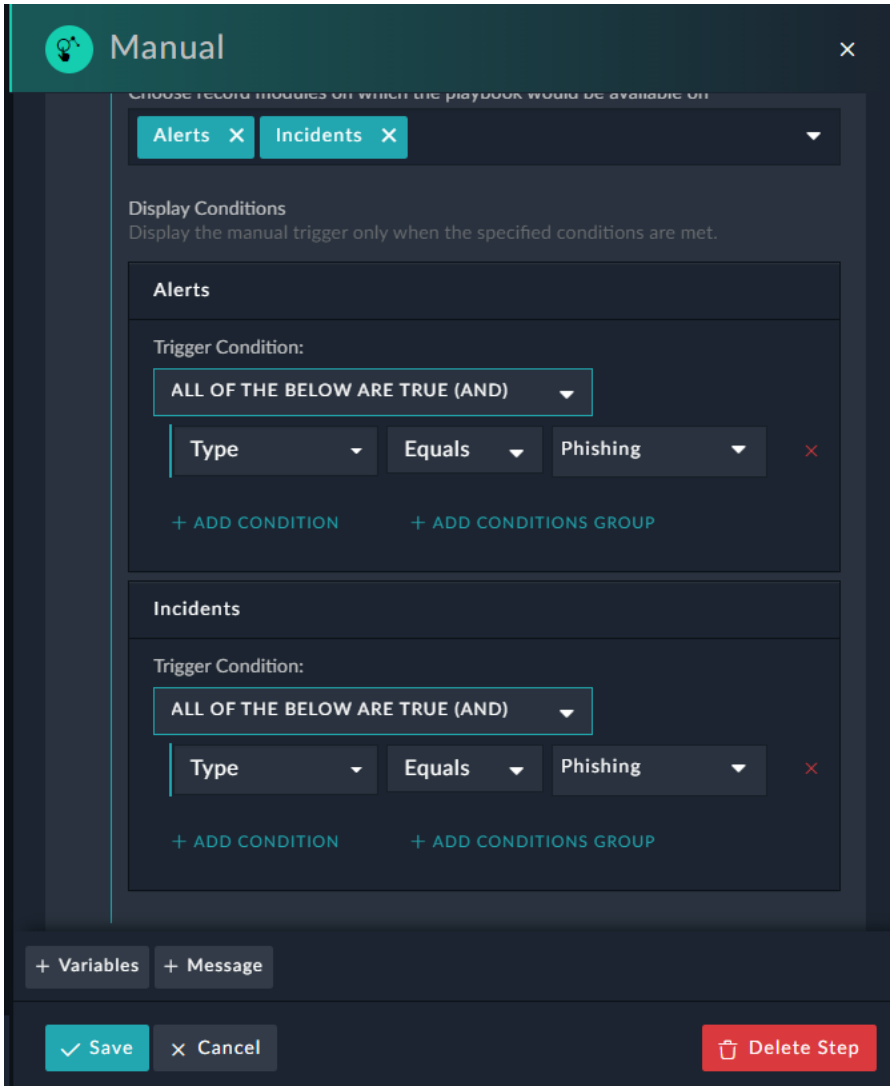
For polling playbook execution results for a playbook triggered using an SVT (Actions) button, for example 'Escalate', by default, FortiSOAR does not wait for any playbook execution results, and displays a "Triggered Successfully" toaster message, once you click the **Escalate**. If you want the playbook to wait it completes its execution and then display the toaster message, then you must add the `SystemWaitForExecution` tag:

You can also define visibility conditions for those playbooks that require record input to run. You can define conditions on records, such as the specific record type or severity or status; thereby enabling users to see only those actions (playbooks) that apply to records that match the defined condition. For example, a `Submit Malware Sample` playbook should be visible for a "Malicious Code" incident, but it should not be visible for an “Unauthorized Access” incident.

In the `Execution Behaviour` section, if you have selected the **Requires record input to run** option, click the **Configure Visibility Conditions** link to add the visibility conditions as shown in the following image:



You can define distinct conditions for each selected module in the playbook, as separate sections for each selected module is displayed; thereby allowing you to apply different display conditions (filters) for each selected module as shown in the following image:



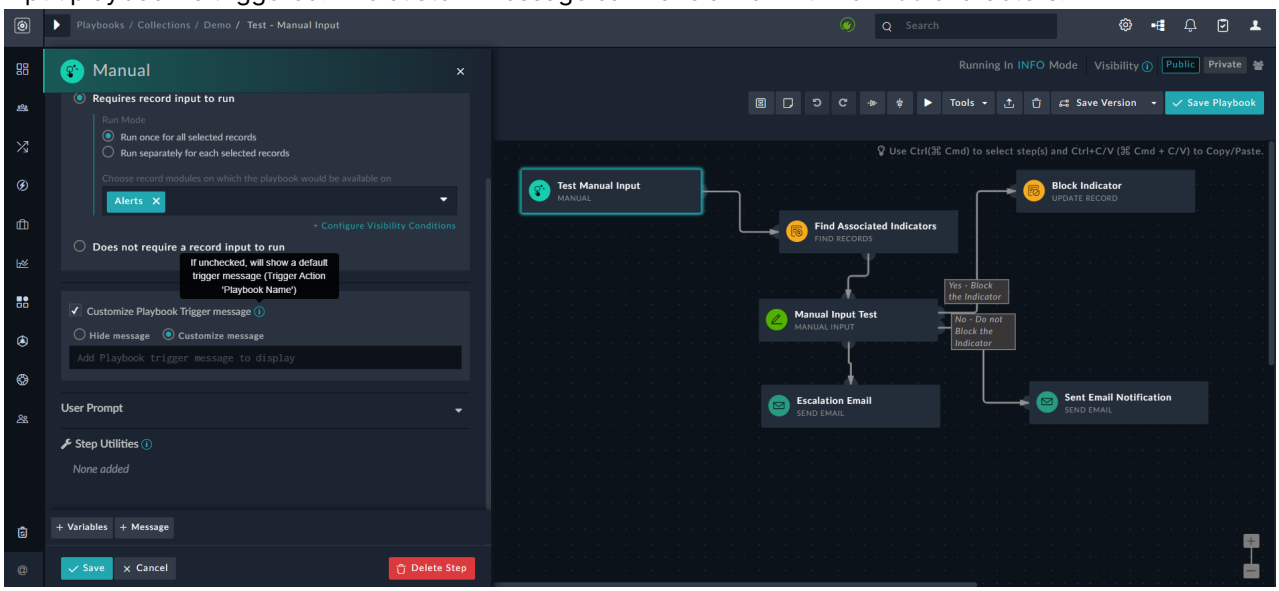
In the above image **Alerts** and **Incidents** modules are selected in the **Choose record modules on which the playbook would be available on** list, and therefore the `Display Conditions` section contains trigger conditions for both the Alerts and Incidents modules. From release 7.6.0 onwards, you can add conditions to trigger a playbook for specific records based on the record's UUID. You can include multiple UUIDs in a list or CSV format.

If you do not specify any display conditions, then all playbooks that you have defined for the modules can be viewed when you select or open a record in the **Execute** list. An example of a condition based on which a playbook is displayed when a user selects a record would be a playbook that is defined to be run only on "alerts or incidents of type phishing." In this case, in the manual trigger step, in the **Trigger Button Label** field you could type `Investigate Phishing eMail`, and in the `Display Conditions` section, you would define a **Trigger Condition**`Type Equals Phishing` for both the Alerts and the Incidents modules as shown in the above image.

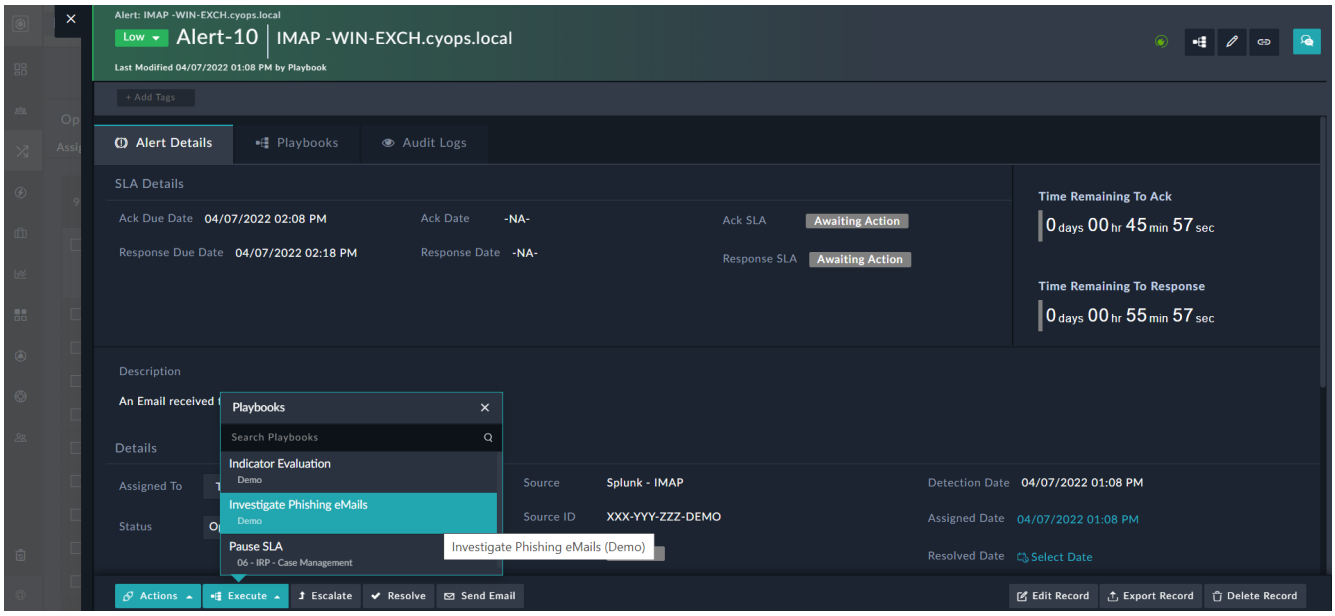
If you add a filter in a `Trigger Condition`, with an `Equals` or `Not Equals` logical operator to a richtext content field, such as **Description**, then you must enclose the content you want to filter in `<p>...</p>` tags.

Additionally, you can decide whether to customize the message that is displayed when a manual trigger playbook is triggered or to disable the "`Triggered action 'Playbook Name' on <number of records> record`" system message.

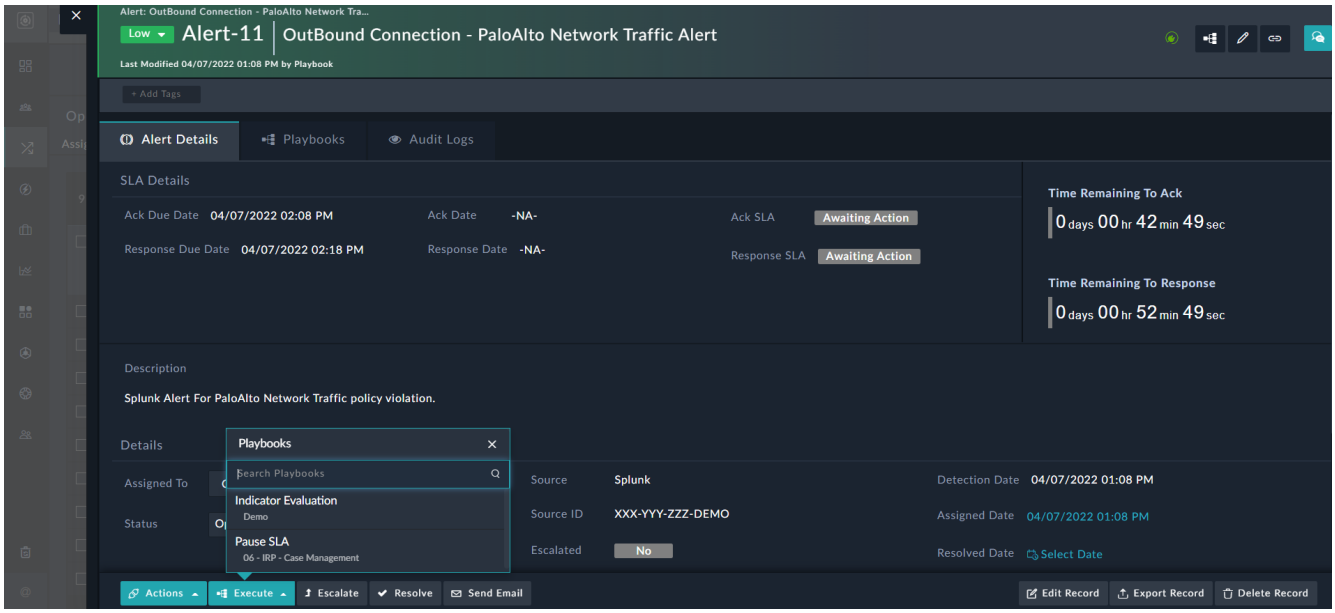
- To display the system message, when a manual trigger playbook is triggered, leave the **Customize Playbook Trigger Message** option as unselected (default).
- To hide the system message from appearing when the manual trigger playbook is triggered, select the **Customize Playbook Trigger Message** option, and then select the **Hide message** option. You might want to hide the messages, in case of playbooks where a manual input form is displayed right away after the playbook is triggered, and therefore the display of this message becomes superfluous and can detract from the user experience.
- To display a custom message to users, select the **Customize Playbook Trigger Message** option. Then, in the **Add Playbook trigger field to display** field, type the custom message to be displayed when the manual input playbook is triggered. The custom message can have a maximum of 255 characters:



Based on the defined visibility condition, then users will see this playbook only when the alert record is of type phishing. For example, you have alert records: `Alert 10` whose type is set to Phishing, and `Alert 11` whose type is set to Policy Violation. When you click the **Alert 10** alert record to view its details, in the **Execute** list you can see the `Investigate Phishing eMail` playbook listed, as showing in the following image:



However, when you click the **Alert 11** alert record to view its details, in the **Execute** list you will not be able to see the `Investigate Phishing eMail` playbook as seen in the following image:

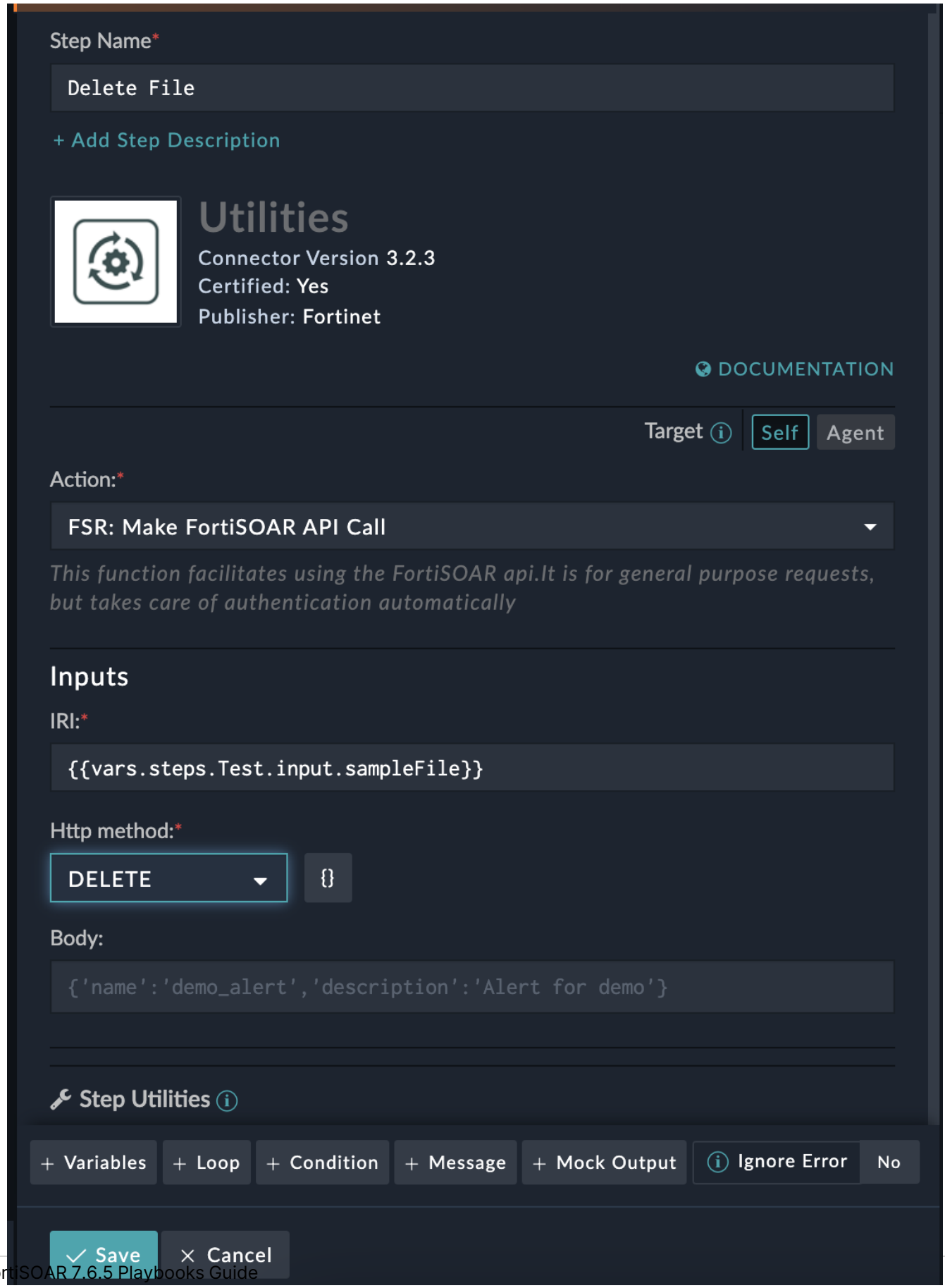


Building a User Prompt

You can build a customized user prompt form by adding multiple types of input fields of standard field format within the UI such as Text, Picklist, Lookup, File, Phone, Integer, Decimal, Date/Time, Dynamic list, Checkbox, Email Field, and Email Template Field. If you select the field format as **Text**, you can also define its **Sub-type** such as Text Field, Domain, Rich Text, etc. Click **Add Field** in the `User Prompt` section, to add an input field to build your user prompt. The User Prompt enables you to create a customized user prompt.

Important: If you use "File" as part of inputs in the 'Manual Input' step, i.e., if users can upload files as part of the input prompt, then you must ensure that once the file has been processed in the playbook, it is deleted. You can

delete the file using the "Utilities" connector's "`FSR: Make FortiSOAR API Call`" step. In this step, specify the `DELETE` method and the IRI of the file to be deleted:


Step Name*
Delete File
+ Add Step Description
Utilities
Connector Version 3.2.3
Certified: Yes
Publisher: Fortinet
DOCUMENTATION
Target
Self
Agent
Action:*
FSR: Make FortiSOAR API Call
This function facilitates using the FortiSOAR api.It is for general purpose requests, but takes care of authentication automatically
Inputs
IRI:*
{{vars.steps.Test.input.sampleFile}}
Http method:*
DELETE
{}
Body:
{'name':'demo_alert','description':'Alert for demo'}
Step Utilities
+ Variables
+ Loop
+ Condition
+ Message
+ Mock Output
Ignore Error
No
Save
Cancel

You can now configure the following in the User Prompt:

- Specify field titles and variable names, instead of having the field title being built automatically.
- Add tooltips for fields.
- Change the action button name to a name of your choice; the default is `Execute`.
- Display a pre-populated form field in the input form for review or modification, before executing any action. One benefit of this feature is the ability to review certain fields that will be used in the playbook, such as a source IP address or closure notes.

You can build the user prompt using custom fields and fields from input record, if you have selected a single module, (e.g., Alerts) and not when you have selected multiple modules. If you select multiple modules, then you can build the user prompt using only custom fields.

The `User Prompt` section also contains an additional field where you can specify default values. The values entered in this field would be displayed when the `User Prompt` is shown to a user. You can either specify any custom value (if your input type is selected as **Custom**) or any default record field (if your input type is selected as **Record Field**). The listing of record fields will be based on the module that you have selected in the **Choose record modules on which the playbook would be available on** field. Once you select the record field, then the data of this field will be loaded from the specified input record and displayed to the user in the `User Prompt`.

The default values will not update the record; they are only used to display content in the User Prompt

An example of building a user prompt in FortiSOAR will be if you want to reassign a number of alert records to another user after specifying a note. This example will also demonstrate how you can use custom field titles and variables and customizing the **Execute** button name. This example assumes that you have selected **Alerts** from the **Choose record modules on which the playbook would be available on** field.

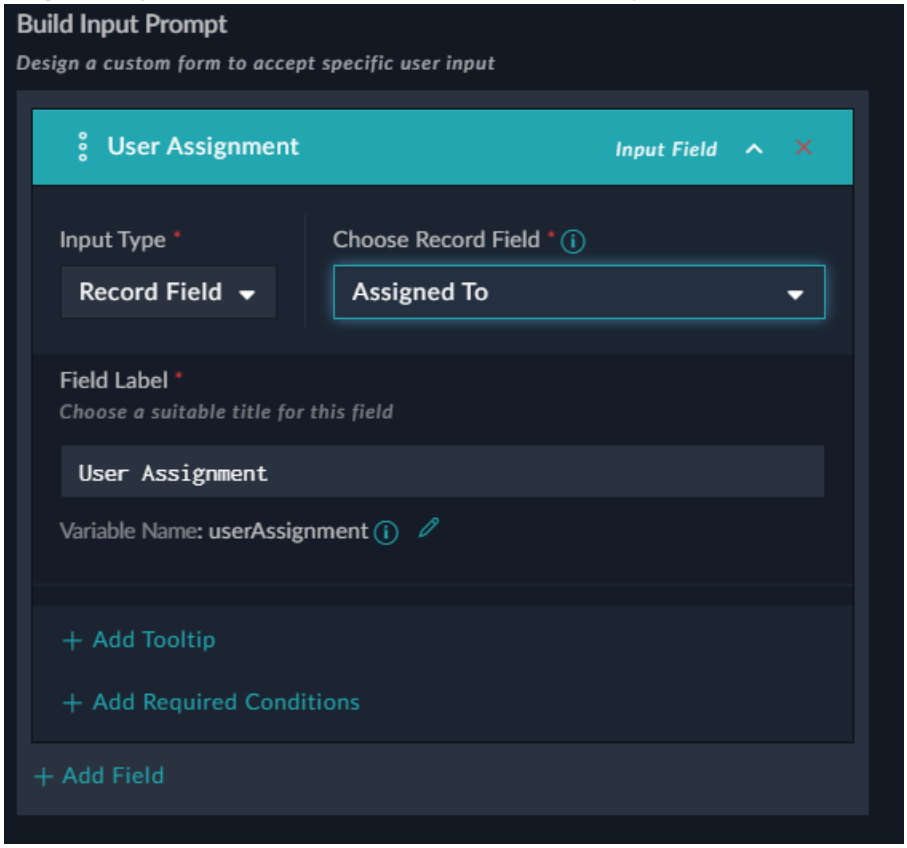
Steps to create this example user prompt is as follows:

1. In the `User Prompt` section, click the down arrow, and then click **Add Field**.
2. To reassign the alert record to another user by using an input record (Assigned To) do the following:
   a. From the **Input Type** drop-down list, select **Record Field**.
   Note: If you have selected multiple modules in the **Choose record modules on which the playbook would be available on** field, you cannot select Record Field from the Field Type drop-down list and you can create the User Prompt using custom fields only.
   b. From the **Choose Record Field** drop-down list, select the field that will be set by default.
   The field listing in the **Choose Record Field** drop-down list is dependent on the module you have selected in the **Choose record modules on which the playbook would be available on** field. For our example, we have chosen **Alerts**.
   For our example, select **Assigned To**.
   c. In the **Field Label** field, type the label of the field that will be displayed in the User Prompt.
   For example, `User Assignment`.
   The **Variable Name** field type gets auto-populated with the variable name, for example, `userAssignment`. You can edit the variable name if you want.
   Important: If you are using Dynamic Values in the next step of the playbook note that Dynamic Values

will display *parameters* in the **Input/Ouput > Input > Parameters** option, and *Input Record Fields*, such as `AssignedTo` in the **Input/Ouput > Input > Records** option.

d. (Optional) If you want to provide more information about the field, then click the **Add Tooltip** link and enter the description in the **Tooltip** field.

e. Starting with release 7.6.5, a new option, **+Add Required Condition** is added, which allows users to specify criteria when a field is made mandatory during step execution. The following options are available:
Not Required: The field is optional.
Required:The field is always mandatory.
Required by condition: The field becomes mandatory when defined conditions are met.



3. To create a custom field for providing a reason or notes for the reassignment, do the following:

a. Click the **Add Field** link.

b. From the **Input Type** drop-down list, select **Custom**.

c. From the **Field Type** drop-down list, select **Text**. You can also select **Email Template Field**, as a field Type. For more information, see User Input Prompt - Using the 'Email Template' Field topic.
Note: If you select the field type as "Text", you can also choose its **Sub-type**, such as Rich Text, Text Area, IP, etc. Also, if you select the field type as "Picklist" then you must select the corresponding picklist, and if you select "Lookup", then you can specify the related module.

d. In **Field Label** field, type the label of the field that will be displayed in the User Prompt.
For example, `Notes for Reassignment`.
The **Variable Name** field type gets auto populated with the variable name, for example, `notesForReassignment`. You can edit the variable name if you want.

e. (Optional) In the **Default Value** field, you can enter the default value for the custom field.
Note: You can specify either a "Static" date/time or a "Custom" date/time as a default value, if your

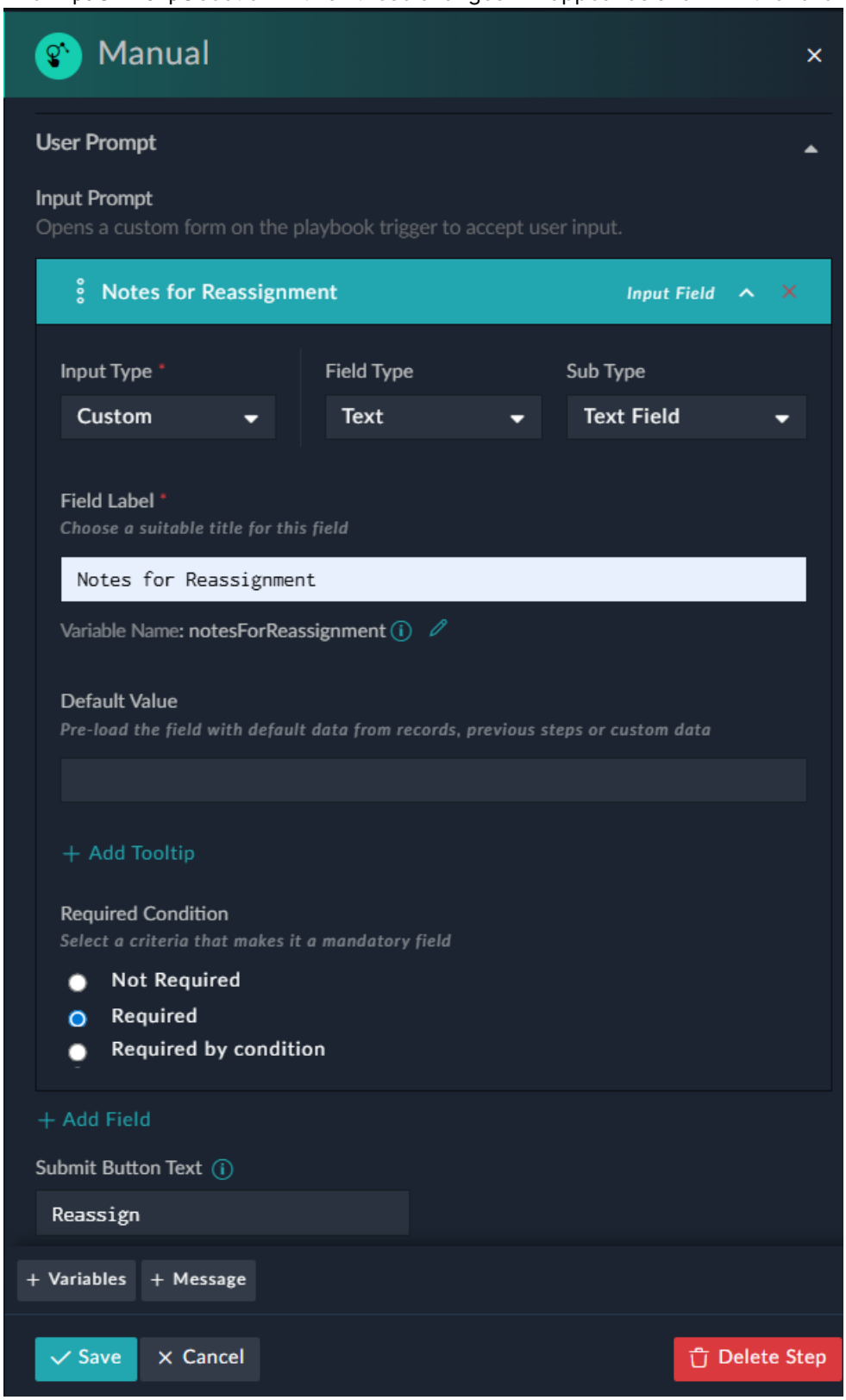
custom field is of type "Date/Time". If you select **Static**, click the **Select Date** icon to display the Calendar and select the required date/time. If you select **Custom**, then you can specify a date/time relative to the current date/time such as 1 hour from now, or 3 hours ago.

f. (Optional) If you want to provide more information about the field, then click the **Add Tooltip** link and enter the description in the **Tooltip** field.

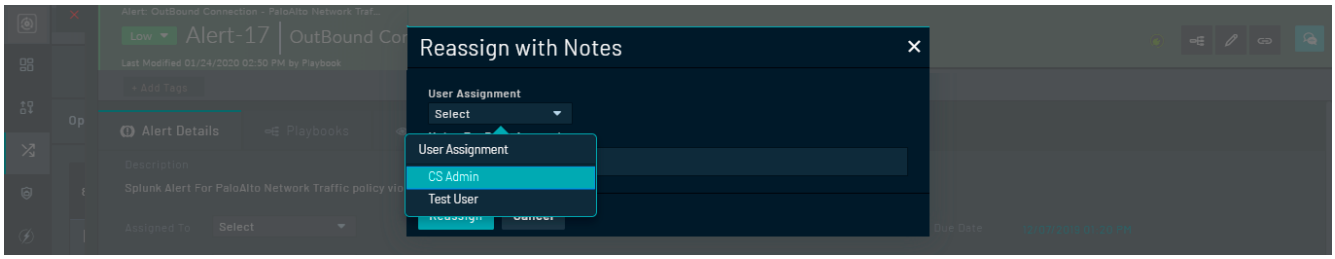
g. Starting with release 7.6.5, a new option, **+Add Required Condition** is added, which allows users to specify criteria when a field is made mandatory during step execution. The following options are available:
Not Required: The field is optional.
Required:The field is always mandatory.
For our example, select this option to ensure that the record cannot be reassigned to another user without adding a note.
Required by condition: The field becomes mandatory when defined conditions are met.

4. To change the name of the action button, which by default appears as Execute, update the **Submit Button Text** field, and type, for example, `Reassign`.

The `Input Prompt` section with all these changes will appear as shown in the following image:



Now, when you execute this playbook after selecting alert records and clicking **Execute** > **Reassign with Notes**, then the `Reassign with Notes` dialog is displayed. The `Reassign with Notes` dialog will contain the **Assigned To** drop-down list, with a list of users to whom this record can be reassigned, a rich text area where the user must add the reassignment notes, and the **Reassign** button which will execute this playbook, as shown in the following image:

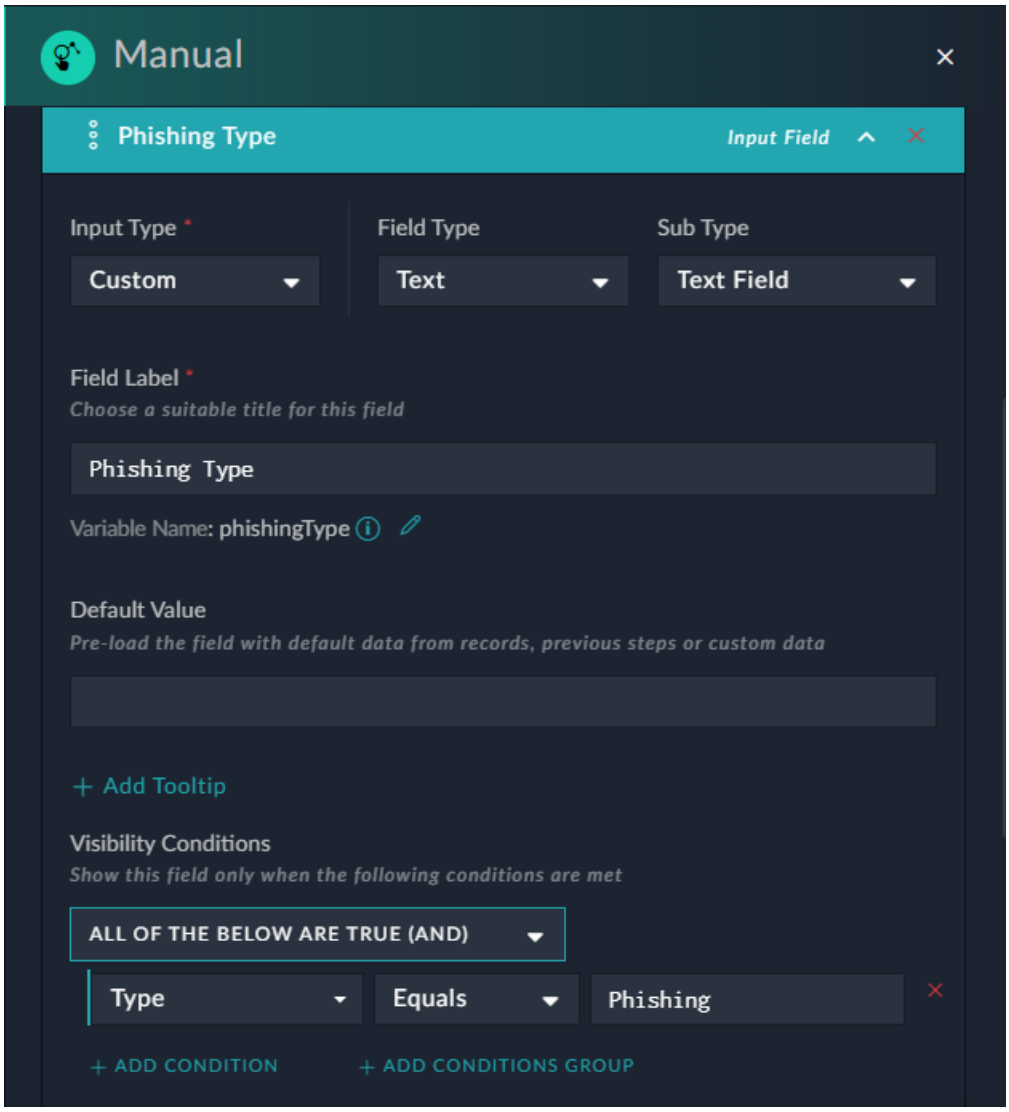


User Input Prompt - Visibility Conditions

You can add visibility conditions to the fields displayed in the user input form, i.e., fields in the user form would be visible based on the conditions you specify. You can define visibility conditions in user prompts both when you trigger the playbook using the *Manual Trigger* option and also during the execution of the playbook using the *Manual Input* step.

For example, when you trigger a playbook on an alert record, you could ask users to specify the type of alert, and you could define additional fields that would be visible if a particular type of alert is selected. For example, if the user selects the 'Phishing' as the alert type, then another field named 'Phishing Type' would be displayed, if the user selects 'Ransomware' as the alert type, then a field named 'Ransomware Type' would be displayed and so on. Based on the user selection, you can further define the playbook execution.
To add visibility conditions as the one described in the example, i.e., display an additional field based on the type of alert, in a User Prompt, do the following:

1. In the `User Prompt` section, click the down arrow, and then click **Add Field**.
2. To prompt the user to set the 'Type' for the alert, select the **Input Type** as **Record Field** . From the **Choose Record Field** drop-down list select **Type**, click the **Mark as Required Field In Prompt** checkbox, and in the **Field Label** field, enter `Type`.
3. Click **Add Field** to create additional fields based on the Type of alert the user selects. For example, to create a 'Phishing Type' field, from the **Input Type** drop-down list, select **Custom**, from the **Field Type** drop-down list, select **Text**, and from the **Sub Type** select **Text Field**. If you have created a picklist with the different Phishing Types, you can specify Picklist and choose the appropriate picklist. Next, click the **Mark as Required Field In Prompt** checkbox, and in the **Field Label** field, enter `Phishing Type`.
   Note: If you add a field as required, for which a visibility condition is defined, then that field is required only when its visibility condition is met, i.e., when the field is visible. For example, in the above step, the Phishing Type field is a required field, however, this field will be required only if the Type of alert is 'Phishing'.
   You can similarly add fields for various types of alerts, such as Ransomware, or Brute Force Attack, etc.
4. Add the visibility condition for the `Phishing Type` field by clicking **Add Visibility Conditions** and specifying the **Visibility Condition** as "`Type Equals Phishing`":



Similarly, you can other visibility conditions for various steps. To define a visibility condition there must be at least two steps in the user prompt.

5. Click **Save** to save your changes to the step, and then click **Save Playbook**.

When you run the playbook on a record, you will observe that if you select 'Phishing' as the type, the 'Phishing Type' field is displayed. If you select any other type, you will observe that no additional field is displayed.

User Input Prompt - Dynamic Lists

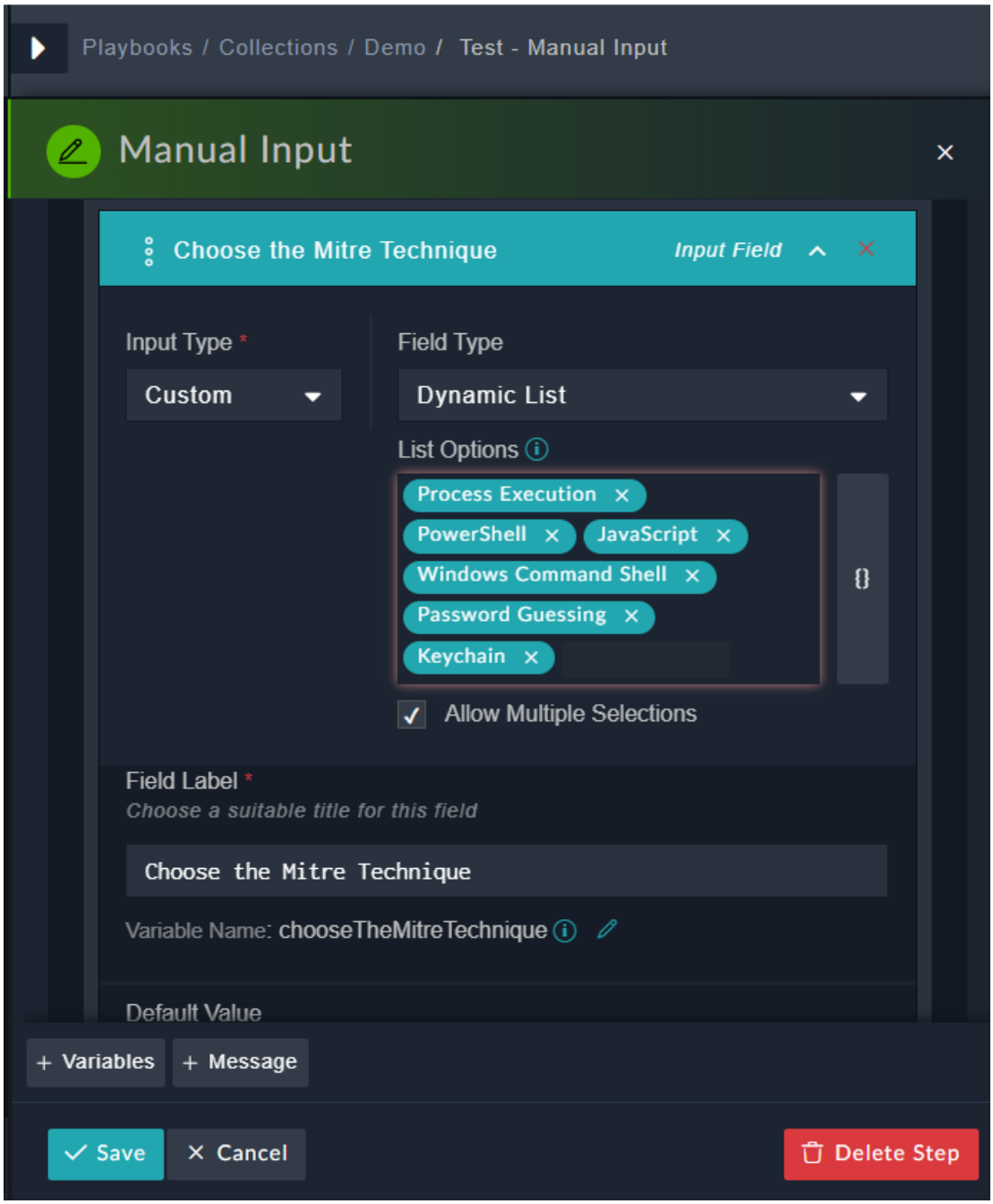
Dynamic List is supported as a '**Custom**' input type in both the Manual Trigger step and the Manual Input step. A 'Dynamic list' is a list with dynamic values that is set using a playbook, i.e., the options of the list are defined using comma-separated values are displayed as a list in a user input prompt. You could use dynamic lists in cases such as:

- **Independent Playbooks**: You might need to create a list for the manual trigger or manual input that can be automatically included as part with the exported playbook, i.e., the playbook step holds the logic of the items and therefore does not require a custom picklist to be exported.

- **Unauthenticated Picklists**: Picklists cannot be loaded in case of unauthenticated inputs. In such cases, pre-defined lists can be used to present multiple options to users for their selection.
- **Constantly Changing Picklists**: There could be picklists such as picklists based on MITRE threat hunting categories, which are ever changing. In such cases you can prompt users to categorize the threat based on the loaded list of categories. Then, based on the selected threat, you can automatically show the sub list for selecting techniques.
- **Filtered Record List**: You can present a filtered lists of records in a manual input to users for selection. The filter would vary depending on logic in the playbook.

For example, if you want to a MITRE techniques dynamic list, in which you can define the various MITRE tactics and then based on the users' selection, display the various techniques associated with the tactic. To add such a dynamic list to a User Prompt of a manual trigger or manual input, do the following:

1. In the `User Prompt` section, click the down arrow, and then click **Add Field**.
2. Select the **Input Type** as **Custom** and from the **Field Type** drop-down list, select **Dynamic List**.
3. Add various options to the dynamic list using comma-separated values in the **List Options** field, if you want users to choose from a list of options:

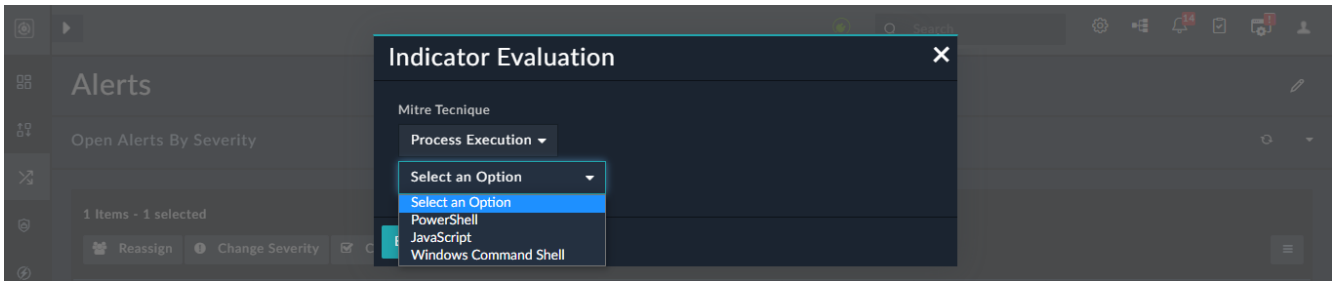


To allow users to select multiple options from the manual input prompt, select the **Allow Multiple Selections** option. For example, if you are displaying a list of indicators to analysts selecting this option enables them to choose multiple indicators.

To require users to choose at least one option from the list, select the **Mark as Required Field in Prompt** option.

4. In the **Field Label** field, add an appropriate title for the user prompt, such as `Choose the Mitre Technique`, then click **Save** to save your changes to the step, and then click **Save Playbook**.

When you run the playbook on a record, you will see that the Dynamic list will appear in the User prompt as shown in the following image:



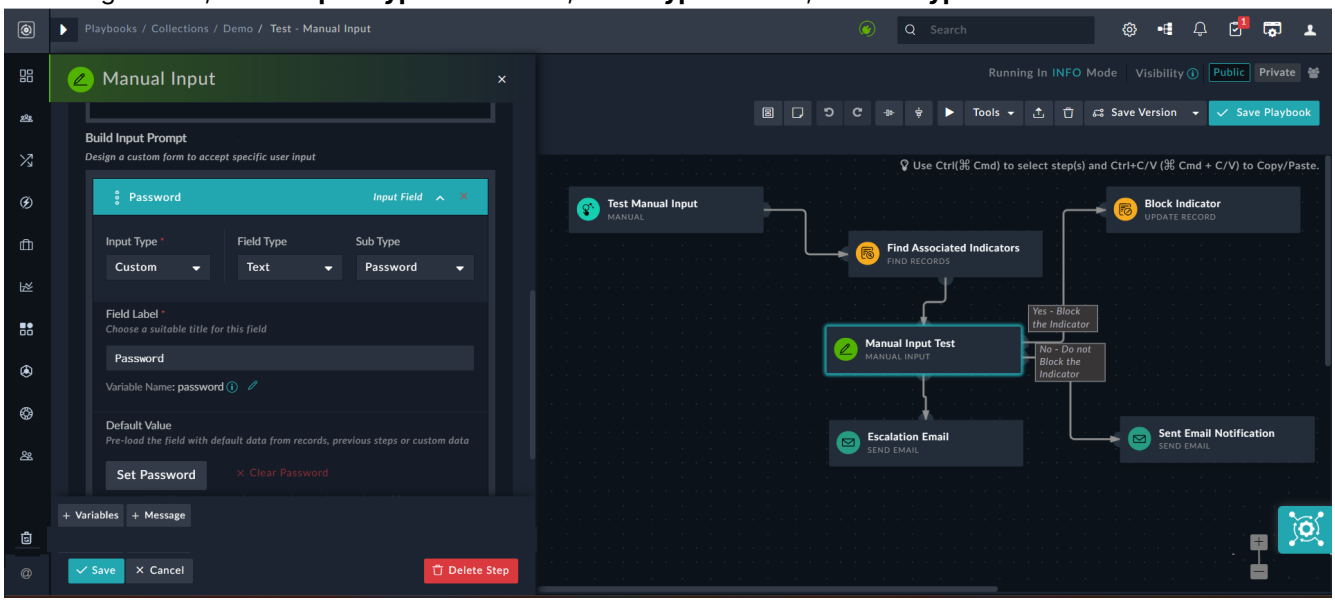
User Input Prompt - Custom date/time field usage notes

When you add a custom **Date/Time** field as an input parameter in an `Input Prompt`, then that Date/Time appears correctly in the Input Prompt, however any create record or update record that uses this custom date/time field will display the created/updated record as **01/01/1970**.

For example, in a Manual Trigger set on the `Alerts` module, when you add a custom **Due Date** field of type `Date/Time Field`, whose due date is set as **Current Date +1 Day**, and the step following the Manual Trigger step is a Create/Update Record step to create/update an alert record that uses the custom due date, then the alert record get created/updated with the Due Date set as 01/01/1970. This happens since the create/update record step requires the date/time in the epoch time, which is not the format in which currently the create record step is receiving the date/time. To fix this, in the create record step, in the **Due Date** field, add the following: `{{arrow.get(vars.input.params.dueDate.int_timestamp}}`. The `{{arrow.get(jinja varibale).int_timestamp}}` converts the value of the date/time field into the epoch date/time.

User Input Prompt - Using the Password field

Starting with release 7.6.5, users can securely provide passwords to FortiSOAR through a manual input prompt. To configure this, select **Input Type** as **Custom**, **Field Type** as **Text**, and **Subtype** as **Password**:



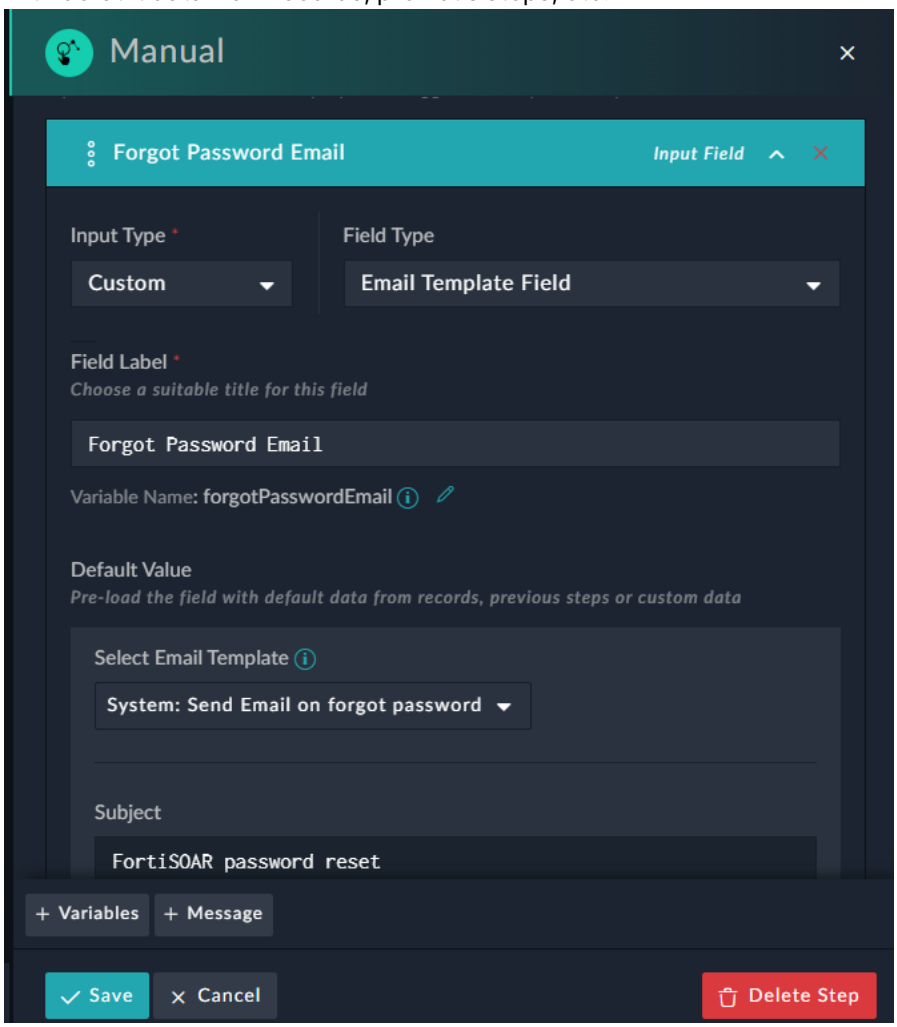
Previously, the only secure way to supply passwords was through connector configurations, which required access to connector pages. This enhancement allows users without connector-page access to securely enter credentials that can be used in automation, such as playbooks that create or update connector configurations.

User Input Prompt - Using the 'Email Template' Field

You can use the 'Email Template Field' as a 'Custom' input field type in the 'Manual Trigger' and 'Manual Input' steps only. The ability to include an email template makes it easier for SOC teams to respond to routine operations. For example, sending emails to users when they have forgotten their password. In this case, SOC teams create a template response to be sent to users, which can be included when you select the **Email Template Field** as a Custom Input.

In the `User Prompt` section, perform the following steps to use an email template as a field in a custom input prompt:

1. From the **Input Type** drop-down list, select **Custom**.
2. From the **Text** drop-down list, select **Email Template Field**.
3. Add the required parameters, such as specifying an appropriate title for the field, marking the field as a required field, etc.
4. In the `Default Value` section, from the **Select Email Template** drop-down list, select the appropriate email template.
   Once you select the email template, the fields of an email, such as Subject, Content, etc., get pre-populated with default data from records, previous steps, etc.



5. Change the default content, as per your requirements, in the **Subject** and **Content** fields. For example, you can change the signature in the **Content** field to your organization's signature from FortiSOAR Admin; and this updated subject and content is displayed to the user in the input form.
   Note: This change made in the email template in the playbook does not reflect in the email templates that are included by default in your FortiSOAR instance, i.e., the email templates present in **Settings** > **System Configuration** > **System Fixtures** do not get updated.
6. Once you have completed adding and updating all the parameters, click **Save** to save the step.

Recommendations

- **Define a Clear Trigger Name**: Use meaningful names in the Trigger Label Button field. This name appears in the Execute dropdown list, so it should clearly describe the action (e.g., *Investigate Phishing Email*, *Run Malware Scan*).
- **Select the appropriate execution behavior**:
  - **Run once for all selected records**: For batch operations where a single playbook instance handles all selected records. This approach reduces system resource usage and simplifies playbook logic.
  - **Run separately for each selected record**: For operations requiring one playbook instance per record.
  - **Does not require a record input to run**: For tasks Use this for tasks not tied to specific records. It is useful for administrative or monitoring actions—for example, manually checking for new alerts in the 'Alerts' module.
- **Register triggers only on relevant modules**: For example, a phishing investigation trigger should be added only to Email Incidents or Alerts, not Assets.
- **Use visibility conditions**: Display relevant playbooks based on record-specific attributes such as Type, Severity, Category, and Status. This helps reduce clutter and ensures users see only context-appropriate actions.
- **Create intuitive user prompts**: Use standard field types (Text, Picklist, Lookup) to collect input before playbook execution. Ensure prompts are clear, validated, and purposeful—designed to enhance playbook logic without adding unnecessary complexity.
- **Test execution behavior**: Use the Playbook Designer's debug tools to test both *run once* and *run separately* options.
  For multi-record selection testing, verify the input format (`vars.input.params.records`) and ensure the logic correctly handles multiple records to avoid unexpected results.
- **Add dedicated buttons**: Update module templates with custom buttons for frequently used manual triggers to enhance accessibility.

On Create

On Create triggers are intended for asynchronous execution, meaning they are non-blocking on the triggering data operation. For example, you can define a playbook that gets triggered when an Incident is created.

This trigger starts the execution of a playbook immediately after a record of the selected model type is created or ingested. Click **On Create Trigger** in the `Playbook Designer`, type the name of the step in the **Step Name** field and then select the module on whose creation you want to trigger the playbook, from the **Resource** drop-down list, for example, `Incidents`. In the 'Trigger Condition' section, you can add conditions based on which you want to trigger this playbook, and then click **Save**. For more information on conditions, see Condition-based Triggers.

Nested filters are also supported on the "On Create" and "On Update" triggers. Support has also been added for Less Than (Before in case of Date/Time fields), Lesser Than or Equal To (On or Before in case of Date/Time fields), Greater Than (After in case of Date/Time fields), Greater Than or Equal To (On or after in case of

Date/Time fields), and Matches Pattern operators in filters. For more information about nested filters and operators, see the Working with Template Widgets topic in the "User Guide."

Playbooks with the 'On Create' trigger will not work in the case records are ingested using the 'Ingest Bulk Feed' playbook step for those created records.

In release 7.4.2, the **Only trigger at the node where record is** checkbox added to the "On Create," "On Update," or "On Delete" playbooks, which allows users to choose if they want to execute these playbooks either on the instance where the record is created or where it is replicated, or on both instances. For more information, see the 'Distributed Tenancy Support' chapter in the *Multi-tenancy Support in FortiSOAR* guide.

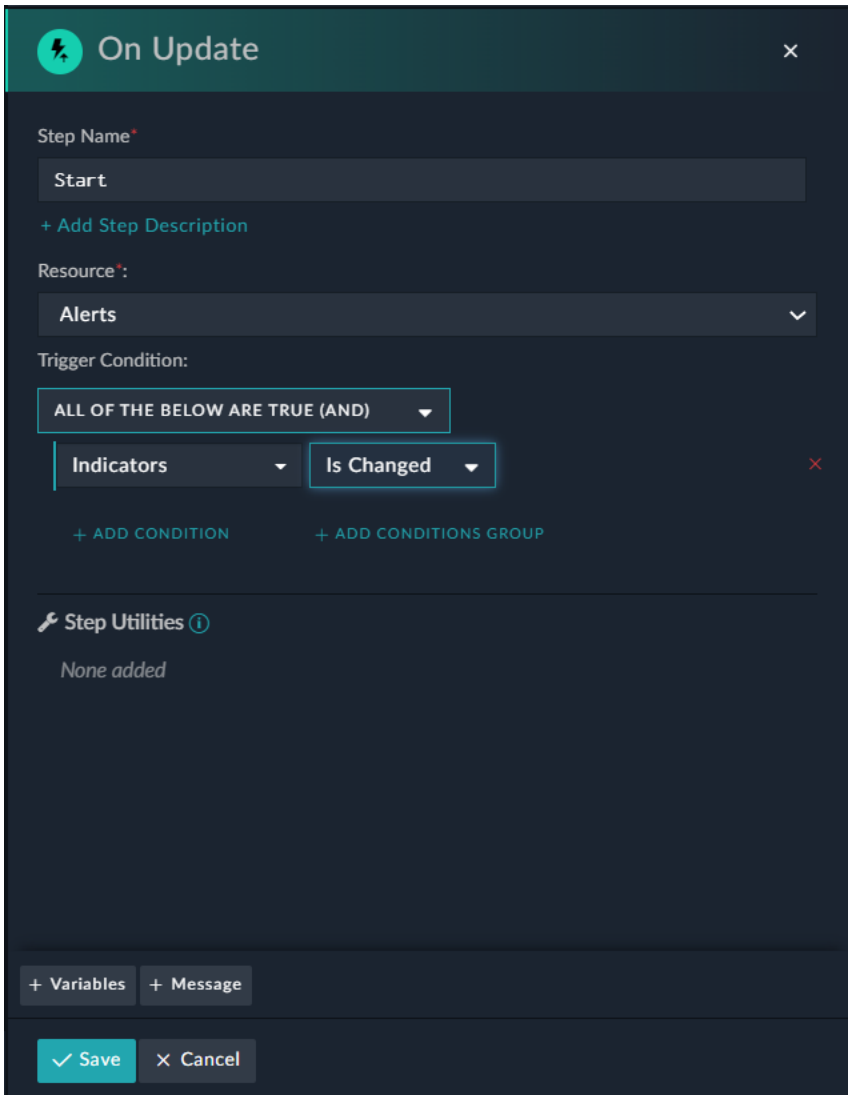
Notes & Best Practices:

- **Use filters to target relevant records**: For example, to triage new incidents, filter by severity, type, and source.
- **Use Nested Filters**: Complex logic is supported.
  Example: `(Severity = High AND Status = New) OR (Owner = SOC))`.
- **Supported Filter Operators**: Advanced operators are supported, including:
  - Less Than / Before (for Date/Time)
  - Less Than or Equal To / On or Before
  - Greater Than / After
  - Greater Than or Equal To / On or After
  - Matches Pattern (for regex-like string matching)
    For details on filters and operators, see the Working with Template Widgets topic in the "User Guide."
- **Avoid over-triggering**: Always use specific conditions to prevent unnecessary executions.
- **Bulk Inserts considerations**: Playbooks with On Create triggers do not run when records are created using the Ingest Bulk Feed step. These programmatic inserts intentionally bypass the trigger.
- **Testing considerations**: Thoroughly test triggers in development environments to ensure asynchronous behavior does not negatively impact workflows.
- **Distributed environments**:Use "Only trigger at the node where record is..." to control trigger behavior in multi-node or cluster setups. For details, see the Distributed Tenancy Support chapter in the "Multi-Tenancy Support Guide."

On Update

This trigger starts the execution of a playbook immediately after a record of the selected model type is updated. You can create an On Update trigger on almost all models, and can add an On Update trigger in the same way you add an On Create trigger. An update could be made to any field within the model, **including linking or changing one or more new relationships**.

When you add the **On Update** trigger to run on a **Is Changed** condition when relation fields are changed, such as indicators for alerts, then the **On Update** trigger will trigger the playbook only when the related record is linked from the same side. For example, while linking an indicator to an alert, the relation can be formed both ways – by updating the indicator record and linking the alert; or by updating the alert record and linking the indicator.



However, an On Update trigger on an alert when indicator 'Is Changed' will only be triggered if the *indicator was linked by updating the alert record*. It will not be triggered when the relation is established while creating or updating an indicator record.

The single update action defines the trigger, so linking multiple records or updating multiple fields simultaneously does not trigger the playbook multiple times. However, multiple inline edits trigger the playbook multiple times. A bulk edit action triggers the Playbook only once.

You can also add conditions based on which you can trigger this playbook. For more information, see Condition-based Triggers.

Playbooks with the 'On Update' trigger will not work in the case records are ingested using the 'Ingest Bulk Feed' playbook step for those updated records.

In release 7.4.2, the **Only trigger at the node where record is** checkbox added to the "On Create," "On Update," or "On Delete" playbooks, which allows users to choose if they want to execute these playbooks either

on the instance where the record is created or where it is replicated, or on both instances. For more information, see the 'Distributed Tenancy Support' chapter in the *Multi-tenancy Support in FortiSOAR* guide.

Key Points:

- Fires on any field update—but triggers only once per update, even if multiple fields are changed.
- Triggers on relationship changes (e.g., an indicator linked to an alert)—but only from the module specified.
  Example:
  An On Update trigger on Alert for "Indicator is changed" runs only if the alert is updated, not the indicator.
- Multiple inline edits trigger multiple executions; bulk edits trigger only once.
- Does not trigger on updates via the Ingest Bulk Feed step.
- For distributed tenancy, use "Only trigger at the node where record is..." to control where the playbook runs. For details, see the Distributed Tenancy Support chapter in the "Multi-Tenancy Support Guide."

Referenced

The Referenced trigger is intended for playbooks that are exclusively called from a `Reference a Playbook` step, which is discussed in a later section. Bear in mind that any dynamic data requirements must be made available from the Parent (s) playbooks to be used during the execution of a Child playbook.

To add Referenced step, click **Referenced** in the `Playbook Designer`, type the name of the step in the **Step Name** field and click **Save**.

Notes:

- **Dynamic data handling**: The parent playbook must provide all required inputs—using input/output variables or dynamic data—before executing the referenced playbook. These variables are configured in the parent and passed into the referenced playbook step to ensure the necessary context for execution.
- **Return results**: Child playbooks can return data or status to the parent, enabling conditional logic or follow-up actions.
- **Use Trigger Blocks**: Create reusable 'Trigger Blocks' that begin with a trigger step to avoid duplicating logic.

Custom API Endpoint

Custom API Endpoint Triggers allow you to specify an arbitrary endpoint that can be used to externally start a playbook using a REST API POST action from another system. All playbooks are triggered using an API on a technical level with the microservices architecture used by the application, but conceptually, this trigger allows for the creation of an endpoint explicitly for use in API-based operations.

The chief aim of the Custom API Endpoint Trigger option is to allow for easy ingestion of data. A RESTful POST method explicitly defined by the authentication method is allowed to trigger a playbook to the defined endpoint. The endpoints of the Custom API Endpoint trigger are not discoverable, unlike the standard API routes within the JSON-LD / Hydra definition. You must know the endpoint name explicitly, and it currently only allows the POST method.

The endpoint name can be any valid name using alphanumeric characters. You should not use special characters in naming the endpoint, or the endpoint might not function correctly.

The following three types of authentication are currently supported:

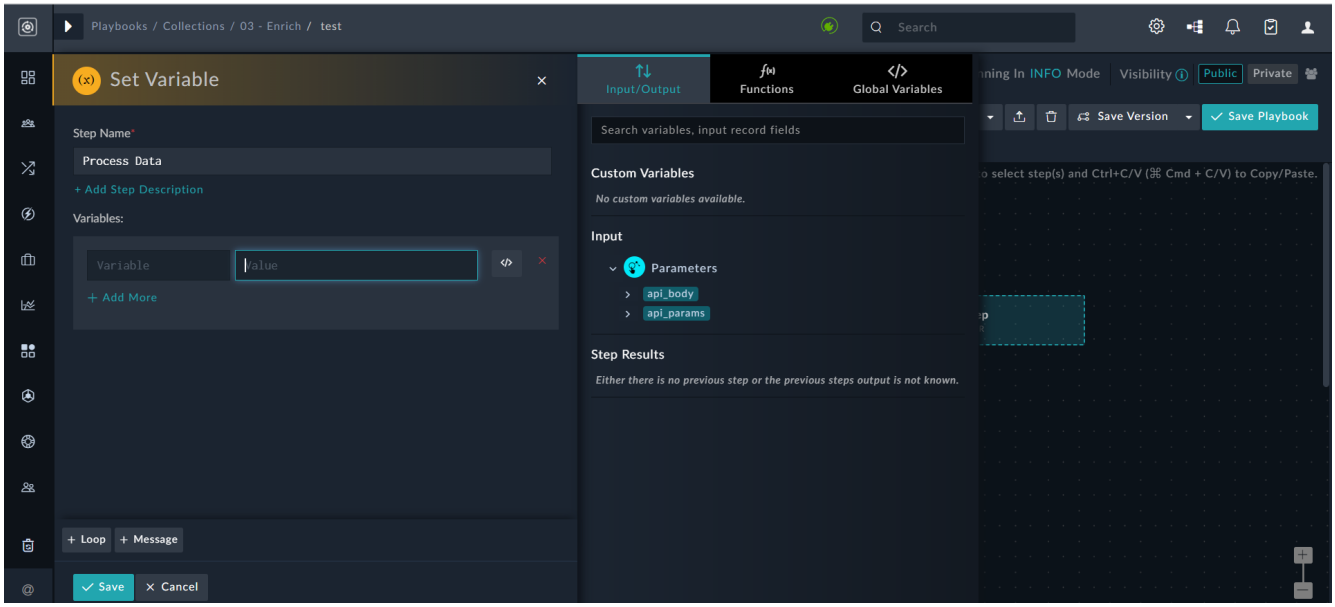
1. Token-based or API-Key based (default) Authentication- the default API method for signing any API request.
   Note: For token-based (HMAC) authentication the timestamp must be in UTC format.

2. Basic Authentication - a Base64 encoded version of the `username:password` present in the header. This requires the username and password of a user without 2-Factor Authentication turned on to properly function. Note that this method also uses a separate endpoint.
3. No Authentication (Not recommended) - no authentication method is applied to the endpoint, and any RESTful POST method will trigger the playbook. This is chiefly aimed at applications where the only option for exporting data is by using a webhook, but this method is not recommended for routine usage due to the lack of security.

You can manually create your own security method with this trigger by defining a specific criterion to be used in a Decision Step verifying information in the full Request blob.

To add an Custom API Endpoint trigger, click **Custom API Endpoint** trigger in the `Playbook Designer`, type the name of the step and the API route in the **Step Name** and **Route** fields respectively, and then select the `Authentication Method` from the ones specified earlier and click **Save**.

You can also create a playbook with a Custom API Endpoint trigger and invoke a custom endpoint to get some input from outside FortiSOAR and then further process that data in the next steps of the playbook. Using Dynamic Values you can access query params (`api_params`) or request body (`api_body`).



Highlights:

- **No discoverability**: Custom endpoints do not appear in auto-generated API documentation and require the exact endpoint name to be known.
- **Authentication Options**: Different types of authentication can be configured to secure the custom API endpoint:
  - **Token-based (API Key) Authentication** (preferred) — ensures secure communication.
  - **Basic Authentication**
  - **No Authentication** — not recommended for production environments due to security risks.
    After setup, the endpoint is invoked by sending a POST request to the defined URL. The playbook processes incoming data using steps such as Decision, Data Transformation, or Integration. Dynamic Values access query parameters (`api_params`) or the request body (`api_body`) to facilitate data processing.

On Delete

This trigger starts the execution of a playbook immediately after a record of the selected model type is deleted. You can create an On Delete trigger on almost all models, and can add an On Delete trigger in the same way you add an On Create trigger.

You can also add conditions based on which you can trigger this playbook. For more information, see Condition-based Triggers.

In release 7.4.2, the **Only trigger at the node where record is** checkbox added to the "On Create," "On Update," or "On Delete" playbooks, which allows users to choose if they want to execute these playbooks either on the instance where the record is created or where it is replicated, or on both instances. For more information, see the 'Distributed Tenancy Support' chapter in the *Multi-tenancy Support in FortiSOAR* guide.

Considerations:

- Use filters to narrow execution scope (e.g., only delete events for closed incidents).
- For distributed tenancy, use "Only trigger at the node where record is..." to define where the playbook should run (original node, replica, or both). For details, see the Distributed Tenancy Support chapter in the "Multi-Tenancy Support Guide."

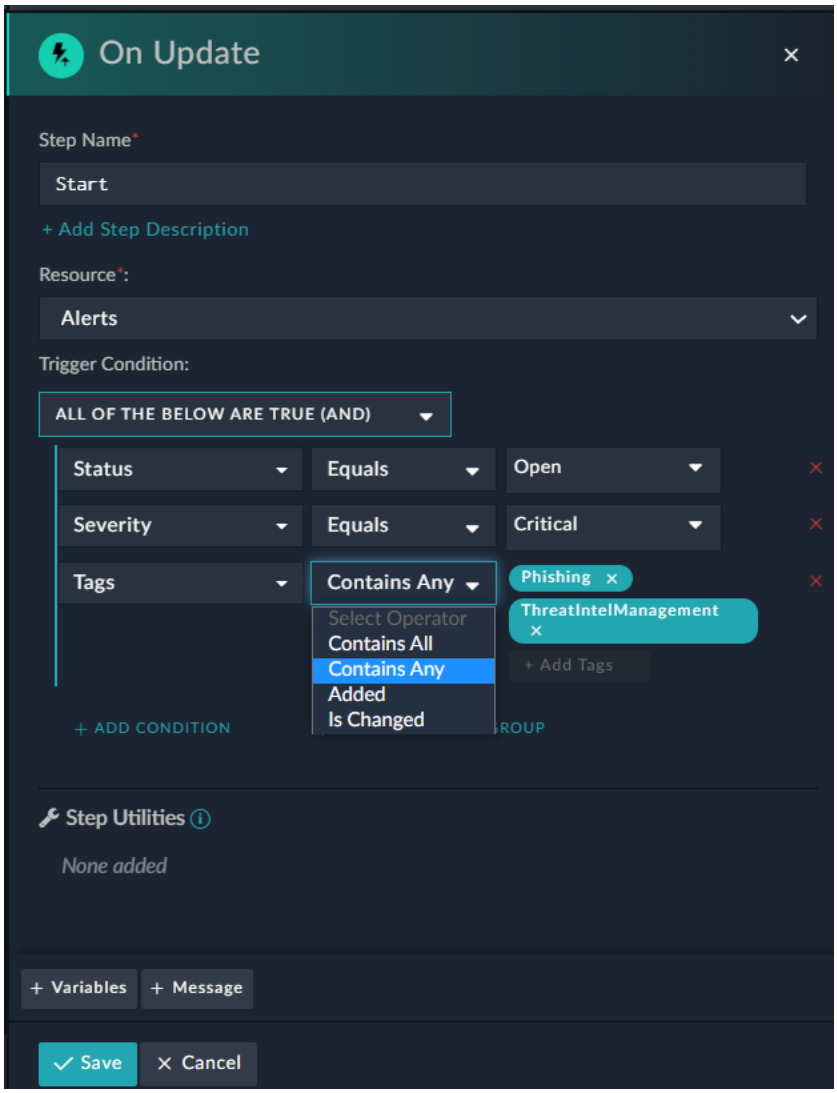
Condition-based Triggers

You can define a condition or nested conditions to trigger a playbook only if the specified filter criteria are met. This streamlines playbook calls and prevents the excessive calling of playbooks.

You cannot apply filters on encrypted fields.

Open the playbook designer and click on On Create, On Update, or On Delete trigger. For example, click **On Update** trigger and then select the module, which when updated will trigger the playbook, from the **Resource** drop-down list, for example, `Alerts`. Once you select the resource, a **Trigger Condition** drop-down list appears. To define the condition based on which the decision to trigger the playbook will be taken, perform the following steps:

1. From the **Trigger Condition** drop-down list, select the logical condition, **All of the below are True (AND)**, or **Any of the below is True (OR)** to trigger the playbook.
   In case of the **AND** condition the playbook gets triggered only if all the conditions specified are met. In case of the **OR** condition the playbook gets triggered if any of the conditions specified are met. The **AND** or **OR** conditions are mutually exclusive, i.e., you can only choose one of them to apply to conditions.
2. Click the **Add Condition** link and then build your condition.
   Note: There is an additional operator **Is Changed** added for the trigger condition. If you select the **Is Changed** operator for a field, then the playbook will be triggered whenever the specified field is changed. For example, if you want to assign **Critical** alerts that are in the **Open** state to a specific user, say `csadmin`, then you can select the **Severity** field and choose the operator as **Equals** and specify **Critical**.
   Click **Add Condition** to define other conditions such as selecting the **Status** field and choosing the operator as **Equals** and specifying **Open**, and then also adding tags, as shown in the following image:



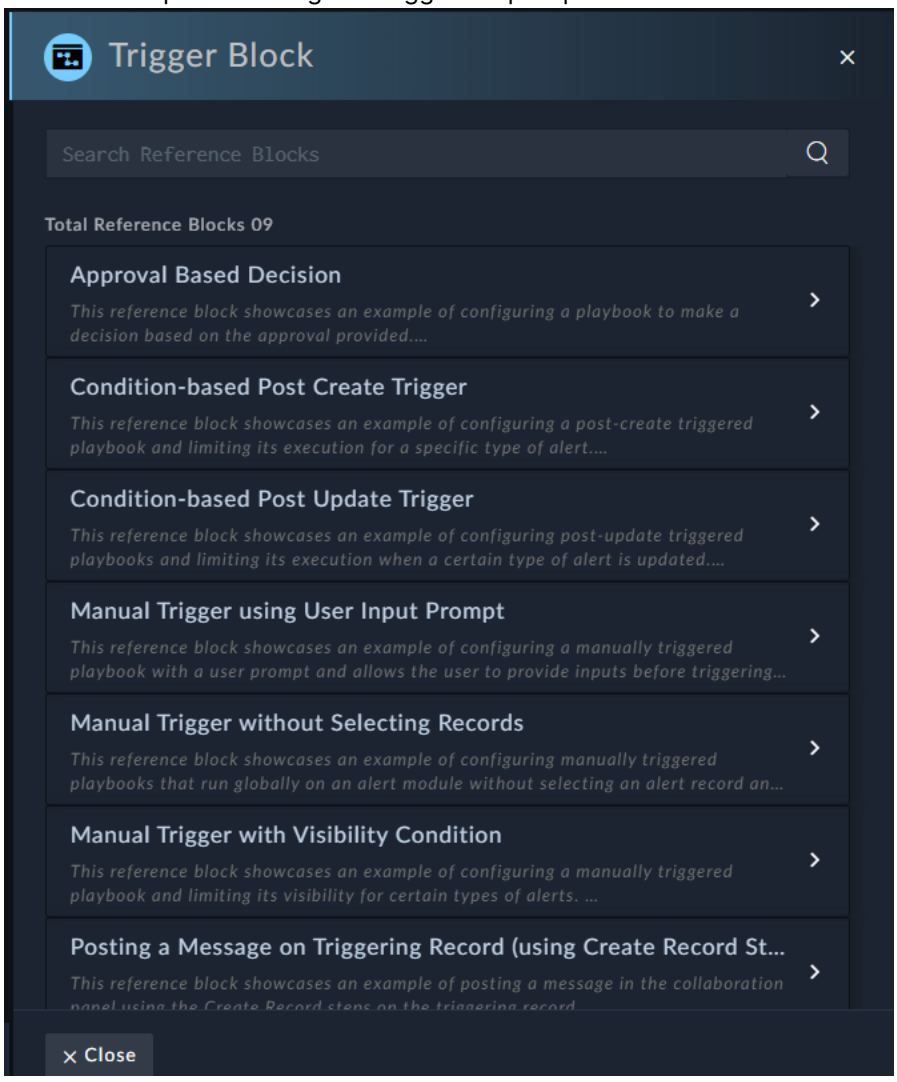
Once you complete adding the conditions, click **Save** to save the playbook.
In this case, once the condition is met, the On Update playbook will be triggered, and based on the steps that you have defined, for example, the `Update Record` step, the alert will be updated and assigned to `csadmin`.
Important: You can also use `Tags` as a condition to trigger the On Create, On Update, or On Delete playbooks. You can add special characters and spaces in tags; however, the following special characters are not supported in tags: `'`, `,` , `"`, `#`, `?`, and `/`. The operators that you can use with **Tags** in the **On Create** and **On Delete** triggers are `Contains`. The operators that you can use with the **On Update** trigger are `Contains All`, `Added`, or `Is Changed`.

If you want to add a group of conditions, then click the **Add Conditions Group** link. For example, if you wanted to create a condition where the alerts have been created in the last calendar month and whose severity is critical and whose status is open or investigating, in such a case you could create a condition group for the status condition. For more information about nested filters and operators that can be used in conditions, see the Working with Template Widgets topic in the "User Guide."

Trigger Block

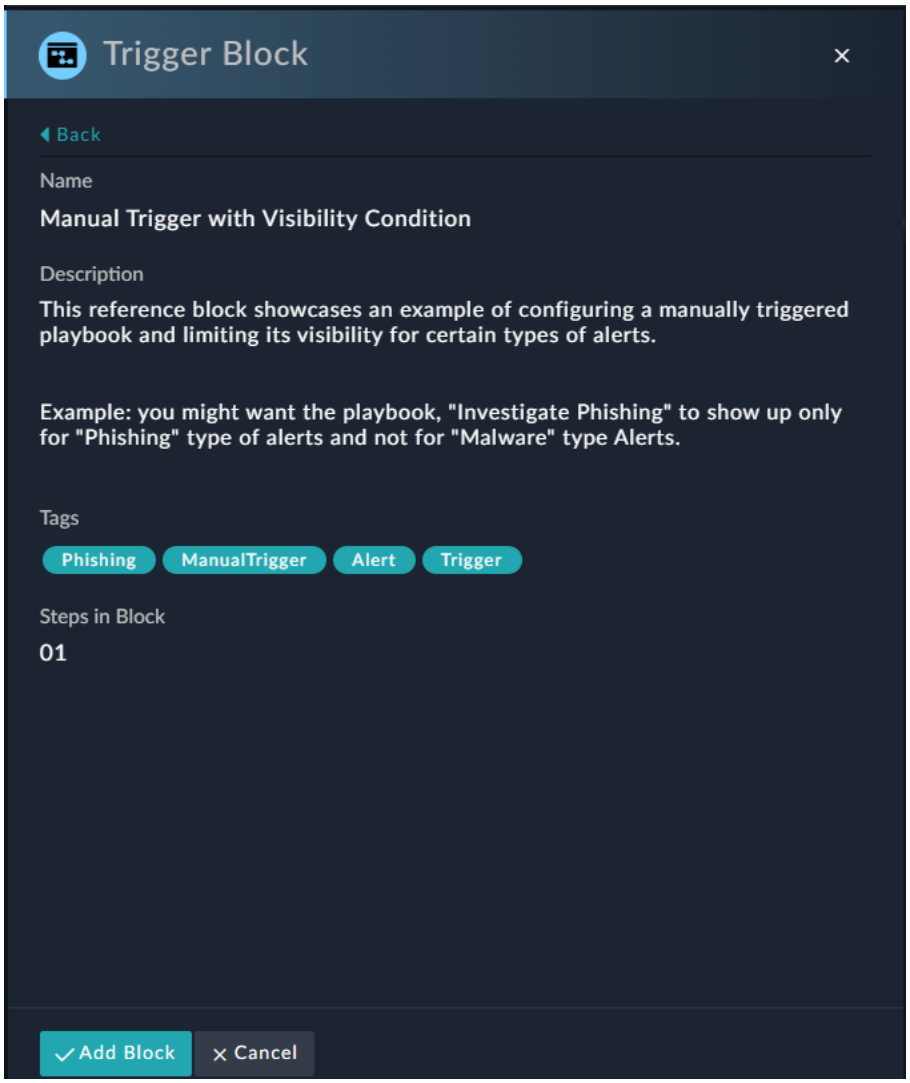
For commonly used playbook blocks that contain the trigger step, use the **Trigger Block** step. The **Trigger Block** step contains ready-made blocks (steps) included with the SOAR Framework solution pack, or blocks that you have created based on the steps mentioned in the *Adding Blocks and Notes* topic, which make it easier for users to get relevant references and samples for various types of available playbook triggers, including ones with visibility conditions, API endpoints triggers, etc. Blocks that include the start (or trigger) step get added as 'Trigger Blocks'.

To add a Trigger Block to a playbook, click **Trigger Block** in the Playbook Designer, which displays a list of all reference steps containing the 'trigger' step as part of the reference block:

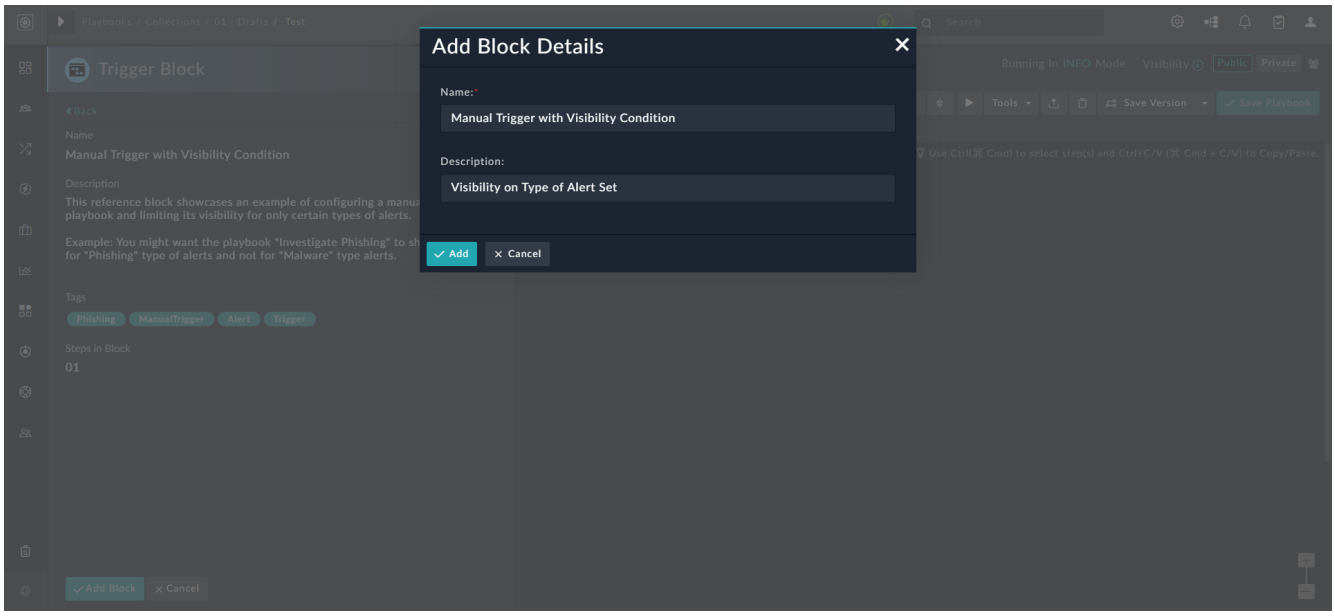


Use the **Search Reference Blocks** field to search for reference blocks using tags (exact match supported), the name of the block, or its description.

Click **>** or click anywhere in the block row to expand a particular block to get information about the block; information includes the description, tags, and number of steps in that particular block. To add a block to your playbook, select the block and click **Add Block**.



Clicking **Add Block** displays the `Add Block Details` dialog where you can edit the name and description for the block and then click **Add**, which adds the block in your playbook designer:



The block is added to the playbook designer with the name and description, i.e., this is the content that will be displayed when you hover on the **Info** icon that you have specified in the `Add Block Details` dialog:

You can edit the added trigger block; however, note that editing the block updates only the step(s) in the current playbook and does not update the steps(s) in the reference (original) block. Clicking the **Save as Reference Block** icon in the current playbook displays the `Save as Reference Block` dialog, where you can make changes to the block and save the trigger block with a unique name. To make any changes to the name or description of the reference (original) block, or update its tags, you must use **Tools** > **Reference Blocks** option.

Trigger Payloads and Data Structure

Within the context of dynamic variables, the trigger step allows access to all data within the inbound transaction using the Dynamic Value prefix within the Jinja2 template formatting, for example, `{{vars}}`.

> See the Dynamic Variables and Dynamic Values chapters for more information on using Dynamic Variables within a Playbook environment.

Standard information that is packaged includes, but is not limited to, the following:

Key	Information Type	Applies To

`auth_info`	This displays the type of authentication invoked by the user who triggered the playbook. It can be no authentication, basic authentication, or CS HMAC authentication.	All
`currentUser`	The IRI of the current user who triggered the playbook.	All
`last_run_at`	The last date of execution for a playbook that is run on a periodic basis.	Scheduled
`request.base_uri`	The root of the host URI on which the playbook is executing, for example, `https://fortisoar.sampleurl.com`	All
`request.uri`	The full URI route of the API endpoint used to invoke the playbook.	All
`input.records`	An array of records under the operation. For post-create, post-update, and manual triggers that have a single records, the array contains only one record that can be accessed using `input.records[0]`.	Manual trigger, Post-Create, and Post-Update triggers
`input.params['api_body']`	The payload of the request in case of the custom API endpoint trigger.	`API` trigger only
`input.params.<param_name>`	Inputs that are specified using the `Input Parameters` option in a playbook.	All
`request.headers`	All the headers sent with the request that invoked the playbook.	All
`request.headers['X-RUNBYUSER']`	The IRI of the current user who triggered the playbook.	All
`previous`	Specific to the `Update` trigger. It shows the original version of the record data before being changed.	`Update` trigger only
`resource`	The module targeted by the playbook.	Database triggers
`request`	The full request object that initiated the playbook.	All
`request.data`	The cleaned data, if in JSON format, associated with the `request.body`.	All
`request.method`	The RESTful method by which the playbook was triggered, only `POST` or `PUT`.	All

Database Triggers (On Create, On Update, and On Delete)

In the case of a database trigger, such as On Create, the record which triggered the playbook is included within the API request and is accessible. The format of the record data will be identical to the format accessible within the standard Module endpoint for that record type.

Sample Data

Standard keys for data available within the `vars.input.records[0]` includes the following when it comes from an internal trigger, such as a On Create. The `%` indicates a placeholder for data that would be present in a real request in the general format.
The following sample data represents an alert. Other modules, such as Incidents, contain data based on their respective module templates:


```
{
        "id": 428214,
        "@id": "/api/3/alerts/b13bcaa8-e255-4e70-84f4-5f79aba79e66",
        "url": null,
        "name": "Test Alert",
        "rule": null,
        "type": {
          "id": 227,
          "@id": "/api/3/picklists/574a6ee2-7265-4701-815e-cff83b053bce",
          "icon": null,
          "uuid": "574a6ee2-7265-4701-815e-cff83b053bce",
          "@type": "Picklist",
          "color": null,
          "listName": "/api/3/picklist_names/a7087b9c-5660-495f-a8ac-c3b198ddb8c2",
          "itemValue": "Other / Unknown",
          "importedBy": [],
          "orderIndex": 19
        },
        "uuid": "b13bcaa8-e255-4e70-84f4-5f79aba79e66",
        "@type": "Alert",
        "dueBy": null,
        "state": {
          "id": 198,
          "@id": "/api/3/picklists/a1bac09b-1441-45aa-ad1b-c88744e48e72",
          "icon": null,
          "uuid": "a1bac09b-1441-45aa-ad1b-c88744e48e72",
          "@type": "Picklist",
          "color": null,
          "listName": "/api/3/picklist_names/2f9ed741-25fe-475a-9f12-64336288eebf",
          "itemValue": "New",
          "importedBy": [],
          "orderIndex": 0
        },
        "tasks": [],
        "alerts": [],
        "assets": [],
        "domain": null,
        "events": [],
        "source": null,
        "status": {
          "id": 201,
          "@id": "/api/3/picklists/7de816ff-7140-4ee5-bd05-93ce22002146",
          "icon": null,
          "uuid": "7de816ff-7140-4ee5-bd05-93ce22002146",
```

```
          "@type": "Picklist",
          "color": null,
          "listName": "/api/3/picklist_names/33e964a9-d607-49f2-813c-7ce46141815a",
          "itemValue": "Open",
          "importedBy": [],
          "orderIndex": 1
        },
        "ackDate": null,
        "emailTo": null,
        "metrics": null,
        "persons": [],
        "@context": "/api/3/contexts/Alert",
        "comments": [],
        "fileHash": null,
        "fileName": null,
        "filePath": null,
        "fsrQueue": null,
        "reporter": null,
        "respDate": null,
        "services": null,
        "severity": {
          "id": 445,
          "@id": "/api/3/picklists/58d0753f-f7e4-403b-953c-b0f521eab759",
          "icon": null,
          "uuid": "58d0753f-f7e4-403b-953c-b0f521eab759",
          "@type": "Picklist",
          "color": "#28B35C",
          "listName": "/api/3/picklist_names/4e80cba3-032f-48b4-ac03-17e3ec247aac",
          "itemValue": "Low",
          "importedBy": [],
          "orderIndex": 1
        },
        "sourceId": null,
        "sourceIp": null,
        "ticketID": null,
        "userName": null,
        "warrooms": [],
        "campaigns": [],
        "companies": [],
        "deviceUID": null,
        "emailBody": null,
        "emailFrom": null,
        "epochTime": null,
        "eventTime": null,
        "fileEmail": null,
        "fileNames": null,
        "incidents": [],
        "processId": null,
        "assignedTo": null,
        "bucketName": null,
        "createDate": 1762858136.792089,
        "createUser": {
```

```
          "id": 3,
          "@id": "/api/3/people/3451141c-bac6-467c-8d72-85e0fab569ce",
          "type": null,
          "uuid": "3451141c-bac6-467c-8d72-85e0fab569ce",
          "@type": "Person",
          "email": "admin@example.com",
          "title": "Admin",
          "avatar": null,
          "userId": "24821c40-fb2d-4bee-a555-e5bd33253464",
          "csActive": true,
          "lastname": "Admin",
          "phoneFax": null,
          "userType": null,
          "@settings": "/api/3/user_settings/3451141c-bac6-467c-8d72-85e0fab569ce",
          "companyId": null,
          "firstname": "CS",
          "phoneHome": null,
          "phoneWork": "+16462759691",
          "accessType": "Named",
          "createDate": 1760077170.046849,
          "createUser": "/api/3/people/3451141c-bac6-467c-8d72-85e0fab569ce",
          "department": null,
          "modifyDate": 1760077170.046849,
          "modifyUser": "/api/3/people/3451141c-bac6-467c-8d72-85e0fab569ce",
          "description": null,
          "phoneMobile": null
        },
        "deviceType": null,
        "fileHashes": null,
        "indicators": [],
        "modifyDate": 1762858136.792089,
        "modifyUser": {
          "id": 3,
          "@id": "/api/3/people/3451141c-bac6-467c-8d72-85e0fab569ce",
          "type": null,
          "uuid": "3451141c-bac6-467c-8d72-85e0fab569ce",
          "@type": "Person",
          "email": "admin@example.com",
          "title": "Admin",
          "avatar": null,
          "userId": "24821c40-fb2d-4bee-a555-e5bd33253464",
          "csActive": true,
          "lastname": "Admin",
          "phoneFax": null,
          "userType": null,
          "@settings": "/api/3/user_settings/3451141c-bac6-467c-8d72-85e0fab569ce",
          "companyId": null,
          "firstname": "CS",
          "phoneHome": null,
          "phoneWork": "+16462759691",
          "accessType": "Named",
          "createDate": 1760077170.046849,
```

```
      "createUser": "/api/3/people/3451141c-bac6-467c-8d72-85e0fab569ce",
      "department": null,
      "modifyDate": 1760077170.046849,
      "modifyUser": "/api/3/people/3451141c-bac6-467c-8d72-85e0fab569ce",
      "description": null,
      "phoneMobile": null
    },
    "recordTags": [],
    "returnPath": null,
    "sourcePort": null,
    "sourceTool": null,
    "sourceType": null,
    "sourcedata": null,
    "attachments": [],
    "commandLine": null,
    "description": null,
    "deviceOwner": null,
    "iPAddresses": null,
    "latestLogin": null,
    "processGuid": null,
    "processName": null,
    "processTree": null,
    "registryKey": null,
    "respDueDate": null,
    "targetAsset": null,
    "userDetails": null,
    "ackSlaStatus": {
      "id": 441,
      "@id": "/api/3/picklists/72979f64-e8b9-4888-a965-957e0ec24818",
      "icon": null,
      "uuid": "72979f64-e8b9-4888-a965-957e0ec24818",
      "@type": "Picklist",
      "color": "#808080",
      "listName": "/api/3/picklist_names/fe36a8f2-fcba-4221-b4ab-1081f596b153",
      "itemValue": "Awaiting Action",
      "importedBy": [],
      "orderIndex": 2
    },
    "assigneddate": null,
    "closureNotes": null,
    "computerName": null,
    "emailHeaders": null,
    "emailSubject": null,
    "resolveddate": null,
    "senderDomain": null,
    "closureReason": null,
    "destinationID": null,
    "destinationIp": null,
    "mitreattackid": null,
    "respSlaStatus": {
      "id": 441,
      "@id": "/api/3/picklists/72979f64-e8b9-4888-a965-957e0ec24818",
```

```
          "icon": null,
          "uuid": "72979f64-e8b9-4888-a965-957e0ec24818",
          "@type": "Picklist",
          "color": "#808080",
          "listName": "/api/3/picklist_names/fe36a8f2-fcba-4221-b4ab-1081f596b153",
          "itemValue": "Awaiting Action",
          "importedBy": [],
          "orderIndex": 2
        },
        "scheduledTask": null,
        "sourceProcess": null,
        "targetProcess": null,
        "ackSLApausedon": null,
        "communications": [],
        "deviceSerialNo": null,
        "killChainPhase": null,
        "mitreTechnique": null,
        "priorityWeight": 1,
        "attachmentNames": null,
        "destinationPort": null,
        "respSLApausedon": null,
        "bytesTransferred": null,
        "escalationReason": null,
        "parentProcessPid": null,
        "registryKeyValue": null,
        "parentProcessName": null,
        "reporterEmailBody": null,
        "alertDetectionDate": null,
        "decodedCommandLine": null,
        "senderEmailAddress": null,
        "emailClassification": null,
        "escalatedtoincident": {
          "id": 355,
          "@id": "/api/3/picklists/2128a09c-153d-4db3-985d-de6be33deae5",
          "icon": null,
          "uuid": "2128a09c-153d-4db3-985d-de6be33deae5",
          "@type": "Picklist",
          "color": "#808080",
          "listName": "/api/3/picklist_names/e131cc54-9ae4-43c9-b830-af6e331fe024",
          "itemValue": "No",
          "importedBy": [],
          "orderIndex": 1
        },
        "resolvedAutomatedly": false,
        "alertRemainingAckSLA": 0,
        "parentProcessCmdLine": null,
        "alertRemainingRespSLA": null,
        "recipientEmailAddress": null,
        "vulnerabilitySeverity": null
      }
```

As part of consolidating inputs for various types of triggers, all request parameters for all the different types of triggers have been consolidated under `vars.input`. For On Create, On Update, or Manual triggers, the record details are available under `{{ vars.input.records }}`. For API triggers, the request data is available under `{{ vars.input.params['api_body'] }}`. To avoid data duplication, `vars.request.data` is being deprecated in FortiSOAR 6.0.0 and it is recommended to use the above equivalents under `{{ vars.input }}`.

Manual Triggers

The Manual trigger payloads have a similar structure to the database triggers, and the payloads of both manual and database triggers are accessible using `vars.input.records`. The `records` array is an array of JSON objects, one object for each record that was passed in as a part of the request.

For instance, if you click the **Execute** button on the grid by selecting five record checkboxes, the data from all five records will be included in the `records` array in their raw format.

The Manual Trigger step also provides you with options to specify whether the action must be Executed **Once** or **For Each Record**. This enhancement makes it more effective to handle multiple selections since now you do not require to write two playbooks and map the second playbook in the first playbook. For more information, see Manual Trigger.

Custom API Endpoint Triggers

Internal triggers will always have a JSON format, but Custom API Endpoint triggers are initiated from external systems and might not always come in JSON format. Currently, custom Custom API Endpoint triggers can accept any format of the inbound body data, but this data might not be accessible within the environment in a structured way.

As an example, an XML request is not available in the environment until it has been parsed by a separate step. This can be done any time after the trigger step but must be done before referencing any variables that would be expected out of the XML structure.

XML can have a more sophisticated data structure than JSON and therefore, might require custom parsing for correct handling of XML data. A custom parsing step to convert XML to a dictionary format is present in the Utilities connector, "Convert XML to Dictionary".

Referenced Trigger

The Referenced Trigger step will always be called from another playbook. Therefore, it can get the environment using `input.params.<param_name>`.

Bear in mind that chaining multiple playbooks can overwrite the variables in your environment, such as the `request` object. Use the `Set Variable` step to give unique names to prevent this from happening. You can use the `Set Variable` step, to create an input parameter with a unique name that will be available in the parent

(calling) playbook. To add an input parameter, in the playbook designer, click the **Tools** menu and select **Parameters**.

Playbook Steps

At the core of Playbooks are Steps. Steps represent discrete elements of data processing during the course of the Playbook.

System modules such as `People`, `System Assigned Queues`, `Approval`, Content Hub, and Data Archival modules are not available for use as resources in playbook steps as they are reserved for administrative purposes.

Steps can be linked together in sequences to determine the flow of the Playbook, starting from the Trigger.

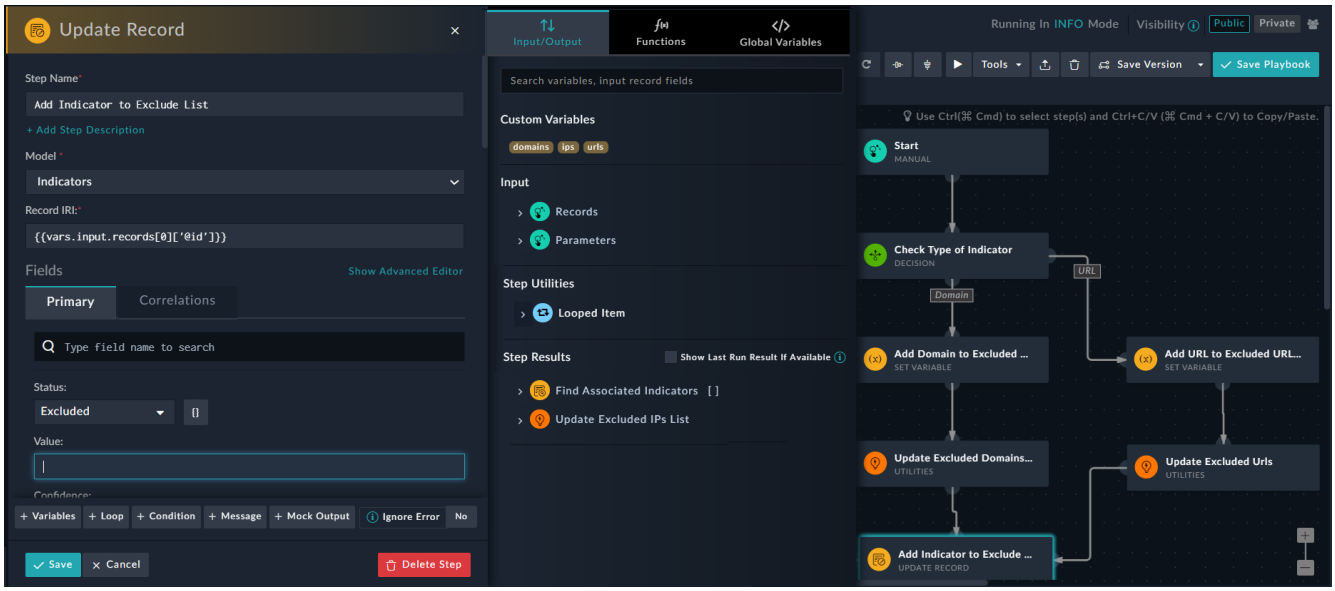
The Playbook Designer displays Playbook Steps only after you have added a Playbook Trigger.

Use dynamic values or variables in playbooks to access values of objects or perform lookups. Dynamic values can be passed to playbook steps as arguments directly, or they may be embedded in a larger string, where they will act more as global variables, getting replaced by a string representation of themselves. For more information, see the Dynamic Variables chapter. You can also use Dynamic Values to generate jinja templates, which can dynamically define various conditions within steps in a playbook. For more information, see the Dynamic Values chapter.

In case of any playbook step, if the input value for any field is in the JSON format, then you must enter the data in single quotes for example, `'{"company":"fortinet"}'`.

To update a picklist using a playbook, you can directly add the jinja for the picklist in the `{{"picklist name"|picklist("itemvalue of picklist")}}` format, for example, `{{"AlertStatus"|picklist("Open")}}`. The IRI Lookup option in the Dynamic Values dialog also allows you to select a picklist. For more information, see the Dynamic Values chapter.

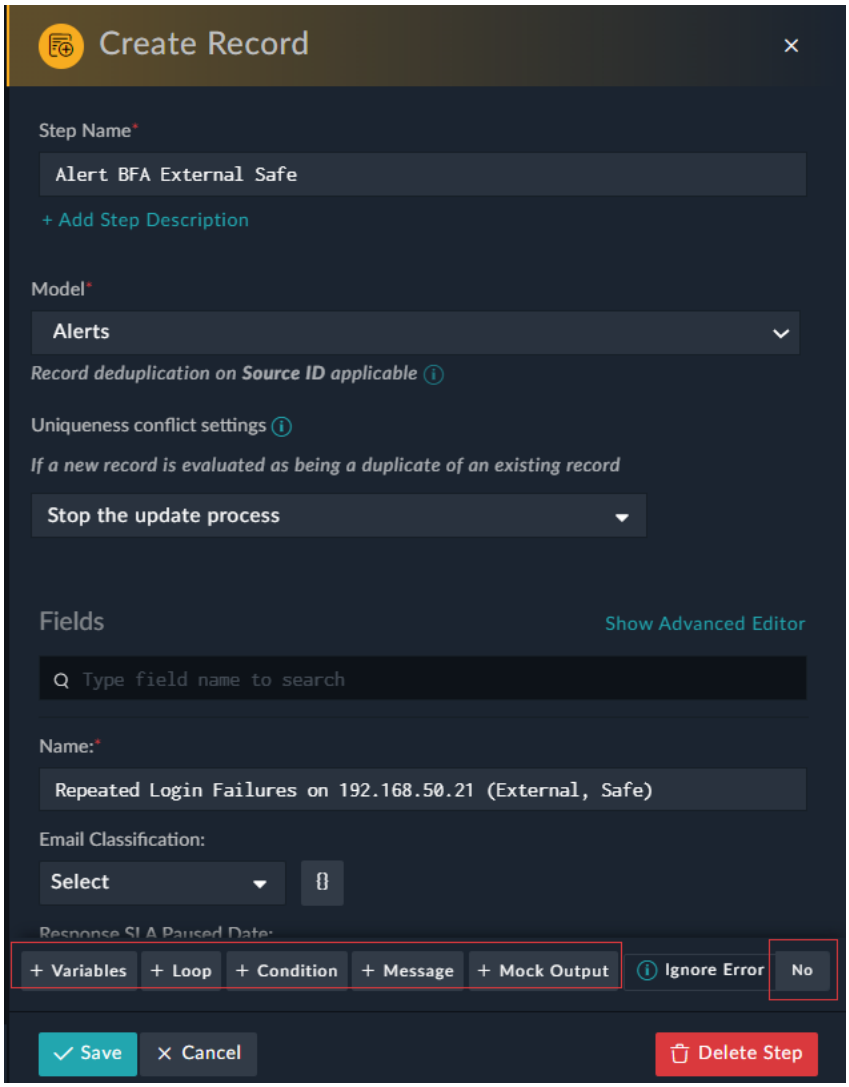
Click the **Dynamic Values** ( ) button to toggle fields such as, **Date/Time**, **Rich Text**, **File Selector**, **Picklist**, **Lookup**, and **Checkbox** fields and add custom (jinja) expressions to these fields. Ability to add jinja expressions to these fields enables you to write advanced playbooks. Once you click the **Dynamic Values** button, you can also use the **Dynamic Values** window to add expressions to these fields. For more information on the **Dynamic Values** dialog, see the Dynamic Values chapter.



Once you have saved the step, a graphic representing the step displays on the designed canvas in the upper left corner. You can create a link between the trigger and the step, for more information, see the Creating and Designing Playbooks chapter.

Double-clicking on the step reopens it and allows the user to edit the step or delete the step entirely by clicking **Delete Step**.

You can add variables, loops, conditions, and custom messages directly in the playbook step itself, and they get added in the `Step Utilities` section. You can also add a sample output (mock output) for cases where you do not want to execute a step but mock the output so that the playbook can move forward. You can also click the **Yes**/**No** button beside the **Ignore Error** checkbox to allow the playbook to continue executing even if the playbook step fails. These actions that you can use to extend a playbook step are present in the footer of the playbook step as shown in the following image:



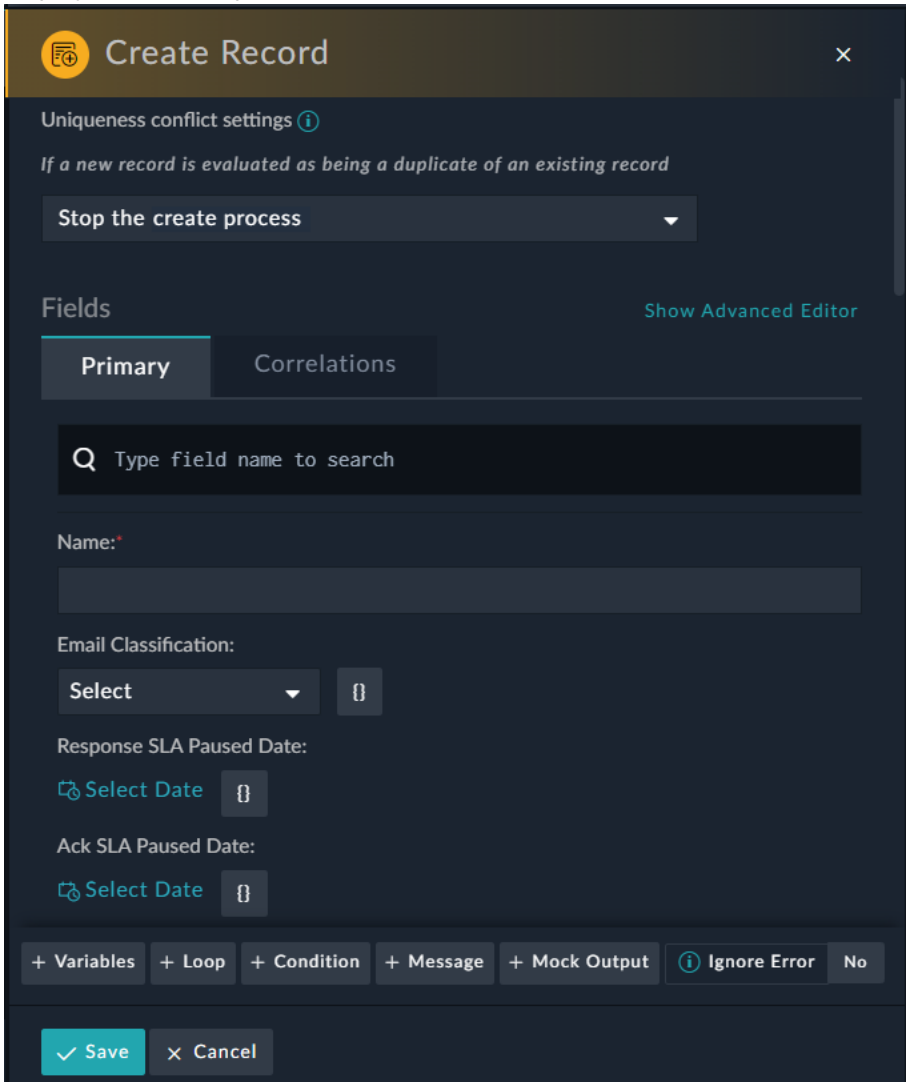
Core Steps

Core steps are used to find, create, update, or manage records within FortiSOAR, or to define variables for use throughout a playbook.

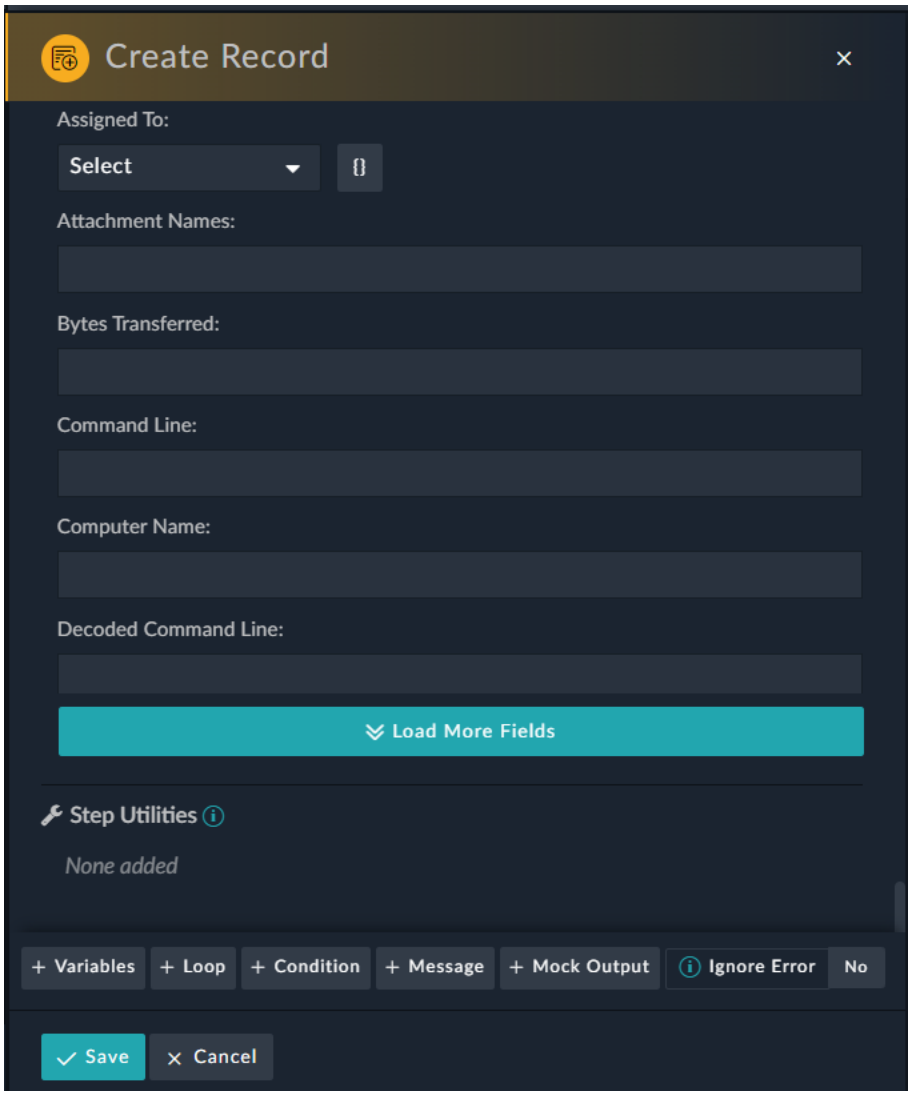
Create Record

Use the **Create Record** step to create almost any record type in the system. All required fields must be entered to match the model metadata for that specific record type. To create a record, select the module in which you want to create the record from the **Model** drop-down menu, which displays the `Create Record` form (**Form Editor**). Note that the fields displayed are specific to the entity type selected, and any conditional data requirements will be activated the same way as if the record was being added using the entity's model itself.

In the Create Record, Update Record and Ingest Bulk Feed steps, fields are divided into 2 tabs - The **Primary** tab, which lists all the primary fields and the **Correlations** tab, which lists all the correlated fields. Each tab has their own search box to search for fields. Search will be work on all the fields in the module and not on just fields displayed in the step.



To enhance the performance of these steps, only the first 30 fields are loaded in both the **Primary** and **Correlations** tabs, after which you will see a **Load More Fields** button. Clicking the **Load More Fields** button loads the next 30 fields, and this continues till all the primary fields in the modules are listed.



Note: If there are required fields in the module, then all the required fields are listed. If required fields are more than 30 then the initial field limit of 30 will be ignored, and all the required fields are listed; post-that if there are more fields left then the **Load More Fields** button is displayed. Also, once you reopen the Create Record, Update Record, or Ingest Bulk Feed steps, the fields that contain values after the required fields are displayed, followed by the **Load More Fields** button (if there are more than 30 fields).

If the data entity needs to reflect data specific to the entity that triggered the playbook, then use *dynamic values* in the fields.

To set the name of the incident name of the triggering entity, put the following in the Name field: `{{vars.input.records[0].name}}`.

Module editor supports the "JSON" field type. You can also convert data of a field of type `text` to JSON, using the `toJSON` Jinja filter. For example, `{{ vars.steps.<step_name>.data | toJSON }}`.

If the fields of the record being entered will always have the same data, enter the text in the corresponding fields and click **Save**.

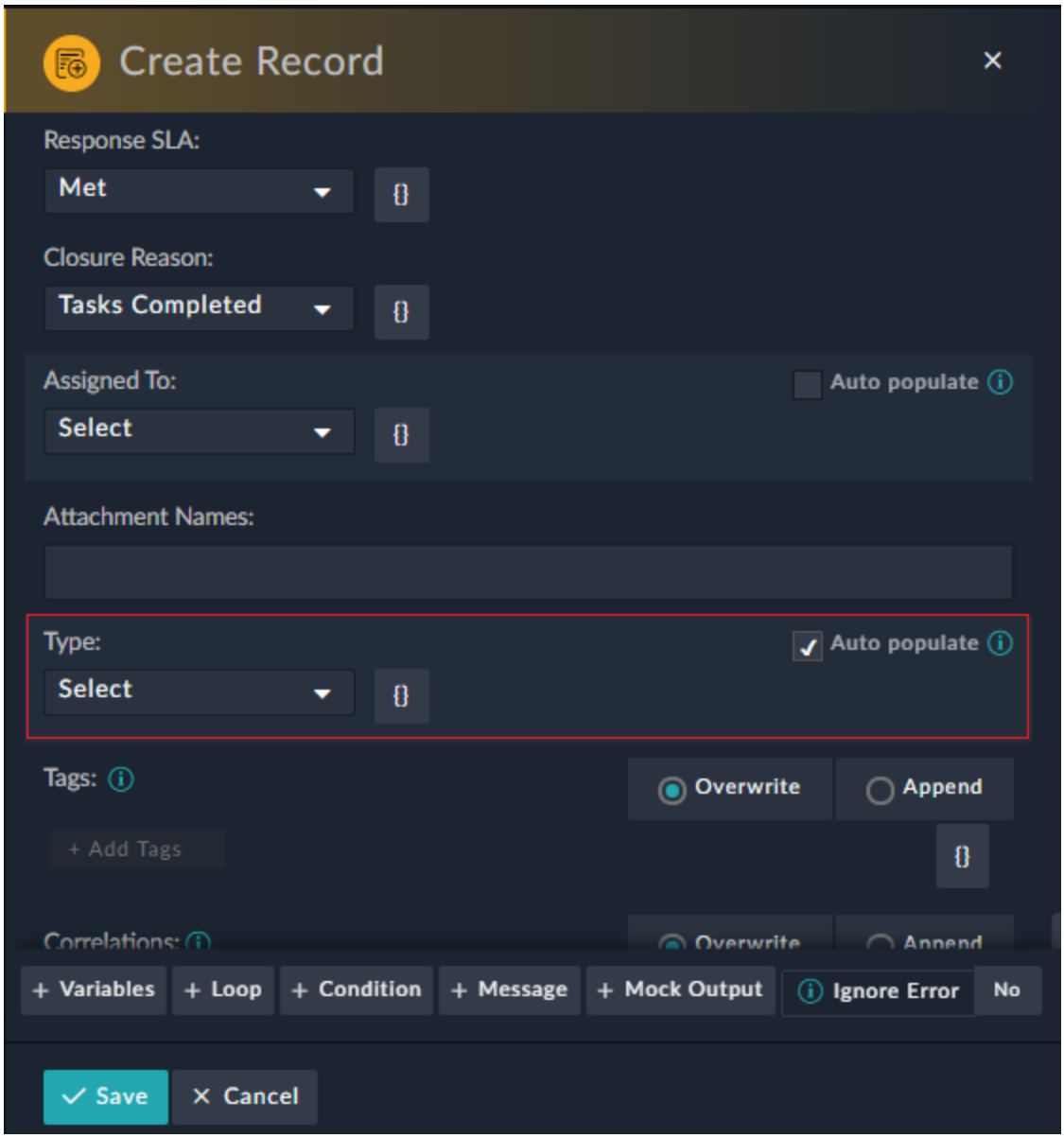
If in the Create Record step, you are specifying any `Date/Time` field in the jinja format, then that date/time field must be in the `epoch` format. To convert the input date/time field to the epoch time, you can either add the following Jinja value: `{{arrow.get(jinja variable).timestamp}}` or use the DateTime Expression library to enter the data directly in the JSON format by clicking the **Dynamic Values** ( ) button button. Clicking **Dynamic Values** ( ) button displays `Dynamic Values`, which displays the fields that you can directly edit either in the format of an attribute map (**Tree** view) or code (**Code** View).
Important: In version 7.0.0, FortiSOAR has updated the arrow library due to which the `timestamp` attribute has been changed into `int_timestamp` for *DateTime* jinja expressions. New playbooks must use the `int_timestamp` for any *DateTime* jinja expressions. For more information see the Dynamic Variables chapter.

You can also specify the date by clicking the **Select Date** link, which displays the Calendar from which you can choose the date/time.

From version 7.0.0 onwards, in case of the 'Create Record' and 'Update Record' steps, if your administrator has enabled any 'Lookup' or 'Picklist' type of field to accept the values generated from the recommendation engine, then you will see an **Auto populate** checkbox appearing beside this field.

To auto populate values in related (many-to-many) fields, in the 'Create Record' and 'Update Record' steps, ensure that the input provided in the playbook is the 'array' (list) format, and not 'comma-separated values'. For example, if you want the **Indicators** multi select picklist to be auto populated with common indicators in alerts records, then you must ensure that the indicators list that you provide in the playbook that contains the 'Create Record' or 'Update Record' step is in the 'array' (list) format.

A example of a field that has been enabled for recommendation is the 'Type' field in the following image:



If you select the **Auto populate** checkbox, and users have not specified any values for such fields, then the value of such fields get auto-populated with the values from the recommendation engine that is based on learning from past similar records.

It is possible to relate records with any valid relationships in the system. You can link the record that you are creating to a record in a related module. The **Create Record** step now displays a list of modules, in the `Correlations` field to which you can link the record that you are creating. For example, if you want to create an alert and therefore you have selected **Alerts** from the **Model** drop-down menu, the `Create Record` form will display related linking module fields, such as `Incidents`, `Indicators`, `Assets`, and `Attachments`. The Create Record step (for the upsert cases) and the Update Record step, the `Correlations` field, displays the records that are already linked to the created record. You can choose to overwrite the older relationships that are added to the created record, by clicking the **Overwrite** option in the `Correlations` field or append the new relationships to the relationships that are already added to the created record, by clicking the **Append** option in the `Corrleations` field.
Important: Avoid using the correlations 'Append' option in the 'Create Record' step of Data Ingestion Playbooks,

as this is not currently supported. Instead, if you have a special use case for linking relationships while creating records, utilize the 'Overwrite' option to add correlations in the 'Create Record' step.

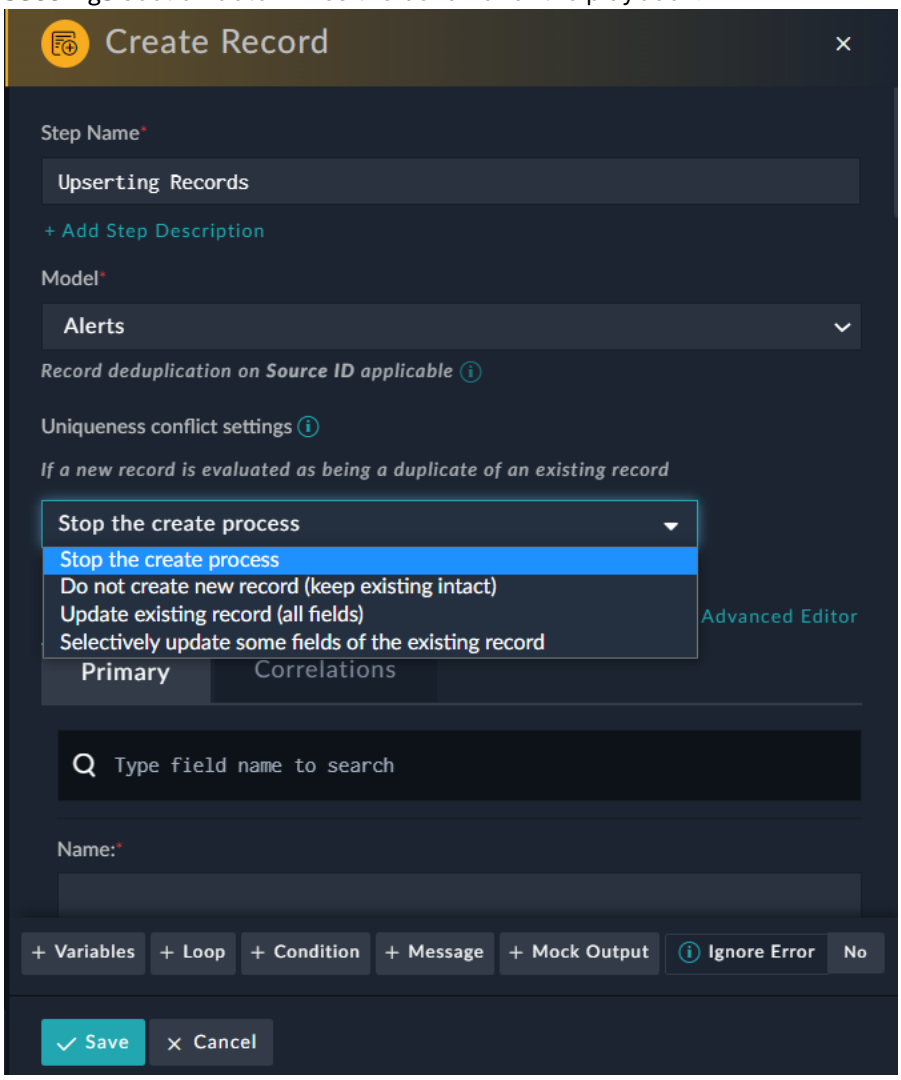
To link the newly created record, in the linking module field, add the IRI of the record to which you want to link the newly created record or add the respective jinja values. You can link multiple records using multiple comma-separated IRIs. For example, to link an alert that you are creating to an incident record, select **Alerts** from the **Model** drop-down menu and in the `Incidents` linking module field, add the IRI of the incident record, such as `/api/3/incidents/9a1142d2-adbf-4faf-a477-d8ff54419808` or add the jinja value or the incident record, such as: `{{vars.input.records[0]['@id']}}`. You can also use the array format to specify the IRI, `["/api/3/incidents/9a1142d2-adbf-4faf-a477-d8ff54419808"]`, or also add the jinja value in the array format, `["{{vars.input.data.records[0]['@id']}}"]`. To get the IRI for a record by navigating to the related module (`Incidents` in our example), for example, **Incident Response** > **Incidents** and select the record that you want to link. In the `address` bar, you will see the complete URL for that record. For example, https://{{*Your_FortiSOAR_IP*}}/modules/view-panel/incidents{{*UUID*}}
https://{{*Your_FortiSOAR_IP*}}/modules/view-panel/incidents/9a1142d2-adbf-4faf-a477-d8ff54419808.

It is recommended that you do not link more than 99 records in a single call. If you need to link more than 99 records, then run the update step in a loop with batches of 99 records.

The behavior of linking records relationships has changed in version 7.0.0 because if there is a record that is linked to thousands of other records, an update to such records causes constant high CPU usage. An example of such a record would be indicators like org name that get extracted as part of every alert and get linked to thousands of alerts. Therefore, it is recommended that you **link a maximum of 99 records in a single call**. This is because, if there are less than 99 records linked then the framework checks if the record being linked is already present in existing relations and if the same record is linked again and again, post-update triggers on relation "isChanged" is not triggered, also the linking is not audited again every time. However, from the 100th linked record, the framework only looks at the `__link`, `__unlink` keys, and hence, if the same record is linked again and again, post-update triggers on relation "isChanged" will get triggered, and also the linking gets audited again every time.

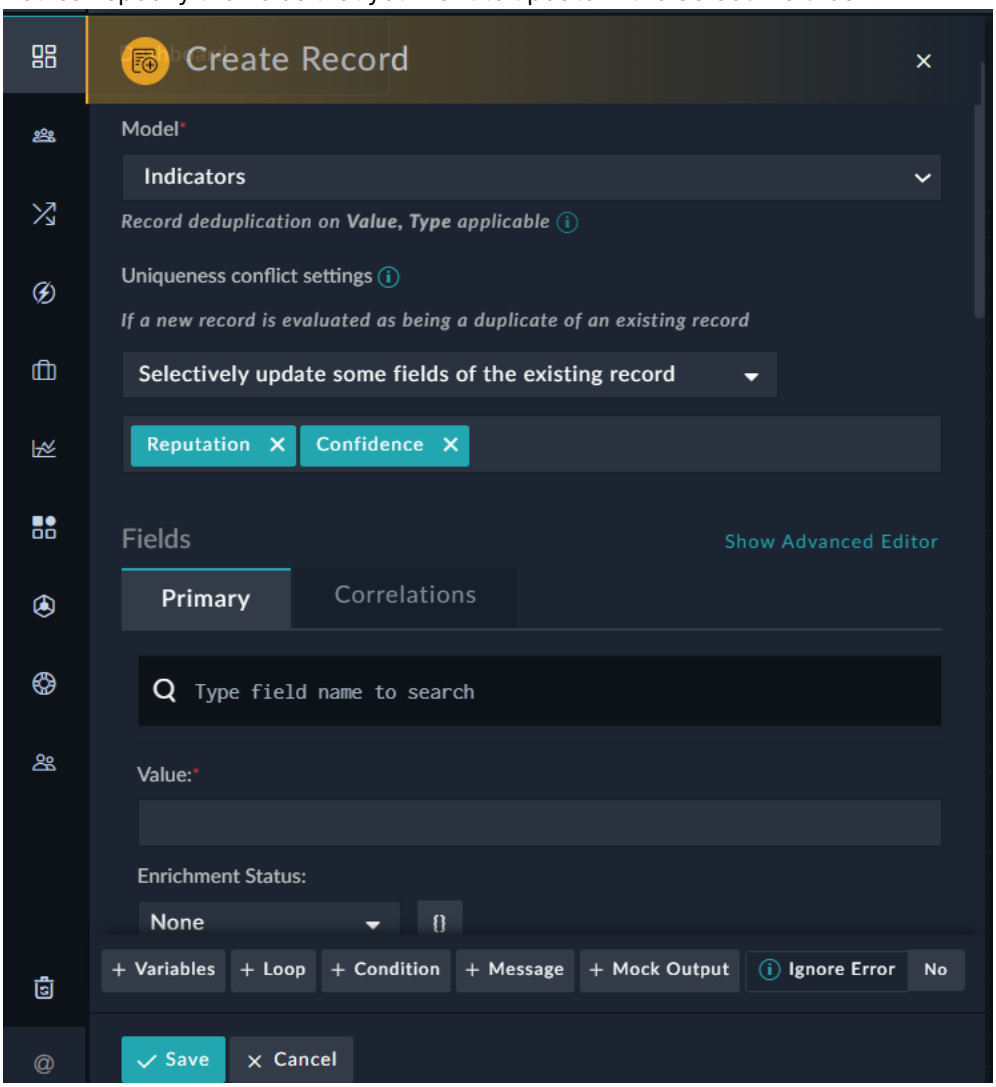
When you are creating a record using a playbook you can also enforce record uniqueness by defining unique constraints on the records of a module. For information on how to define record uniqueness using the Module Editor, see the Creating and Editing Modules topic in the "User Guide."

For modules that have unique constraints defined, the option that you choose in the `Unique conflict settings` section determines the behavior of the playbook:



- **Stop the create process**: This is the default behavior. The playbook fails if a duplicate record is found.
- **Do not create a new record (keep the existing intact)**: The playbook does not make any changes to the existing record and the existing record is returned *as is* as a result of execution of this step. The subsequent steps of the playbook work on the existing record if they refer to this step result.
- **Update existing record (all fields)**: The playbook updates the existing record with the new values that you have specified in this step.
- **Selectively update some fields of the existing record**: The playbook updates selective fields and/or correlations of the existing record with their updated values. Select this option if you don't want to replace all the fields of an existing record. For example, if an indicator does not exist, then you would want to create an indicator record with its reputation set to 'TBD'; however, if the indicator record exists, then you would want to only increase its sightings, i.e., update its reputation.

You can specify the fields that you want to update in the **Select Field** box:



For example, if *Source ID* is specified as a unique constraint on an "Alert" module, then you cannot create a record having the same source ID. However, if you have selected the **Update existing record (all fields)** or the **Selectively update some fields of the existing record** option, then either the complete existing record is replaced with the updated values or selective fields of the existing record are replaced with their updated values.

You can also update the correlations if you select the **Selectively update some fields of the existing record** option, For example, if you have created an alert and then extracted an indicator, for example, `gumblar.cn` with its status and reputation set as 'TBD'. Then enrichment playbooks are run which update the reputation to 'malicious', and investigation playbooks are run which update the status to 'blocked'. Now, another alert with the same indicator gumblar.cn get extracted with its status and reputation set as 'TBD'. Now, you have the option to update only selective fields in the correlation like reputation or last seen for the indicator.

Note: If you have imported playbooks into your FortiSOAR system or have upgraded your FortiSOAR system, and you have playbooks that contain the 'Create Record step with the Upsert' option, i.e., you have selected the **Update existing record (all fields)** option, then such playbooks will update only those fields that are selected by users for upgrade, the remaining fields are ignored.

Upsert behavior for uniqueness will not work for fields that are marked as encrypted.

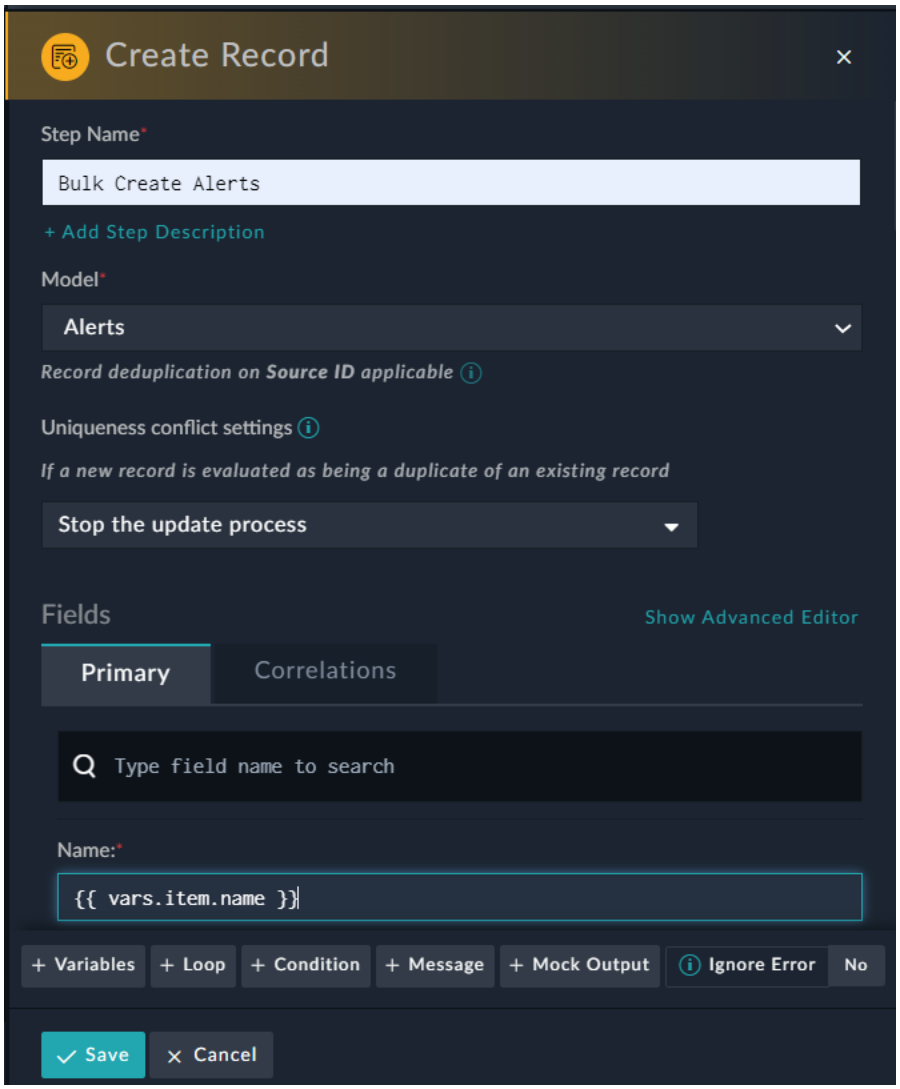
You can add tags in the **Create Record** and **Update Record** steps. You can add tags to the record that you are creating using the `Tags` field. Special characters and spaces are also supported in tags; however, the following special characters are not supported in tags: `'`, `,` , `"`, `#`, `?`, and `/`. Tags are useful in searching and filtering records. When you are updating a record, the `Tags` field, displays the tags that are already added to the created record. In the Create Record step (for the upsert cases) and the Update Record step, you can choose to overwrite the older tags that are added to the created record, by clicking the **Overwrite** option in the `Tags` field or append the new tags to the tags that are already added to the created record, by clicking the **Append** option in the `Tags` field.

Once you create the **Create Record** step, the playbook is automatically prompted to create a data record as specified in the step with either specific static text or record-relevant data using dynamic values.

When a record is created from a playbook, then that record's ownership includes the teams that are part of the "Playbook Appliance" including the admin team (SOC team). So, the record will be visible to all members of the teams that are part of the "Playbook Appliance", and their siblings and parents in the team hierarchy. If you want to change the ownership of the record, in the playbook, after the step to insert the record, add the immediate next step that will assign the desired team or user as the owner of the record.
Additionally, from release 7.5.0 onwards, picklist values and their attributes configured in the module when passed using an API are validated. For example, when creating an alert record and specifying its status, the value of the status passed must be one of the options from the 'AlertStatus' picklist. If a value from any other picklist is passed, the API call will fail. Releases prior to 7.5.0 did not have this validation, allowing API calls to pass. Therefore, from release 7.5.0 onwards, the playbook execution will fail for playbooks created with any attribute that is mapped to a picklist value not part of the configured picklist name. However, the picklist values and their attributes configured in the module are not validated for the 'bulk feed' step of records when passed using an API.

Create or Upsert Records in Bulk

You can also create or upsert records in bulk by using the **Bulk** option in the `for each loop` for "Create Records" and "Update Records." To create multiple records in a single request, for example, while ingesting from a data source, select the **Loop** option in the "Create Record" step. The Loop option has three modes: Bulk, Sequential, or Parallel. Provide the list of JSON inputs containing the sourcedata as the input to the loop and refer to each element as `{{ vars.item }}` in the step. For example, if you can provide the following JSON as input to the Loop option in the Create Record step to create alerts in FortiSOAR: `[{"name": "Name 1", "source": "FortiSIEM"}, {"name": "Name 2", "source": "FortiSIEM"}]`
You can ensure that the two alerts created in FortiSOAR have the corresponding names by using `{{ vars.item.name }}` against the `Name` field in the step.

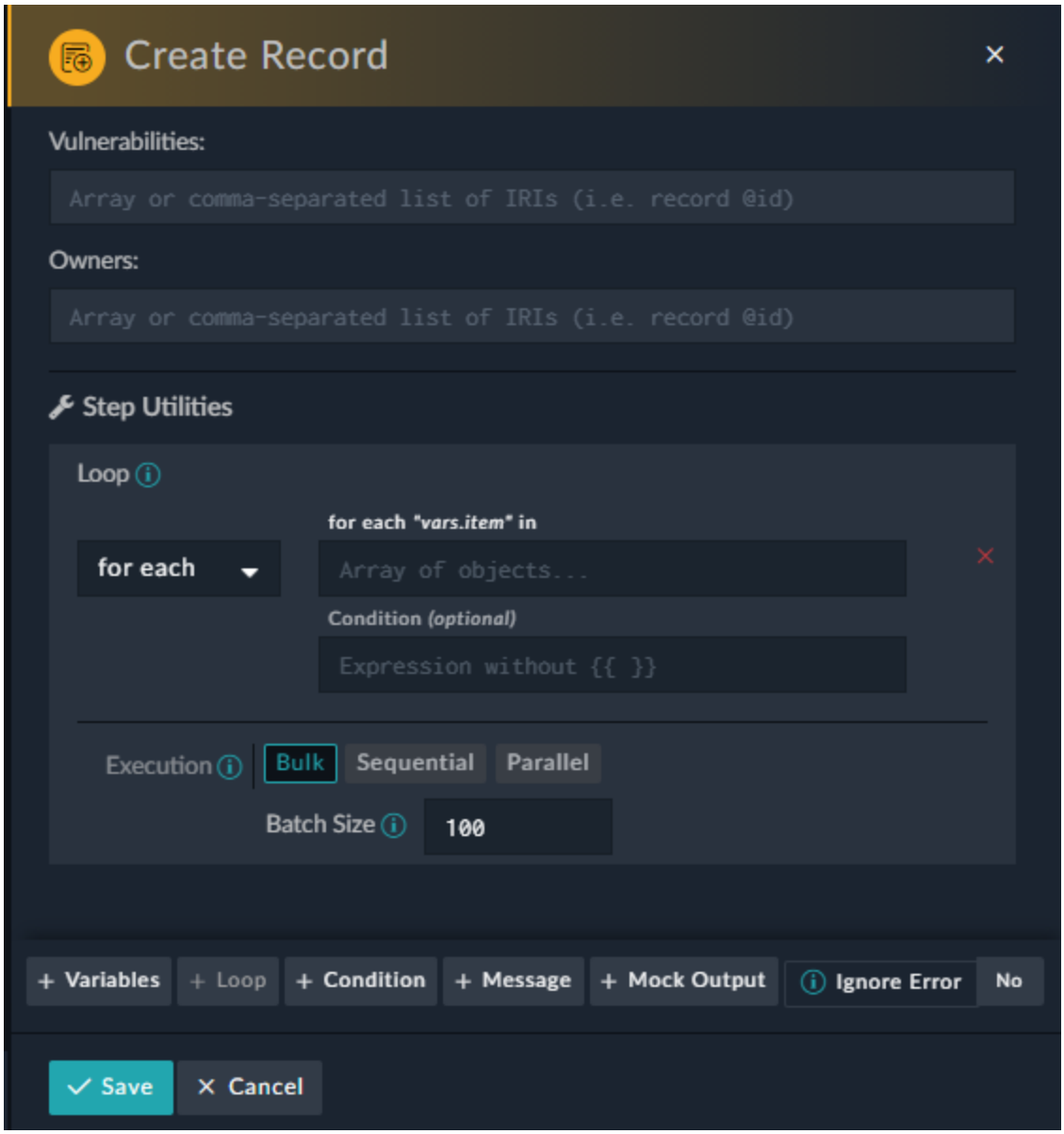


The **Bulk** mode creates all records in a single API request and is the most optimal and recommended method of creating or upserting records in bulk. This is also the default mode when you add a new Create Record step in a loop. The **Sequential** modes sends the API records separately for individual records, and one after another. So, the playbook step can abort at the first failure, without proceeding to create further records. The **Parallel** modes sends separate API requests for each record creation but using multiple threads to do so.

Batching large datasets when using the 'Bulk' option

A single batch can handle 100 to 200 records depending on the record size. If you are inserting larger number of records that causes the API call to time out, then you can insert records in batches.

From version 7.0.0 onwards, the 'Bulk' option has been enhanced to support batching of large number of records, by default, in the Create/Update record steps. To support this, the '**Batch Size**' option for the Bulk execution type has been added making it easy to bulk insert, upsert, or update large number of records. By default, the batch size is set to 100 records. You can increase or decrease this batch size depending on the record size. The following image shows a sample 'Create Record' step that is inserting a batch of 100 records:



Key Points:

- **Uniqueness Conflict /Upsert Behavior**: When a module has unique constraints (e.g., Indicator Source ID), configure how the playbook should handle conflicts caused by duplicate records. These settings determine the behavior when inserted data matches existing records with the same unique values. Options include:
  - **Stop the Create Process** (default): Use to ensure only unique records are created. Stops the process if a duplicate is detected.
  - **Do Not Create a New Record** (Keep the Existing Intact): Use to skip record creation when a duplicate exists, and continue working with the existing record.
  - **Update Existing Record** (All Fields): Use to replace the entire existing record with new data—suitable for overwriting outdated or incorrect information.
  - **Selectively Update Some Fields of the Existing Record**: Use to update only specific fields (e.g., reputation, status, last seen) while preserving all other data.

- **Layout and Field Handling**:
  - **Form has Two tabs**:
    Primary – displays the main fields required to create the record.
    Correlations – displays fields for linking correlated records (e.g., linking incidents to alerts).
    Each tab includes a search box for easier navigation. By default, the first 30 fields are loaded; click "Load More Fields" to load the next set of fields.
  - **Required Fields**: All required fields are always displayed—even beyond the 30-field limit.
  - **Dynamic Values**: Use to populate fields using data from previous steps.
    Example:
    To set the record name based on the triggering entity, use the following dynamic value (Jinja):
    `{{vars.input.records[0].name }}`
  - **Date/Time Fields**: Use epoch format for Date/Time Fields in Jinja. Convert input date/time using:
    `{{arrow.get(jinja_variable).timestamp }}`
  - **Handling Picklists**: Picklist values and their attributes are validated during record creation via the API. For example, when setting an Alert record's status, the value must be one of the predefined entries in the `AlertStatus` picklist. Providing a value outside this list will cause the API call to fail.
- **Record Linking** (Correlations): Link a new record to existing records in related modules (e.g., linking a new alert to an incident). Specify how to handle existing relationships:
  - **Overwrite**: Replace previous relationships.
  - **Append**: Add new relationships without removing existing ones.
    Note: Avoid using 'Append' in Data Ingestion Playbooks. as it may not be supported in some cases.
- **Ownership Assignment**: Records are automatically assigned to teams specified in the Playbook Appliance (e.g., SOC team). To assign a specific user or team, add a step immediately after the Create Record step.
- **Tags**:Tags can be added during record creation and used later for filtering and searching. Existing tags can be overwritten or new tags appended to a record's current tags.
  Note: Special characters such as `'`, `#`, `?`, and `/` are not supported in tags.
- **Bulk Record Creation**: Use a loop to create or update multiple records with one of three modes:
  - **Bulk Mode** (Recommended): Creates or updates all records in a single API request, providing the most efficient method for bulk record creation.
  - **Sequential Mode**: Processes each record individually with separate API requests, ideal for stopping the process upon the first failure.
  - **Parallel Mode**: Processes records in parallel using multiple threads, offering faster execution but higher resource consumption.
- **Batching Support**: For larger datasets, control the batch size—default is 100 records. Adjust this value as needed to prevent API request timeouts.
- **Data Integrity**: FortiSOAR enforces validation rules (e.g., picklist values, unique constraints) during the record creation process.

Update Record

Use the **Update Record** step to update a record in a module within FortiSOAR.

In the Playbook Designer, click the **Update Record** step and add the step name in the **Step Name** field, add the field to be updated in the `resource` field, add the module name and UUID of the record to be updated in the `collection` field (for example, you want to update the `Alerts` module, you will enter `api/3/alerts/{{uuid}}`), and then click **Save**.

The UI of the **Update Record** step displays an `Update Record` form that contains fields depending on the module you select in the **Model** drop-down menu, like the Create Record Step.

You must add the UUID or IRI of the Record you want to update in the **Record ID** field. In the **Record ID** field add either the IRI of the record that you want to update or add the jinja value of the record.

You can add details and field values to the "Update Record" step similar to the "Create Record" step.

If in the Update Record step, you are specifying any `Date/Time` field in the jinja format, then that date/time field must in the `epoch` format similar to the "Create Record" step. You can use the methods described in the "Create Record" step to convert the input date/time field to the epoch time. However, there is a difference between the "Create Record" step and the "Update Record" step, if you choose to enter the data directly in the JSON format by clicking the **Dynamic Values** button, which displays `Dynamic Values`. Dynamic Values appears empty in the case of Update Record (unlike the Create Record step, which displays fields according to the module you have selected) since you require to add only those fields in the JSON format that you want to update and do not require to see all the fields.

Once you add the record ID, you can update any of the fields of that record in the `Update Record` form directly and click **Save**. Once you click save, the data in the record that you specify by the record ID gets updated based on the changes you have made.

Find Records

Use the **Find Records** step to find a record in a module within FortiSOAR, using a query or search criteria.

In the Playbook Designer, click the **Find Records** step, and add the step name in the **Step Name** field, and then select the module in which you want to search for the record in the **Module** field.

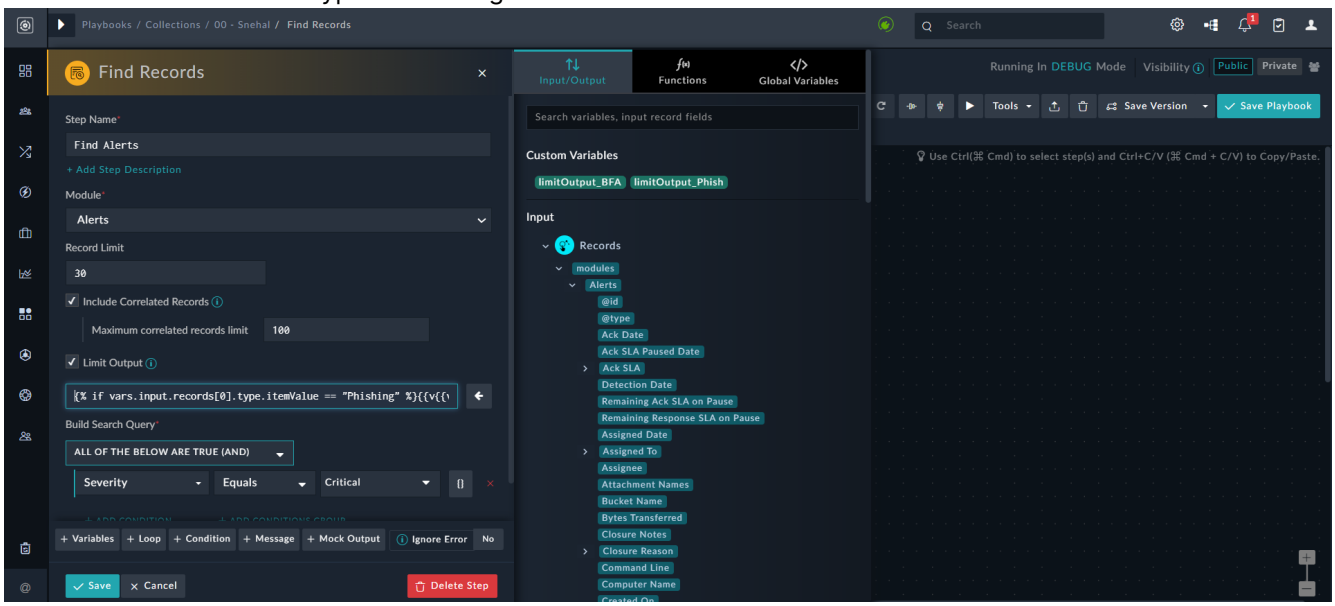
The Find Records step by default fetches only 30 records, if you want to change the number of records to be fetched, then enter the number of records to be fetched in the **Record Limit** field. For example, in the following image above we have entered `100` in the **Record Limit** field, which means that up to 100 records will be fetched.

The number of records that can be fetched using the 'Find Record' step has been limited to 5000. To override this default number, which is *Not Recommended*, you need to follow the steps mentioned in the Increasing the number of records that can be fetched using the 'Find Record' step topic.

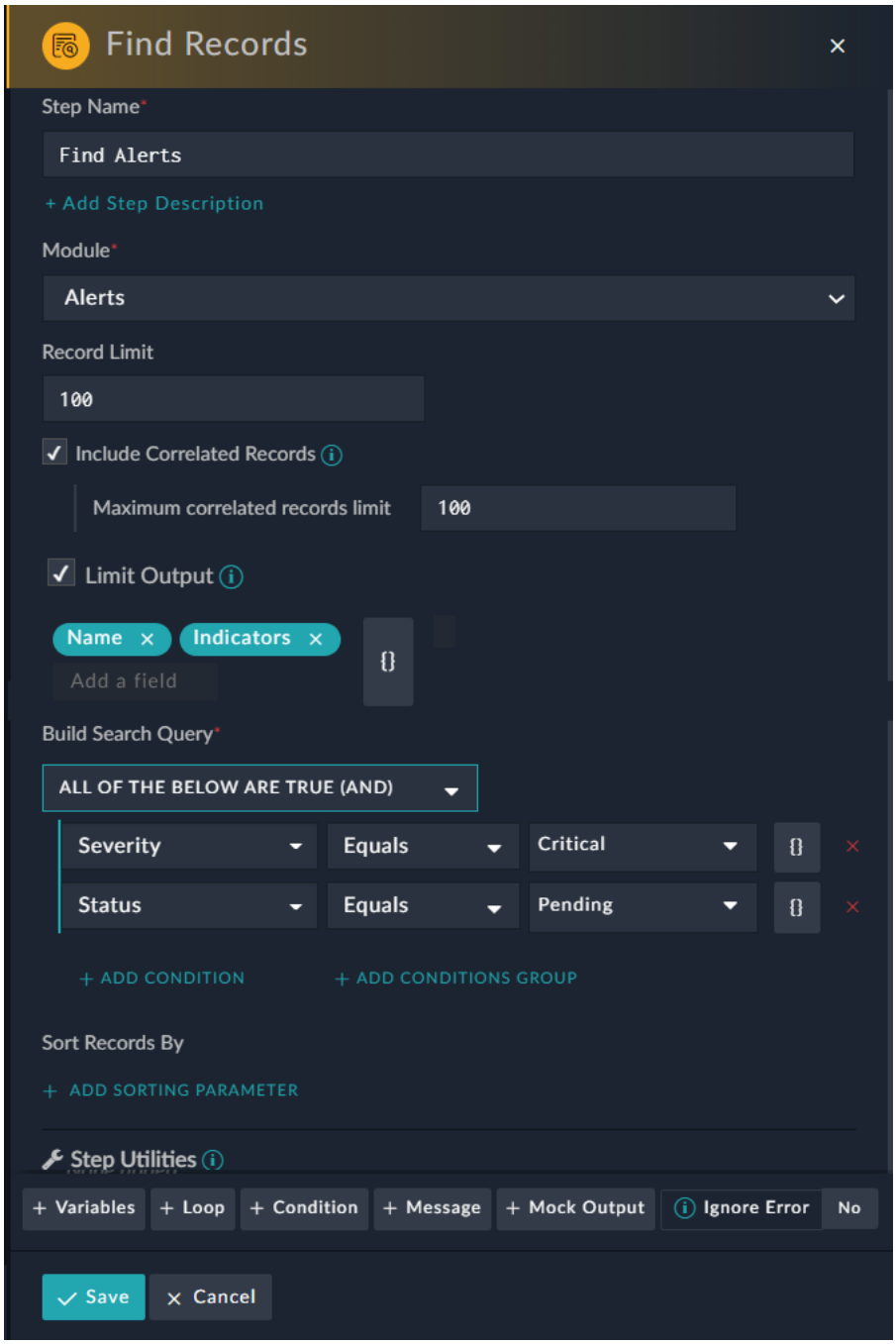
To include records that are correlated with the records that are being fetched using the 'Find Records' step. If you want to include correlated records, then select the **Include Correlated Records** checkbox. By default, the **Include Correlated Records** option is cleared for performance efficiencies. Once you select the **Include Correlated Records** checkbox, you can specify the maximum number of correlated records that you want to fetch in the **Maximum correlated records limit** field. Specifying the maximum number of correlated records to be fetched can help in avoiding the playbook timeout issue. The number that you can specify in the **Maximum correlated records limit** field must be a positive number and must be greater than '1'. By default, the number of correlated records to be fetched is limited to '100' correlated records.
Note: If **Include Correlated Records** option is cleared, then you can select only fields of the selected module. Therefore, to be able to choose related records such as indicators tasks, etc., you must ensure that you select the **Include Correlated Records** option.

You can select the **Limit Output** checkbox to restrict and refine your search results to only those fields that you require, improving usability and performance. For example, if you want to limit the output to display *only* the "name of the record" and "related indicators", then you should select the **Limit Output** checkbox, and in the **Add**

a field list that follows the **Limit Output** checkbox, select **name** and **indicators** fields. From release 7.4.2 onwards, you can use **Add Custom Expression** to define Jinja expressions for limiting the output of the 'Find Records' step. For example, if you want to limit the output of the 'Find Records' step to fetch certain fields based on the alert type. Prior to release 7.4.2., to achieve this, users would either have to add multiple Find Record steps for each alert type by adding a decision step, or if they wanted to use the output data, then 'Set Variable' steps would have to be added to remove the unnecessary keys from the JSON output. With this improvement, users can predefine the variables, `limitOutput_BFA`, and `limitOutput_Phish` in the following example image, and utilize these variable in the 'Limit Output' field to fetch the precise fields depending on the alert type, such as fetch "name", "emailBody", "emailHeaders" fields if the alert type is 'BFA', or "name", "sourceIP", "destinationIP" if the alert type is 'Phishing':



Based on the selected module, the **Nested filters** component gets populated in the `Build Search Query` section using which you can build the search query to find records and the click **Save**.



You can use **Nested filters** to filter records using a complex set of conditions. Nested filters group conditions at varying levels and use **AND** and **OR** logical operators so that you can filter down to the exact records you require.

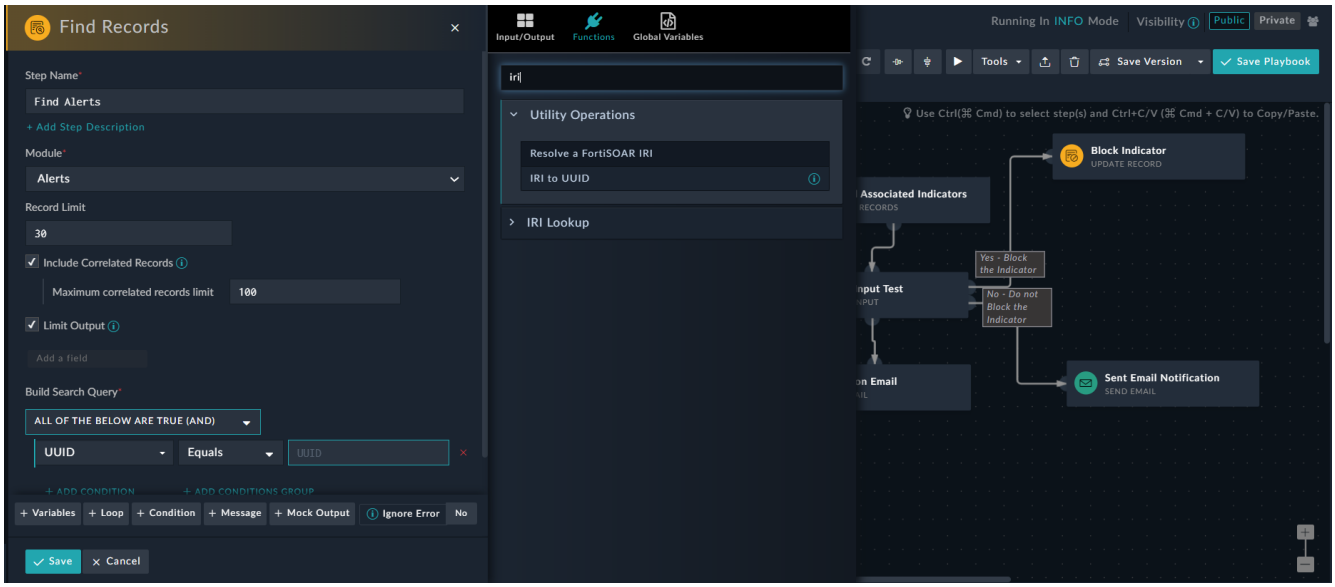
If you assign a "Custom" filter to a datetime field, such as Assigned Date, then the date considered will be in the "UTC" time and not your system time.

For more information on nested filters, see the Working with Template Widgets topic in the "User Guide."

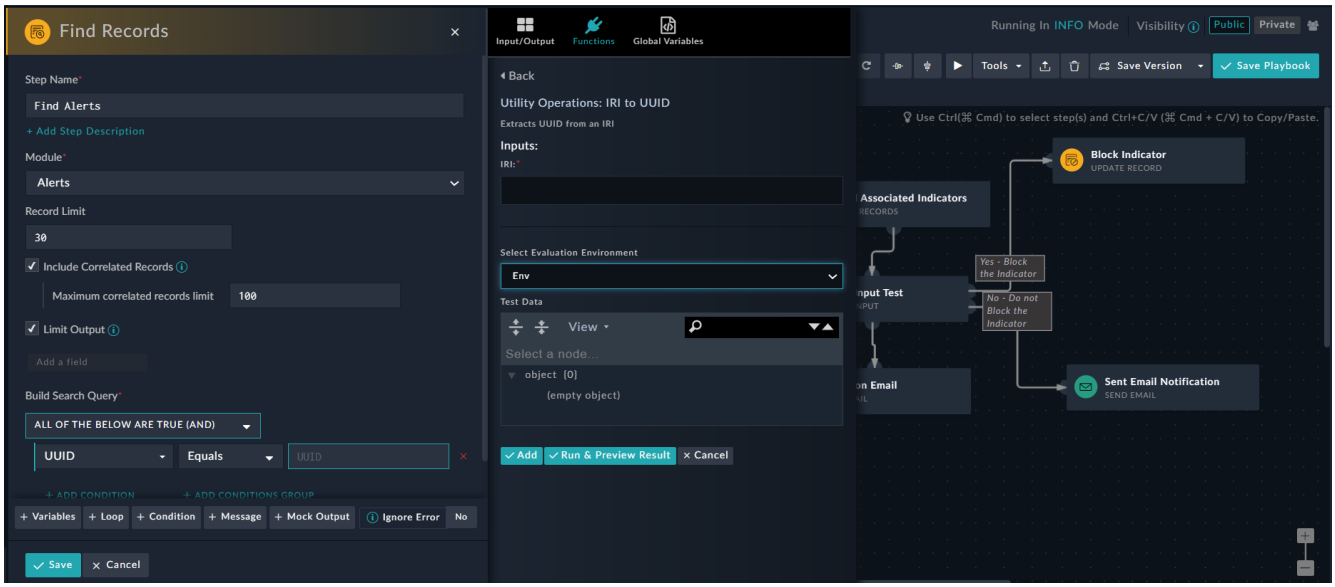
You cannot search or filter encrypted fields.

You can also write Jinja to build your search query in the **Nested filters** component in the `Build Search Query` section. You can either write you own Jinja or use the Dynamic Values dialog to add jinja to the field. See the Dynamic Values chapter for more information. You can also toggle between the Jinja and the original field type, for example in the image above; the **Severity** field displays the field as a drop-down list (which is the original field type). Click the {} icon to enter Jinja for this field. Similarly, the **Status** field displays the Jinja that has been entered in the field. Click the icon to toggle back to the original field type, which is a drop-down list.

You can also search records using a UUID. To search using UUID, in the **Nested filters** component in the `Build Search Query` section, select **UUID** from the Select a field drop-down list, select the operator such as **Equals** from the Select Operator drop-down list, then click the filter field to display the `Dynamic Values` window. Click the **Functions** tab and then search and click the **IRI to UUID** expression:



In the `Utility Operations: IRI to UUID` pop-up, enter a valid FortiSOAR IRI and click **Add**:



You can either add the IRI value directly or again use Dynamic Values to enter a jinja expression for the IRI. For more information, see the Dynamic Values chapter. The `Utility Operations: IRI to UUID` converts a valid IRI to a UUID using which you can search for records.

You can also sort the fetched records easily by clicking the **Add Sorting Parameter** link and choose the field based on which you want to sort the records in the `Sort Records by` section. You can also specify whether you want to sort the records in the **Ascending** or **Descending** order. For more information on sorting records, see the Working with Template Widgets topic in the "User Guide."

Increasing the number of records that can be fetched using the 'Find Record' step

From version 7.0.0 onward, the number of records that can be fetched using the 'Find Record' step has been limited to 5000. It is not recommended to change the value to a higher number and instead, you should use pagination by making an API call and navigate to the next page in a loop. However, if you yet want to override this default and increase the number of records to be fetched, do the following:

1. Edit the `/opt/cyops-api/config/parameters_prod.yaml` file to add the following code:

   ```
   api_platform:
       collection:
           pagination:
               maximum_items_per_page: 5000
   ```

   Change the value of the `maximum_items_per_page` parameter from '5000' to the desired number.
   Important: The above code must be added at the same level as '`parameters`' in the `parameters_prod.yaml` file.
2. Restart php-fpm and run cache clear:
   `# sudo systemctl restart php-fpm`
   `sudo -u nginx php /opt/cyops-api/bin/console cache:clear`

Key Points:

- **Uniqueness Conflict /Upsert Behavior**: When a module has unique constraints (e.g., Indicator Source ID), configure how the playbook should handle conflicts caused by duplicate records. These settings determine the behavior when inserted data matches existing records with the same unique values. Options include:

    - **Stop the Create Process** (default): Use to ensure only unique records are created. Stops the process if a duplicate is detected.
    - **Do Not Create a New Record** (Keep the Existing Intact): Use to skip record creation when a duplicate exists, and continue working with the existing record.
    - **Update Existing Record** (All Fields): Use to replace the entire existing record with new data—suitable for overwriting outdated or incorrect information.
    - **Selectively Update Some Fields of the Existing Record**: Use to update only specific fields (e.g., reputation, status, last seen) while preserving all other data.
- **Layout and Field Handling**:
    - **Form has Two tabs**:
      Primary – displays the main fields required to create the record.
      Correlations – displays fields for linking correlated records (e.g., linking incidents to alerts).
      Each tab includes a search box for easier navigation. By default, the first 30 fields are loaded; click "Load More Fields" to load the next set of fields.
    - **Required Fields**: All required fields are always displayed—even beyond the 30-field limit.
    - **Dynamic Values**: Use to populate fields using data from previous steps.
      Example:
      To set the record name based on the triggering entity, use the following dynamic value (Jinja):
      `{{vars.input.records[0].name }}`
    - **Date/Time Fields**: Use epoch format for Date/Time Fields in Jinja. Convert input date/time using:
      `{{arrow.get(jinja_variable).timestamp }}`
    - **Handling Picklists**: Picklist values and their attributes are validated during record creation via the API. For example, when setting an Alert record's status, the value must be one of the predefined entries in the `AlertStatus` picklist. Providing a value outside this list will cause the API call to fail.
- **Record Linking** (Correlations): Link a new record to existing records in related modules (e.g., linking a new alert to an incident). Specify how to handle existing relationships:
    - **Overwrite**: Replace previous relationships.
    - **Append**: Add new relationships without removing existing ones.
      Note: Avoid using 'Append' in Data Ingestion Playbooks. as it may not be supported in some cases.
- **Ownership Assignment**: Records are automatically assigned to teams specified in the Playbook Appliance (e.g., SOC team). To assign a specific user or team, add a step immediately after the Create Record step.
- **Tags**:Tags can be added during record creation and used later for filtering and searching. Existing tags can be overwritten or new tags appended to a record's current tags.
  Note: Special characters such as `'`, `#`, `?`, and `/` are not supported in tags.
- **Bulk Record Creation**: Use a loop to create or update multiple records with one of three modes:
    - **Bulk Mode** (Recommended): Creates or updates all records in a single API request, providing the most efficient method for bulk record creation.
    - **Sequential Mode**: Processes each record individually with separate API requests, ideal for stopping the process upon the first failure.
    - **Parallel Mode**: Processes records in parallel using multiple threads, offering faster execution but higher resource consumption.
- **Batching Support**: For larger datasets, control the batch size—default is 100 records. Adjust this value as needed to prevent API request timeouts.
- **Data Integrity**: FortiSOAR enforces validation rules (e.g., picklist values, unique constraints) during the record creation process.

Ingest Bulk Feed

Use the specialized Ingest Bulk Feed step to insert and update large volumes of records, primarily used while ingesting from Threat Intel Feeds, or others such as Vulnerabilities and Assets.

The step is significantly faster than the "Create Record" step. Currently, however, only primary fields, tags, lookups, and picklists are supported in the Ingest Bulk Feed step. You cannot add many-to-many relationships while adding records through this step.

Some specifics about this step:

- A single audit entry is created for a batch of records inserted and not a per record audit.
- Records created using this step are not peer replicable (master or tenant nodes in a multi-tenancy environment).

Playbooks with the 'On Create' or 'On Update' trigger will not work in the case records are ingested using the 'Bulk Feed' playbook step for those created or updated records. Additionally, the picklist values and their attributes configured in the module are not validated for the 'bulk feed' step of records when passed using an API.

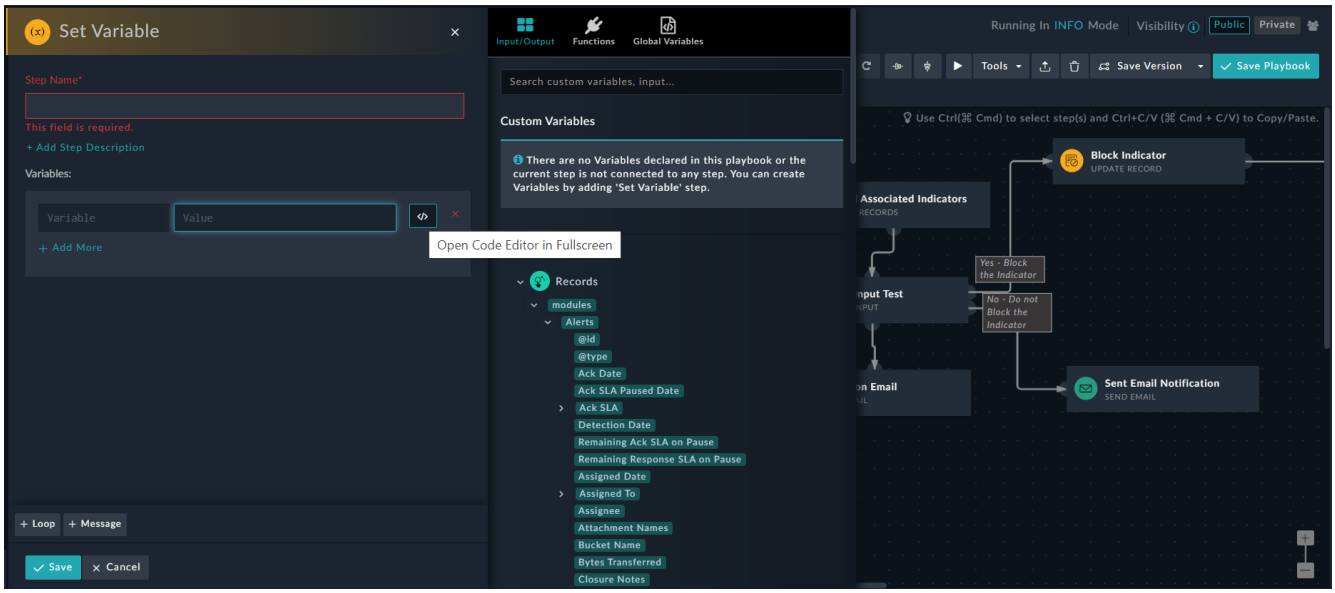
Considerations

- Prepare data with correct fields and formatting; avoid complex relationships (e.g., many-to-many).
- Monitor and process data after bulk ingestion.
- Records are not replicated across nodes in multi-tenant environments.
- A single audit entry is created for the entire batch, not per record.
- Playbooks with "On Create" or "On Update" triggers do not execute for bulk-fed records.
- Picklist values are not validated during bulk ingestion.

Set Variable

Use the **Set Variable** step to record a specific variable or variables for future use. Enter the variable name in alphanumeric characters and then define the value. The value may be a dynamic value itself. The scope of the variable created using `Set Variable` is *local*.

To create a variable, in the Playbook Designer, click the **Set Variable** step and add the `Name` and `Value` for the variable and then click **Save**. You can define multiple set variables in a playbook. To add dynamic values (Jinja) or variables, or access values of objects, or perform lookups, click the Dynamic Values button to display the 'Dynamic Values' window. For more information, see the Dynamic Values chapter.



You can also click the **Open Code Editor in Fullscreen** button to open a code text editor making the experience of adding and editing the code more user-friendly. Clicking the **Open Code Editor in Fullscreen** button opens the code editor in the full-screen mode. To exit the full screen, press ESC or click **Exit Fullscreen**.

Do not use reserved words, which are listed in the Reserved Keywords section as the variable name.

Once defined, the variable can be referenced in any remaining steps or in any child playbook, regardless of how many levels deep, the child playbooks are called.

The format for calling a variable is `{{vars.%name%}}`.

You can declare variables directly in the step, using the Variables option. See Variables for more information.

You can also add a Loop to iterate the Step Variable step or add a custom Message to the step. For example, adding a message such as "Computing xyz..." in cases where the set variable step is transforming some data.

Tips

- Set variables early (e.g., in a "Configuration" step) for better readability and reusability.
- Variables are local to the playbook but accessible in subsequent steps or child playbooks. Reference variables using `{{vars.%name%}}`

Evaluate Steps

Used to define logical flows within the playbook. These include decision points, user approvals, and manual inputs.

Decision

The **Decision** step serves as conditional validation within the playbook. You can specify "if this, then that" criteria that directs the playbook to execute specific steps based on the results of a specific condition. Many organizational processes differ depending on particular criteria, and to accomplish this; you can use the Decision step.

Use the Decision step to allow the playbook to specify, "If criteria = x, then do this next step." However, you can configure the Decision step with a variety of operators (equals, does not equal, <, >, etc.) and you can even chain logical conditions with AND/OR logic, allowing the organization's playbooks to define granular specifications for executing a specific sequence of steps.

To add a decision step to a playbook, click the **Decision** step. Initially, the Playbook Designer displays only the **Step Name** field, with no conditions. Type the step name and click **Save**. You can either create a Decision step with just the Step Name specified for now or create the possible conditions first then create the Decision step and then identify the condition once the Decision step and the potential outcome steps are connected. You can also define the entire step setting, or workflow, for the decision step, even if the connecting step is unavailable, allowing you to write the complete logic of the decision and then plug in the steps later.

The decision step functions in such a way that it evaluates multiple (alternative) conditions until any of them is fulfilled. This means that when the **Decision** step finds one condition that is fulfilled, then it skips the other conditions. If none of the conditions are fulfilled, then the default condition or route is defined.

You can define a default route that the playbook should take if none of the defined conditions are fulfilled. You can also add a description to describe the various routes that can be taken by the playbook.

You can also add a custom Message to the Decision step. For example, adding a message with the outcome of the decision step such as "`Blocked the IP <IP address> based on the reputation retrieved from FortiGuard`".

Example

A playbook execution route is based on the severity of the alert that gets created in the system at the hands of a third-party integration. If the alert that is created is not assigned any severity or a severity that does not match the severity that is defined in any of the conditions, i.e., Low or Minimal in our example, then it follows a default assignment, which is that the alert record assignment is updated to a Tier1 Analyst. This would be **Step A Alert Assigned to Tier1 Analyst**. If the alert is created with *Critical* severity, then that alert gets assigned to an Administrator (**CS Admin**). This would be **Step B Alert Assigned to CS Admin**. If the alert is created with *High* or *Medium* severity, then that alert gets assigned to a Tier2 Analyst. This would be **Step C Alert Assigned to Tier2 Analyst**.

The steps to create a playbook based on the above example are as follows:

1. Create a playbook, for example, `Alert Assignment` Playbook.
   FortiSOAR displays the Playbook Designer. The procedure for creating playbooks is mentioned in the *Playbooks Overview* section.
2. Add a On Create trigger, by clicking **On Create Trigger** in the `Playbook Designer`, type the name of the step in the **Step Name** field, for example, `Alert Creation` and then from the **Resource** drop-down list select the module on whose creation you want to trigger the playbook, for our example select `Alerts` and then click **Save**.
3. Drag-and-drop a connector point to connect to another playbook step. FortiSOAR adds a placeholder step on the playbook designer page and opens the `Steps` tab which displays all the available playbook steps. Click the **Decision** step and type the step name as `Assignment Based on Severity` and click **Save**.

4. Drag-and-drop connector points from the `Assignment Based on Severity` decision step and create the routes that the user can follow, i.e., create Step A, B, and C
5. Create Step A, where the alert is created with no severity, as follows:
   a. Click **Update Record**.
   b. Type the name of the step in the **Step Name** field, for example, `Step A Alert Assigned to Tier1 Analyst`.
   c. From the **Model** drop-down list, select **Alerts**.
   d. In the **Record IRI** field, use the Dynamic Values window and select the current record, `Input/Output >Input > Records > module > alerts > @id`.
   e. From the **Assigned To** drop-down list, select **Tier1 Analyst**, and click **Save**.
   Note: The DateTime field in a playbook step, for example, in a condition step, does not have the "Is Null" option in the Select Operator drop-down list.
6. Create Step B, where the alert is created with Severity Equal to Critical, as follows:
   a. Click **Update Record**.
   b. Type the name of the step in the **Step Name** field, for example, `Step B Alert Assigned to CS Admin`.
   c. From the **Model** drop-down list, select **Alerts**.
   d. In the **Record IRI** field, use the Dynamic Values window and select the current record, `Input/Output > Input > Records > module > alerts > @id`.
   e. From the **Assigned To** drop-down list, select **CS Admin**, and click **Save**.
7. Create Step C, where the alert is created with Severity Equal to High, as follows:
   a. Click **Update Record**.
   b. Type the name of the step in the **Step Name** field, for example, `Step C Alert Assigned to Tier2 Analyst`.
   c. From the **Model** drop-down list, select **Alerts**.
   d. In the **Record IRI** field, use the Dynamic Values dialog and select the current record, `Input > Records > module > alerts > @id`.
   e. From the **Assigned To** drop-down list, select **Tier2 Analyst**, and click **Save**.
8. Add conditions to the **Decision** step as follows:
   a. **For the Default Step**:
      i. Click **Add Default Condition**.
      If you want to use jinja to add advanced expressions and create complex conditions, you can click the **Show Advanced** link and add jinja in the condition text box. By switching to the 'Advanced' mode, you have complete flexibility to write Jinja-based conditionals, such as:
      - Checking if a key exists in the json:
        `"x" in vars.variable`
      - Comparisons:
        ```
        vars.variable == 5
        vars.variable >= 5
        vars.variable != 5
        vars.variable != []
        ```
      ii. From the **Select A Step to Execute**, select **Step A Alert Assigned to Tier1 Analyst**.
      You can optionally also add a tooltip, that describes the route this step or condition will take.
   b. **For the alternative steps**:
      i. Click **Add Condition**.
      ii. In the `Condition 2` section, use the *Condition Builder* to build the Severity Equals Critical condition as follows: From the **Select a field** drop-down list, select **Severity**, from the **Operator** drop-down

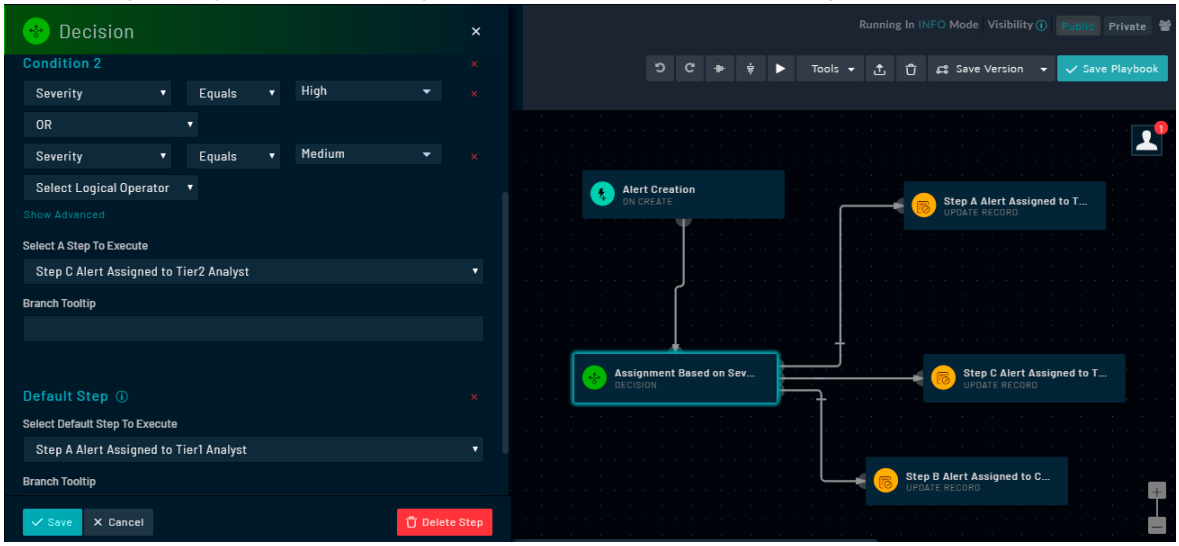
list, select **Equals**, and from the **Select** drop-down list, select **Critical**.
Click the **Show Advanced** link and add jinja in the condition text box to add jinja-based conditionals.

iii. From the **Select A Step to Execute**, select **Step B Alert Assigned to CS Admin**.
You can optionally also add a tooltip, that describes the route this step or condition will take.

iv. Click **Add Condition**.

v. In the `Condition 3` section, use the *Condition Builder* to build the Severity Equals High or Medium condition as follows: From the **Select a field** drop-down list, select **Severity**, from the **Operator** drop-down list, select **Equals**, and from the **Select** drop-down list, select **High**, and from the **Select Logical** drop-down list select **Or**, and then select **Severity**, from the **Operator** drop-down list, select **Equals**, and from the **Select** drop-down list, select **Medium**.

vi. From the **Select A Step to Execute**, select **Step C Alert Assigned to Tier2 Analyst**.
You can optionally also add a tooltip, that describes the route this step or condition will take.



In case of the No Trigger step for the **Condition Builder** you must add advanced jinja expressions in the **Condition** field. In the case of the *Manual Trigger*, if you have selected multiple modules, then for the **Condition Builder** you must add advanced jinja expressions in the **Condition** field.

Notes & Recommendations:

- **Define Clear and Precise Conditions**: Use specific, unambiguous criteria (e.g., `Severity = Critical, Status = Open`) to make decisions. Start with simple, high-priority conditions first, then fallback or less important ones.
- **Use Logical Operators Wisely**: Logical operators (AND/OR) are powerful tools to combine multiple conditions, but they should be used carefully.
  Use `AND` when multiple conditions must all be true (e.g., `Severity = High AND Assigned = NULL`).
  Use `OR` when only one condition needs to be true (e.g., `Severity = High OR Severity = Critical`).
  For complex scenarios, group related conditions together for better readability and logical flow.
- **Set a Default Route**: Always define a default path to ensure the playbook continues even if no conditions match.. This is useful for error handling and ensuring that the playbook doesn't fail or hang. The default route can also be used to trigger fallback actions, like notifying an administrator or assigning a low-priority analyst, if no specific condition is met.

- **Use Dynamic Values and Jinja Expressions**: Leverage dynamic inputs (such as **Input/Output > Input > Records > @id**) from previous steps to create real-time, flexible conditions.
  Use Jinja for advanced logic by writing expressions directly in the condition builder..
  Examples:
  To check if a key exists in a JSON response, use "key" in `vars.response`
  To compare variables, use `vars.severity == 'High'`
- **Modularize Conditions and Actions**: Break complex logic into smaller, manageable decision steps. Avoid one large step with many conditions—use multiple steps if needed for easier debugging and maintenance. Keep decision logic (conditions) separate from actions (e.g., notifications, record updates) to improve clarity and flexibility.
- **Document Conditions and Routes**: Add tooltips or descriptions for each condition or route to clarify its purpose.
  Example:
  Add a message in the route description to explain why certain conditions trigger specific actions, such as: "`Alert severity is Critical, so it is assigned to csadmin`."
- **Test Multiple Scenarios**: Validate all possible paths, including edge cases like missing or incorrect data that trigger the default route. Use varied inputs to ensure the playbook behaves as expected and make adjustments as needed.
- **Avoid Overloading with Conditions**: Don't overload a single decision step with too many or deeply nested conditions. If logic becomes complex, split it into multiple decision steps or move it to child playbooks. Complex logic can impact playbook performance—test and monitor as your playbooks scale.
- **Ensure Readability and Maintainability** :
  - Use clear and descriptive names for decision steps to improve readability..
    Example:
    Use "What is Alert Severity?" for a playbook that follows distinct routes based on alert severity—Critical, High, Medium, and Low.
  - Maintain a consistent structure for conditions, routes, and actions throughout the playbook to ensure clarity.
  - Modularize complex logicinto smaller, modular playbooks to improve readability and simplify maintenance.

Wait

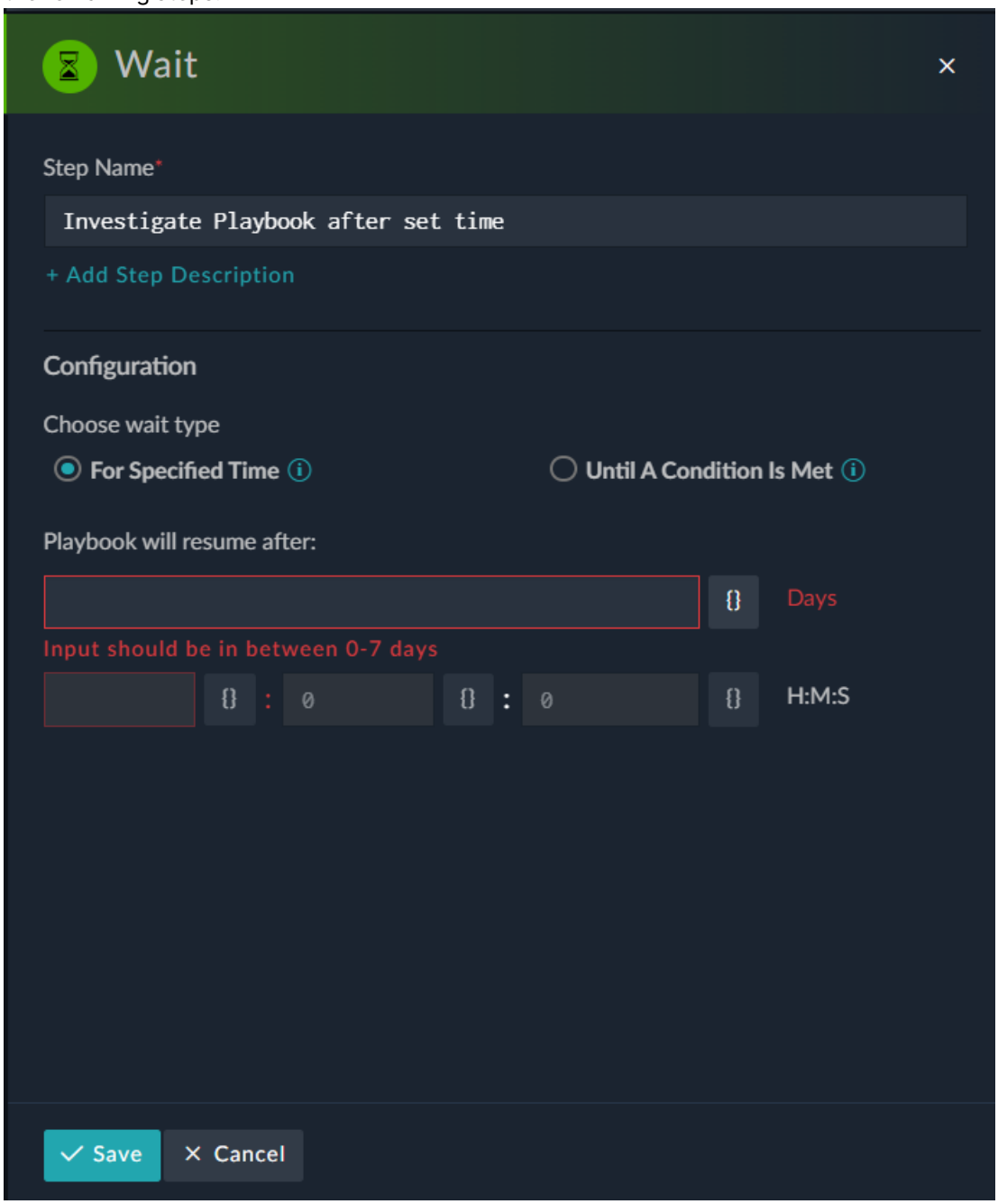
Use the **Wait** step to specify the time, or the condition to be met, before a playbook resumes executing its steps. It helps define the specific time that the playbook has to wait, or conditions to be met, for an action to occur in an external system or to allow for SLAs to elapse before continuing with the course of the playbook. For example, investigation playbooks have to wait for indicator enrichment to complete before beginning the subsequent investigation.

The playbook (workflow) is put back in the execution queue exactly after the specified wait time or once the wait condition is met. However, in the case of a workflow queue build-up, the resumed workflow has to wait for its turn to be executed, leading to a longer total resume time.

To configure the **Wait** step, click the **Wait** step. In the **Step Name** field, type the name of the step, and optionally add a description of the step in the **Description** field.

In the `Configuration` section, choose **For Specified Time** (Default) or **Until a Condition is Met** as the wait type.

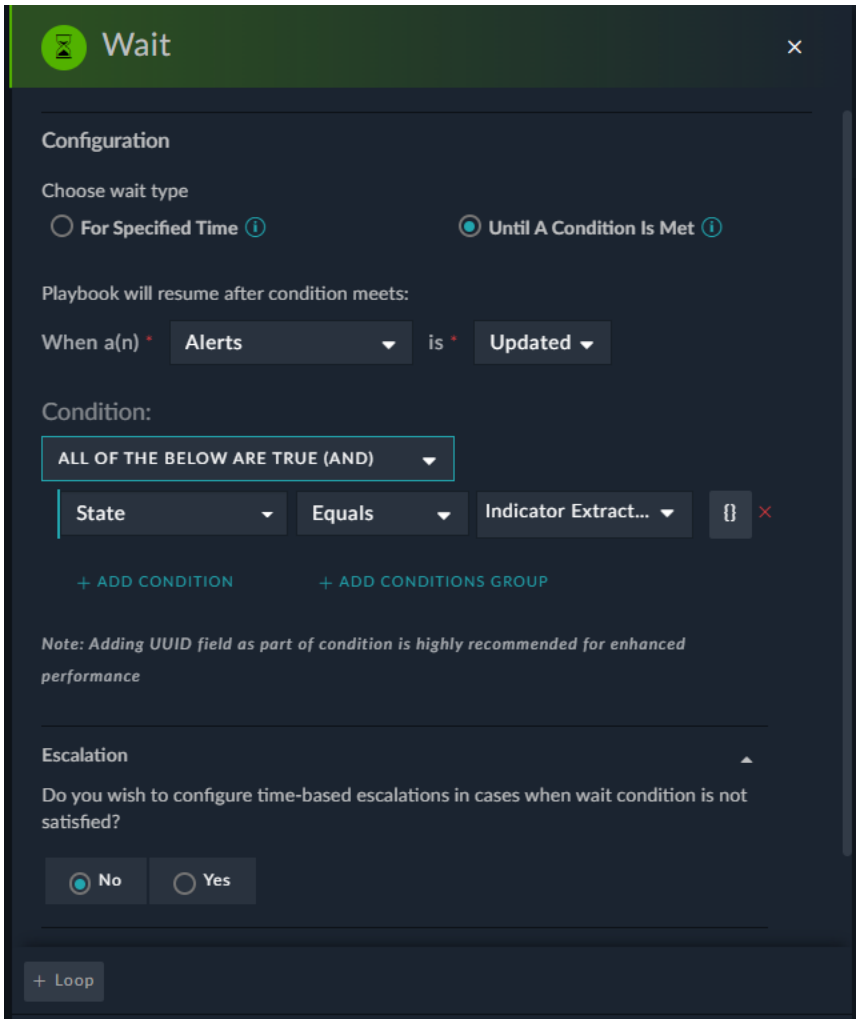
If you choose **For Specified Time**, enter the values in the **Days**, and **H:M:S** (Hours, Minutes, and Seconds) fields in the `Playbook will resume after` section. This option specifies the time the playbook waits before executing the remaining steps:



NOTE: For playbook optimization, playbooks that remain in the 'awaiting' state for more than 7 days are automatically terminated. As a result, the maximum allowable wait time that can be set for any playbook is 7 days.

You can also add custom Jinja expressions in the fields by clicking the Dynamic Values ({}) icon. Click **Save** to save the Wait step.

If you choose **Until a Condition is Met**, specify the condition that needs to be met before the playbook executes its remaining steps in the `Playbook will resume after condition meets` section. For example, in the case of a playbook that requires to wait till indicators associated with alerts are extracted, you can add the condition as follows:



Additionally, you can also configure a time-based escalation for cases where the wait condition is not satisfied. Click **Save** to save the Wait step.

Once you have saved the step and the **Wait** step appears on the Playbook Designer canvas, place the **Wait** step between steps that require to wait for a specific time or the fulfillment of a specific condition.

If a child playbook contains a "Wait" step, then it runs synchronously with the parent playbook, i.e., the parent playbook will wait for the child step to complete and only then resume its workflow. Earlier, if a child playbook contained a wait step, it would run asynchronously from the parent playbook, i.e., the parent playbook would continue its workflow independent of the child playbook and without waiting for the child playbook to complete its workflow.

Key Points

- **Optimize Wait Times**:
  - Minimize Wait Times: Avoid excessively long wait times (over several hours or days) that can cause the playbook to remain idle for extended periods. Set wait times to the minimum necessary for the specific task.
  - **Use Shorter Intervals**: Instead of setting a single long wait period, break it into shorter intervals. For example, check every 5 minutes for a condition. This helps improve the responsiveness of the playbook and minimizes unnecessary delays.
    Note: FortiSOARautomatically terminates playbooks that wait for more than 7 days, so ensure your wait times are within this limit to avoid unexpected terminations.
- **Condition-Based Waiting** (preferred): IInstead of relying on fixed time waits, use the "Until a Condition is Met" option. This ensures the playbook continues as soon as the required data or state change occurs, such as when an external system finishes processing or a specific field in a record is updated.
  Note: If your playbook depends on a system process (e.g., enrichment or an external API call), configure the wait to resume once that process is completed.
- **Escalate Time-Based Delays**: If a condition is not met within a reasonable time, configure an escalation (e.g., notifying an admin or triggering an alternate action) to handle delays effectively.
  Additionally, if waiting for a condition leads to extended delays, automatically trigger follow-up actions to prevent the playbook from stalling, such as re-checking the condition, initiating a manual review, or alerting a user.
- **Child Playbooks**: When using a Wait Step in child playbooks, ensure the child playbook runs synchronously with the parent. This ensures proper synchronization, preventing the parent playbook from proceeding before the child playbook completes.
- **Test and Monitor**: Regularly test your playbook with different wait conditions, both time-based waits and condition-based waits, to ensure it behaves as expected. Consider edge cases, such as system downtime or unexpected delays in external systems, and ensure your playbook can handle these situations (e.g., through error handling or alternate routes).

Approval

Use the **Approval** step to halt the execution of Playbook steps until the approval is received from the person or team that you have specified as an approver. Only once the approval is received will the Playbook move ahead with the workflow as per the specified sequence. Until the approval is not received, the **Execution Playbook Logs** will display the Playbook status as `Awaiting`. Once you complete adding an approval step, which includes adding the approver, the approver gets a notification for approval and the approver either accepts or rejects the approval request. Once an approval request is complete, the original playbook that contains the approval step resumes the execution of the remaining playbook steps. You can select only a single team or user as an approver.

In FortiSOAR release 7.4.1 the Approval step has been updated to honor RBAC, customizations, etc. Also, the response from the approval step now includes the approval status. However, from release 7.4.1 onwards, you can use only system notification as a mode of approval and cannot use **Email**.

In the case of systems upgraded to release 7.4.1, your existing Approvals steps will appear as were present in releases prior to 7.4.1 and if **Email** as a mode of approval (apart from the default, which is system notification) had been setup, then that would work. However, when you add new playbooks with the Approval step, those cannot have **Email** as a mode of approval.
Also, note that in the case of a distributed MSSP setup, pushing an approval playbook from a FortiSOAR (master) node that is on a higher version to a tenant node that is on a lower version is not supported.

Permissions Required

- To view and interact with approval notifications, you must be assigned a role that has a minimum of `Create`, `Read`, and `Execute` permissions to the `Playbooks` module `Read` and `Update` permissions on the `Approvals` module.
- To create a playbook and add an approval step or any other step, you must be assigned a role that has a minimum of `Create`, `Read` and `Update` permissions on the `Playbooks` module, and a minimum of `Read` permissions on the `People` and `Security` modules.
- To add an approval step or any other step to an already existing playbook, you must be assigned a role that has a minimum of `Read` and `Update` permissions on the `Playbooks` module, and a minimum of `Read` permissions on the `People` and `Security` modules.

Examples of usage of an approval step:

- A case for when you can use an approval step could be when you want to get an approval from the SOC team before blocking an indicator.
- Another case for when you can use an approval step could be when you have sent a URL to a third-party URL authenticator to identify whether the URL is malicious or not. If you get a report from the third-party URL authenticator that the URL is malicious, then you want to block that URL. However, before you block that URL, you require approval from the manager of your SOC, and therefore here you would use an approval step.
- Another example would be when you want an Incident to be deleted from the system. However, before the deletion, you require approval from an Incident Lead, and therefore here you would use an approval step.

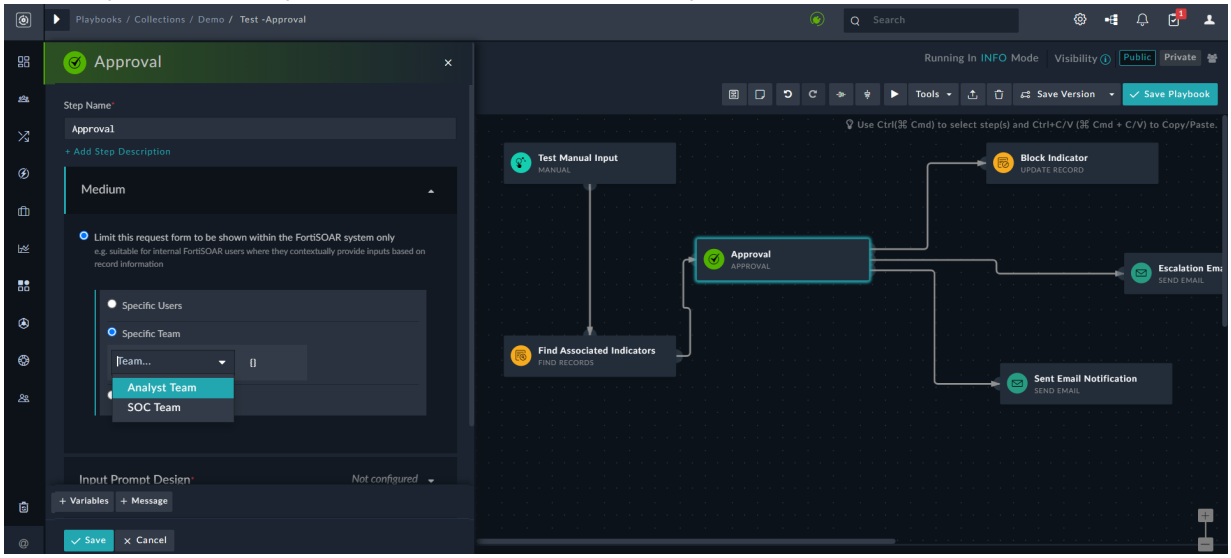
Adding an Approval Step

This topic describes how to add an Approval step for release 7.4.1 and later and uses the first example, where we want to get approval before blocking an indicator:

1. Open FortiSOAR and click **Automation** > **Playbooks** in the left navigation bar.
2. On the `Playbook Collections` page, click on an existing playbook collection.
   This opens the `Playbook` page, click on the playbook in which you want to add an approval step.
   This opens the playbook in the `Playbook Designer`.
3. Click the **Approval** step in the `Evaluate` section.
4. For the Approval step, in the **Step Name** field, add the name of the step.
5. Click the **Medium** menu to choose the medium of delivery for the approval.
   Important: Approval is accessible only to users within FortiSOAR. If you want approval to be accessible to non-FortiSOAR users to provide inputs, use the Manual Input step. Select one of the following options to determine who is responsible for responding to the approval input prompt:

a. **Specific Users**: The approval is visible and actionable by users, other than the user who is assigned to the record, who need to provide the input. When you select this option, then the People multi-select list appears from which you can select users who require to take the decision. You can also add a custom expression in this field.

b. **Specific Team**: The approval is visible and actionable by team(s) who requires to provide the input. This means that any user who is part of the selected team(s) will be able to provide the input. When you select this option, then the Team multi-select list appears from which you can select specific teams(s) that can provide their input. You can also add a custom expression in this field.



Note: The teams or users who are specified as owners, i.e., to whom this approval is assigned, must have access to the record and appropriate permissions to perform the steps required to complete the approval.

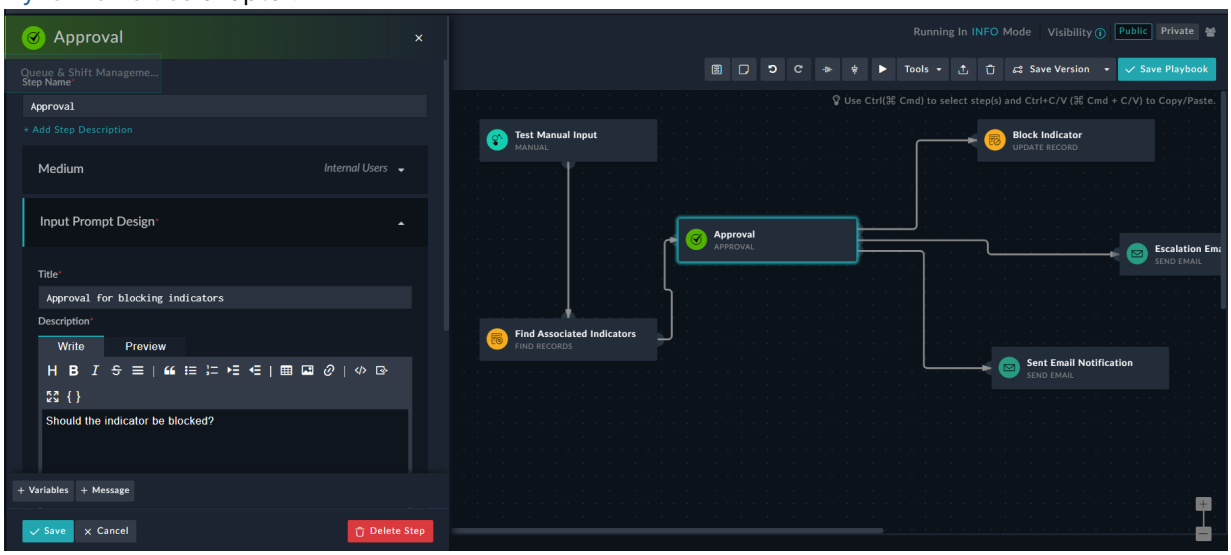
c. **No specific assignee**: The approval is visible and actionable to everyone in the FortiSOAR instance.

6. Click the **Input Prompt Design** menu, and configure the design for your approval prompt:

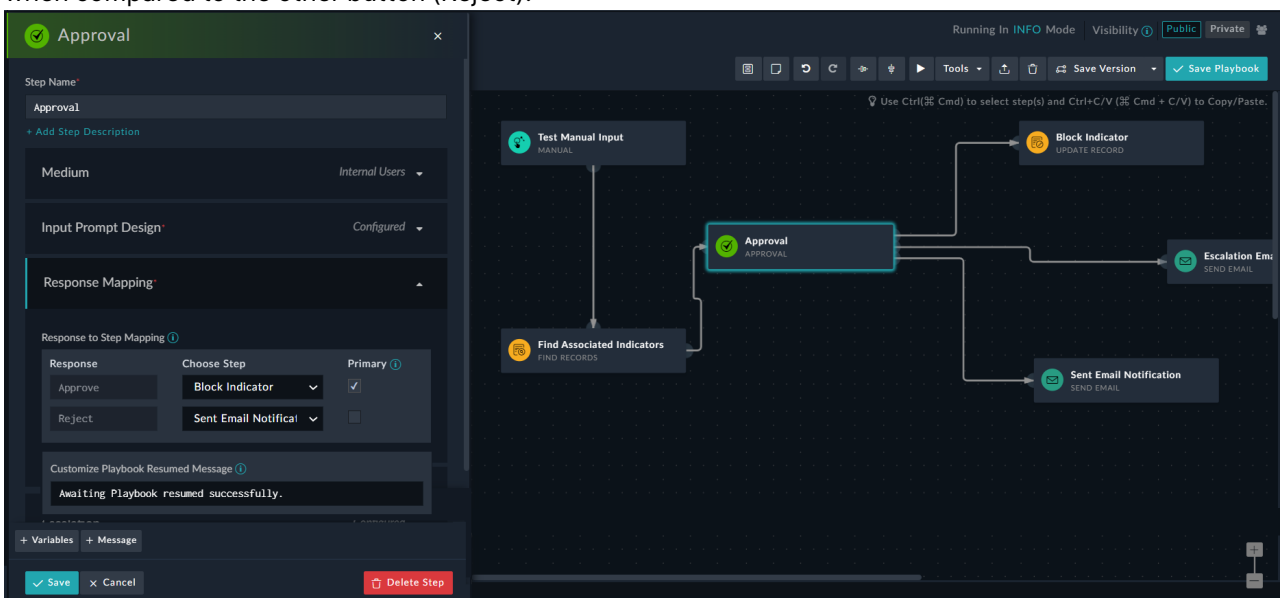
a. In the **Title** field, enter the title for the prompt. For example, `Approval for blocking indicators`

b. In the Description field, enter the description of the approval request, which can include the reason for the approval request. For example, `Should the indicator be blocked?`
Use Dynamic Values if you want to add Jinja to the description field . For more information, see the

Dynamic Values chapter.



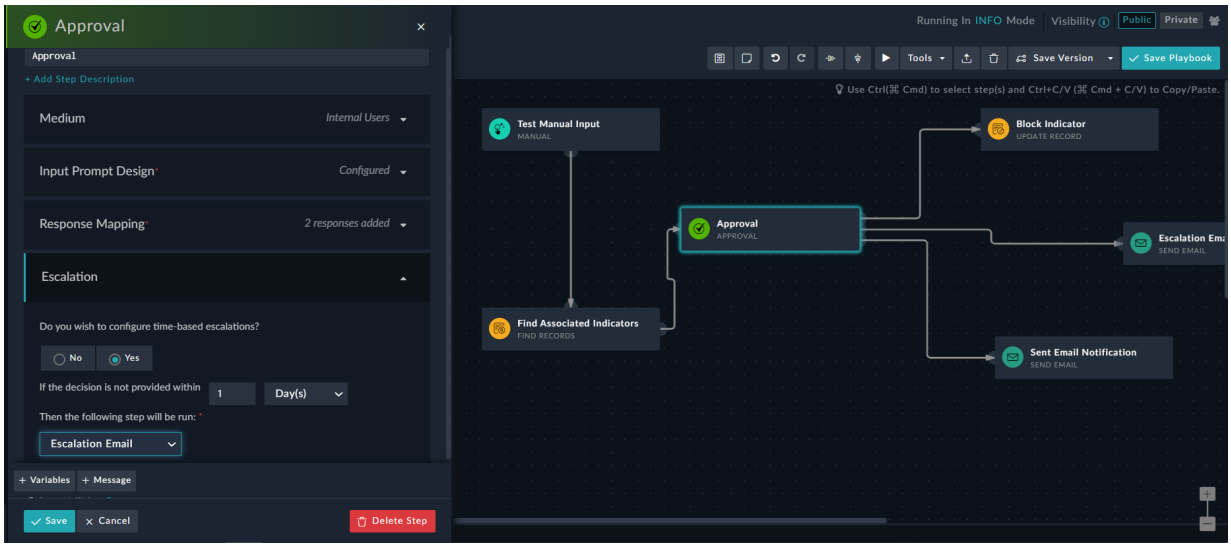
7. Click the **Response Mapping** menu to add the custom response options that the user can choose from when presented with the decision.
   You can add the custom response for the decision first so that you can define the complete workflow and then create the corresponding playbook steps.
   For our example, in the `Response to Step Mapping` section, choose the corresponding steps for the Approve and Reject options. For the **Approve** option, in the **Choose Step** field, select the **Block Indicator** step to block the indicator. For the **Reject** option, in the **Choose Step** field, select the Send Email Notification step to send a notification to the SOC team (admin team) so that they can be informed that this indicator is not blocked and they can take further steps if required.
   The "Approve" option is always set as the primary option, i.e., the **Primary** checkbox is always selected. A primary response (Approve) adds a distinct visual style to that option button, making it more prominent when compared to the other button (Reject):



   Additionally, you can customize the message that is displayed when a approval playbook is resumed by typing a custom message in the **Customize Playbook Resumed Message** field. The default system message "`Awaiting Playbook resumed successfully`" is displayed when the approval playbook is

resumed if you do not set any custom message. The custom message can have a maximum of 255 characters.

8. (Optional) Click the **Escalation** menu to define actions that should be taken in case the approval is not given within the specified time frame. From the `Do you wish to configure e-based escalation?` section, choose **No** or **Yes**.
   If you choose **No**, then there is no time-based escalation.
   If you choose **Yes**, then you must specify the following:
   a. The time within which the approval must be given. In the **If the decision is not provided within** field. You can specify the time in Days, Hours, or Minutes within which the approval must be given. The minutes option can be used where approval is quickly required, such as 15-20 minutes for time-sensitive operations.
      NOTE: The maximum duration for an escalation that you can set is 7 days. If no approval or input is provided within this time, the playbook will be terminated, ensuring efficient workflow management.
   b. The Escalation step must be selected from the **Then the following step will be run** field. For example, if you want to send an email notification to the managers, then you can define that step as **Escalation Email** and connect it to the Approval step and choose that option from the **Then the following step will be run** field:

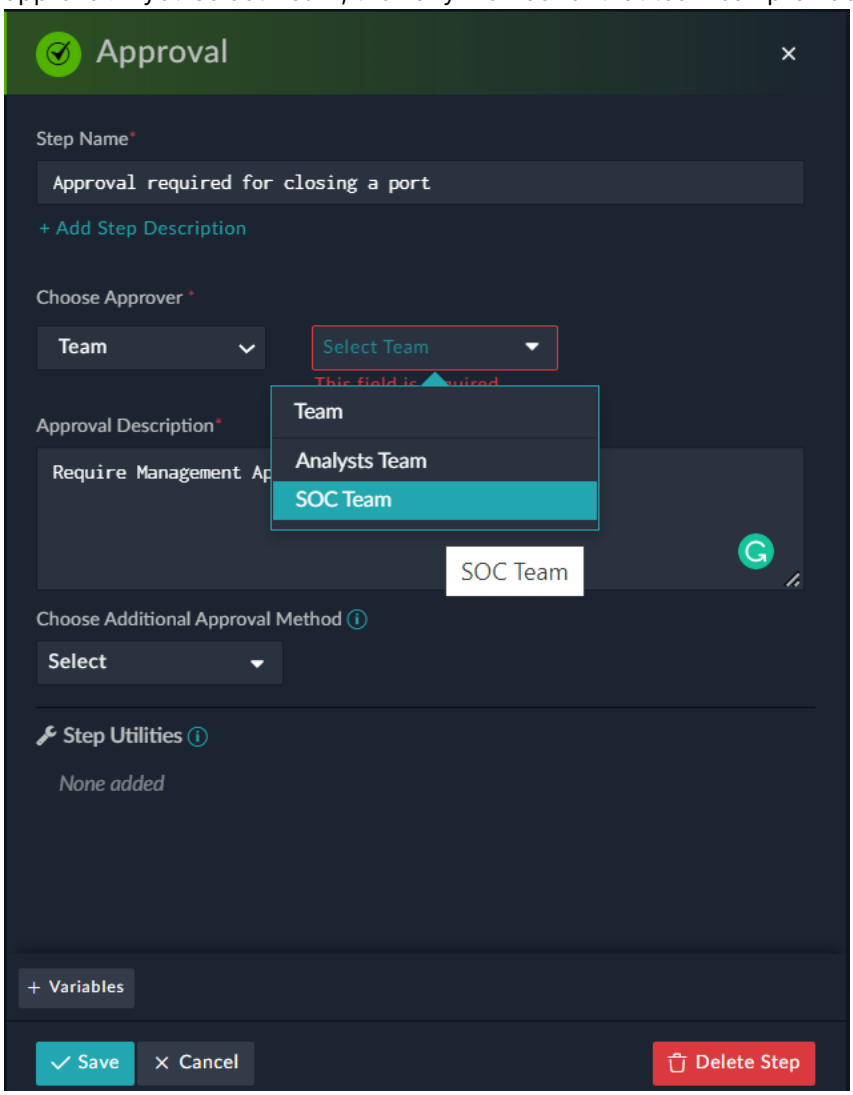


9. Click **Save** to save the approval step.

Adding an Approval step prior to release 7.4.1

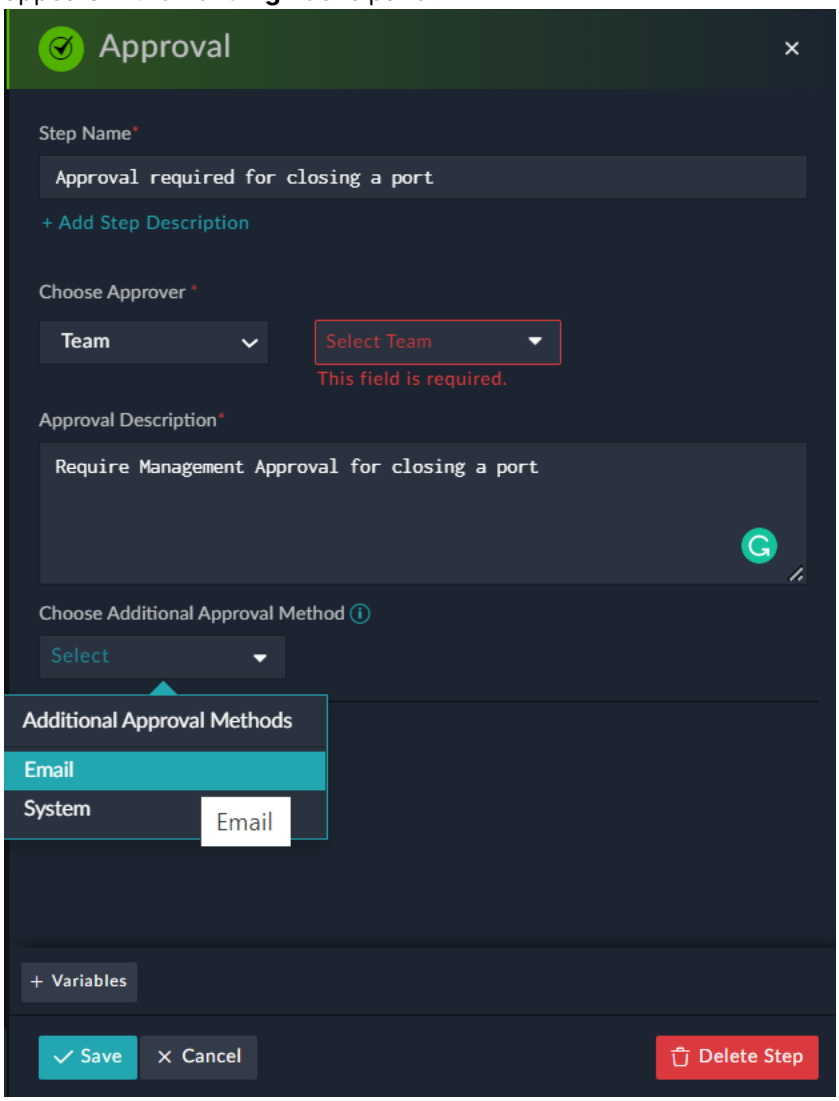
1. Open FortiSOAR and click **Automation** > **Playbooks** in the left navigation bar.
2. On the `Playbook Collections` page, click on an existing playbook collection.
   This opens the `Playbook` page, click on the playbook in which you want to add an approval step.
   This opens the playbook in the `Playbook Designer`.
3. Click the **Approval** step in the `Evaluate` section.
4. For the Approval step, in the **Step Name** field, add the name of the step.
5. From the **Choose Approver** drop-down list, choose **Team** or **User**, which displays a **Select** link. Clicking the **Select** link displays a `Team` or `User` pop-up based on the approver type you have chosen.
   The Team or User pop-up lists all the existing teams or users. Select the team or user who can provide

approval. If you select **Team**, then any member of that team can provide the approval.



6. In the **Approval Description** field, add the description of the approval request, can include the reason for the approval request.
   The approvers can view this description in the approval notification.
7. (Optional) To add system or email as a mode of approval from the **Choose Additional Approval Method** drop-down list, choose **System** or **Email**.
   If you do not choose anything from this drop-down list, then the approval notification appears only in the **Pending Tasks** panel.
   If you choose System, then the approval notification appears in both the **Notifications** and **Pending Task** panels.
   If you choose **Email**, you will receive a notification email for the approval and the approval notification will

appears in the **Pending Tasks** panel.

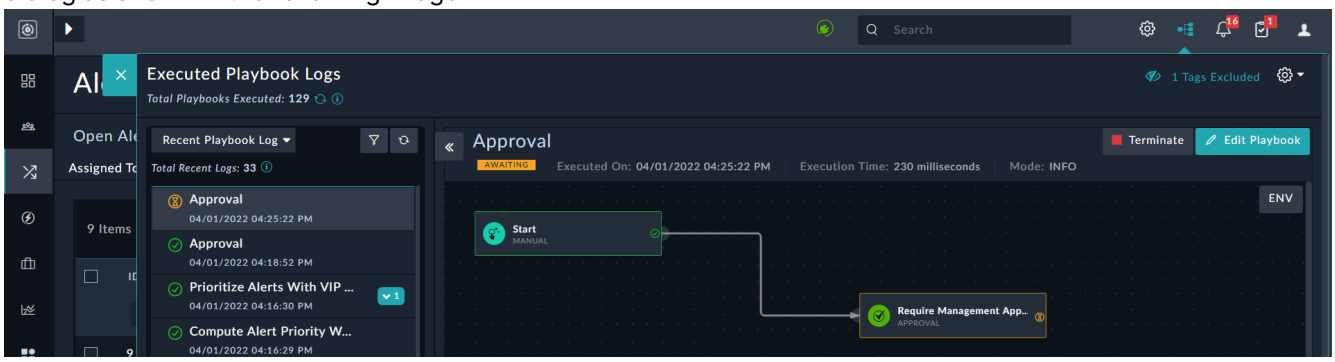


8. Click **Save** to save the approval step.

Playbook status for Approvals

Click the **Executed Playbook Logs** icon in the upper-right corner of FortiSOAR to view the logs and results of your executed playbook. Clicking the **Executed Playbook Logs** icon displays the Executed Playbook Logs

dialog as shown in the following image:

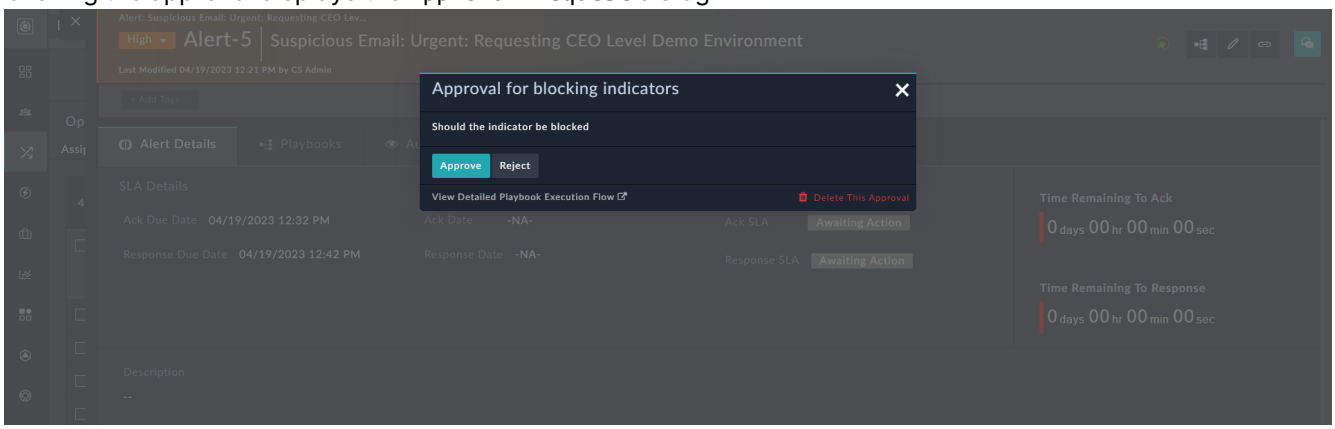


- Until the approval is not received, the **Execution Playbook Logs** will display the Playbook status as `Awaiting`.
- If the approval is rejected or granted, the **Execution Playbook Logs** the playbook continues to execute remaining steps, as defined for approval rejection or acceptance, and if the playbook completes executing all the steps, the **Execution Playbook Logs** will display the Playbook status as `Finished`.

For information on Execution History, see the Playbook Execution and Debugging chapter.

Approval notification using the System mode

Once you complete adding an approval step which includes adding the approvers, the approvers get a notification for approval. Users who have the appropriate permissions for approval receive the notification. You can view the notifications for approvals in the **Approvals** tab of the **Pending Tasks** panel. The **Pending Tasks** icon is present on the top-right corner in FortiSOAR, and whenever a user gets an approval request the number present on these icons increases by '1'. Click the **Pending Tasks** icon to open the 'Pending Tasks' panel, where you can perform actions required for approval and to see detailed information on the same. Click the **Approvals** tab, to open the list of pending approvals. The approvals contain the description for the approval request. Clicking the approval displays the `Approval Request` dialog:



The approver can click **Approve** or **Reject** to approve or reject the request.

When a user logs into FortiSOAR and uses the system method to approve a request, FortiSOAR displays 'Unauthorized access', though the original playbook resumes the execution of the remaining playbook steps and moves to the 'Finished' state (if there are no further errors in the playbook). This is because the 'Approval' module inserts an approval record using a playbook the ownership of that record always remains with the SOC team (admin team) irrespective of the team or user who triggered the playbook. To solve this issue, you must add the team(s) who will provide approval to be part of 'Playbook Appliance.' For more information about Appliances, see the Security and Access Management chapter in the "Administration" Guide.

Once an approver completes an approval request, the notification dialog displays the approval request as approved using a green check symbol, and the notification is removed from the notification window.

The number of historical (legacy) approvals, i.e., those that were in place before your FortiSOAR instance was upgraded, are not displayed in the **Pending Tasks** icon. However, the historical approval count is considered after you click the **Pending Tasks** icon.

Approval notification using the Email mode for releases prior to 7.4.1

If you have chosen **Email** (only in the case when you are on releases earlier than 7.4.1) in addition to the system as a mode of approval, an email will be sent to the email ID that has been configured for the users in their profile. See the Security and Access Management chapter in the "Administration" Guide for more information on configuring user profiles.

If you have selected a team to provide approval in the approval step, then the email notification is sent to all the team members, who have appropriate permissions, and any of the team members can provide approval.

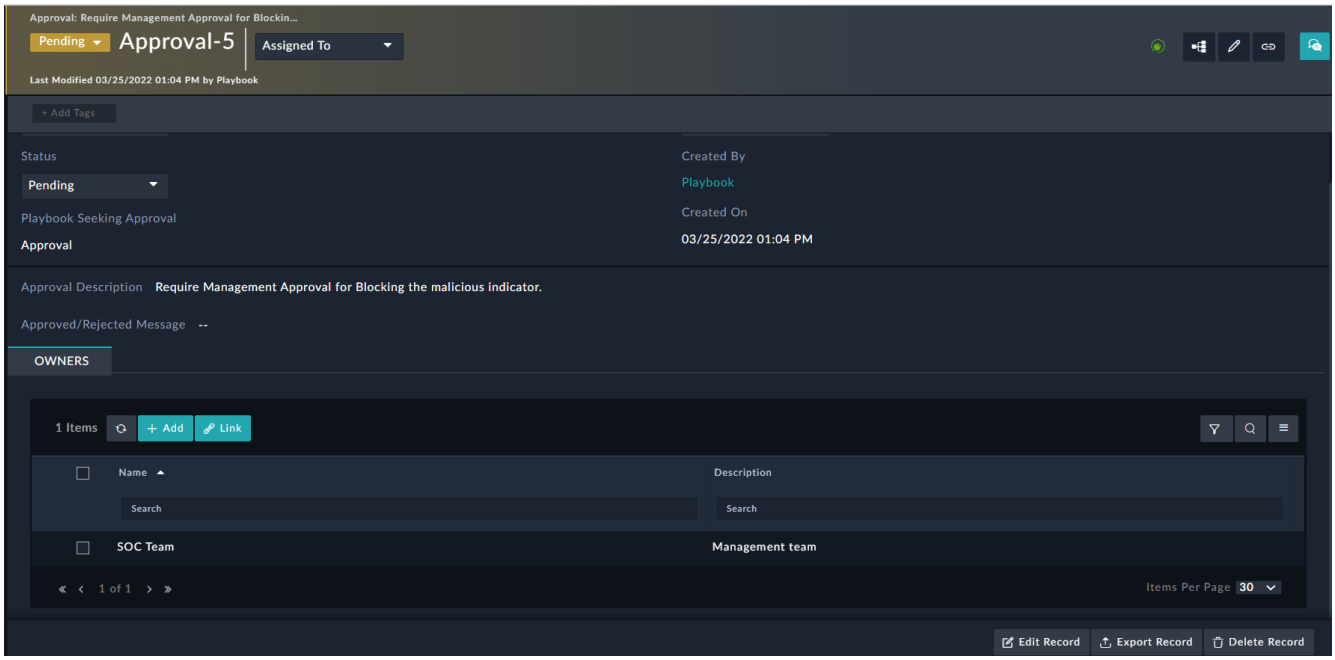
The approval email notification contains a link to an Approval Request dialog, which contains the name of the playbook from which the request has been sent and also the description of the approval required. Once an approver clicks on the link in the email the `Approval Request` dialog is displayed, and the approver can click **Approve** or **Reject** to approve or reject the approval request. The approver can add comments in the **Comments** field that explain the reason for the approval or rejection of the request.

Users can choose to approve the request using the system mode as well since apart from the email notification; a system notification is also sent for the request. If a user uses the system method to approve a request, then FortiSOAR displays 'Unauthorized access', though the original playbook resumes the execution of the remaining playbook steps and moves to the 'finished' state. This is because when the approval record is inserted using a playbook, then the ownership of that record always remains with the SOC team (admin team) irrespective of the team or user who triggered the playbook. To solve this issue, you must add the team(s) who will provide approval to be part of 'Appliance.' For more information about Appliances, see the 'Security' chapter in the "Administration" Guide.

Once an approver completes an approval request, using any mode of approval, the notification dialog displays the approval request as approved using a green check symbol, and the notification is removed from the notification window.

Viewing details of an approval record prior to 7.4.1

Once you trigger a playbook a record for the same is created in the Approvals module, and you can view and edit the details of the approval in this record as shown in the following image:



The `Approvals` module is not included as part of the default modules. Therefore, you must add the Approval module using the **Navigation Editor** if you want the Approvals module to appear in the FortiSOAR left navigation. For information on how to add modules to the FortiSOAR left navigation, see the Navigation Editor topic in the *Application Configuration and Customization* chapter of the "Administration Guide."

You can edit details of approval add or edit the description of the approval or update the approval or rejection message. You can also reassign the task of approval to another user in cases such as the user to whom the approval was originally assigned is unavailable.

When you reassign the approval to another user that user will not get the notification of that assignment unless you have chosen **Email** as the additional method of approval while configuring the **Approval** step. If you have only configured the **System** method of approval, then the reassigned user will not get an approval request notification.

Viewing details of the approval playbooks

You can view the details of the approval by clicking the *Execution History* tab to view the logs and results of your executed playbook. For more information on Playbook Execution History, see the Playbook Execution and Debugging chapter..

Using the output of the Approval step in other playbook steps

To use the output of the approval step in other playbook steps or to display the result of the approval step in the **Step Results** option in Dynamic Values, you must add the following jinja to the step that requires to use the output of the `Approval` step:

- To get result of the approval, i.e. true or False: `{{vars.steps.<nameOfTheApprovalStep>.approved}}`
- To get the comment or message associated with the approval: `{{vars.steps.<nameOfTheApprovalStep>.message}}`
- To get the user who is the approver: `{{vars.steps.<nameOfTheApprovalStep>.user}}`

Tips:

- **Reference Approval Results**: To use the outcome of the approval step in later playbook steps, reference the approval output using Jinja expressions::
  - Approval Result (True/False): `{{vars.steps.<nameOfTheApprovalStep>.approved}}`
  - Approval Comment or Message: `{{vars.steps.<nameOfTheApprovalStep>.message}}`
  - Approver's Username: `{{vars.steps.<nameOfTheApprovalStep>.user}}`
    These expressions allow dynamic incorporation of the approval decision into the playbook's workflow.

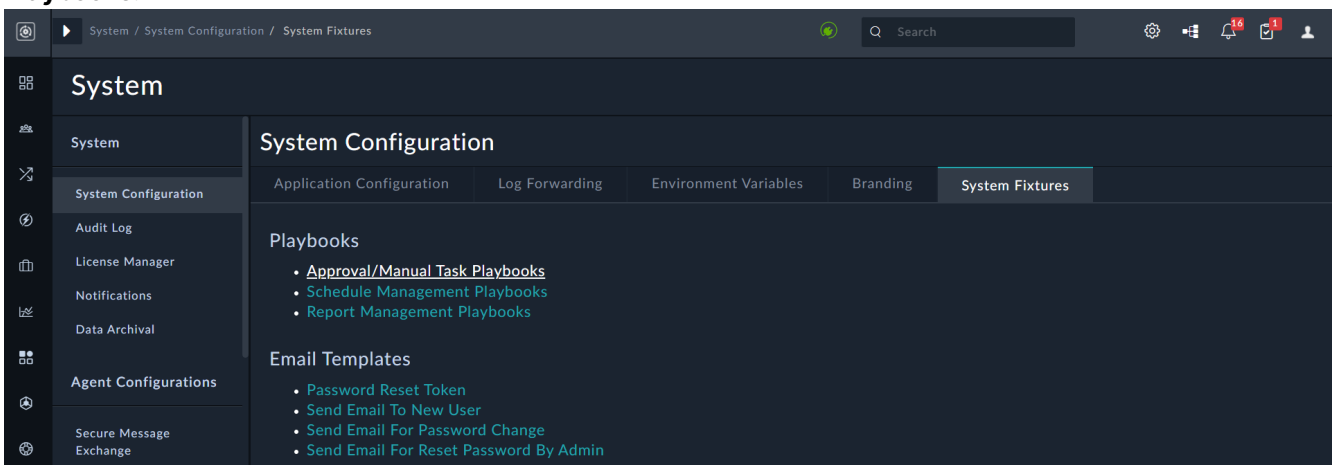
Manual Task

Use the **Manual Task** step to pause the execution of the playbook till you complete a manual task such as a manual shutdown of a server, or starting or stopping a firewall, that is part of an automated workflow.

Once you click the **Manual Task** step, a form containing the fields from the `Task` module is displayed. Enter content for the fields in the `Task` module, such as the name of the task, person to whom the task is assigned, the status of the task, and the date by when the task is to be completed. Once you click **Save**, this record is added in the `Task` module, and a FortiSOAR system-playbook begins to run in the background, which keeps checking the status of this task.

Once a user changes the **Status** of the added manual task in the `Task` module, to either **Skipped** or **Completed**, then the system-playbook gets notified about the status change and in turn the system playbook resumes the execution of the original playbook that had requested the manual task.

Note: You can change the condition for when the manual task should resume, for example, you can specify that the manual task should resume only when the user changes the **Status** of the manual task to **Completed**. You must update the `System` playbooks if you want to configure the manual task conditions. You can view system playbooks by clicking the **Settings** icon, then clicking the **System Configuration** option, and then clicking the **System Fixtures** tab. On the `System Fixtures` page, in the `Playbooks` section, click **Approval/Manual Task Playbooks**.



Using the output of the Manual Task step in other playbook steps

To use the output of the manual task step in other playbook steps or to display the result of the manual task step in the **Step Results** option in Dynamic Values, you must add the following jinja to the step that requires to use the output of the `Manual Task` step:

- To get the ID of the manual task: `{{vars.steps.<nameOfManualTaskStep>['task_data']['@id']}}`
- To get status of the manual task: `{{vars.steps.<nameOfManualTaskStep>.status}}`

Tips

- **Monitor Task Status**: Use the following Jinja expressions to reference the output (task status or ID) of the manual task step in later playbook steps
  - Task ID: `{{vars.steps.<nameOfManualTaskStep>['task_data']['@id']}}`
  - Task Status: `{{vars.steps.<nameOfManualTaskStep>.status}}`
    These expressions allow you to reference the task's data, such as its status or ID, in later steps of the playbook for decision-making or logging purposes.
  - **Resumption Upon Completion**: Once the manual task is marked as completed or skipped, the playbook will automatically resume execution from where it was paused.

Manual Input

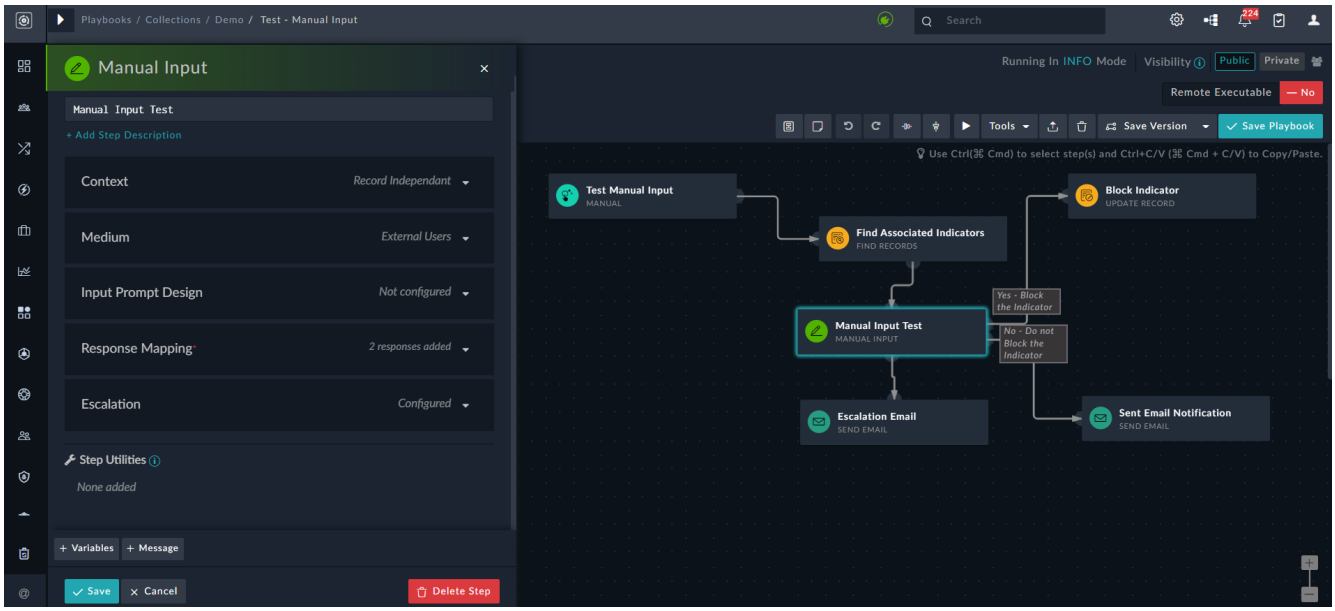
Use the Manual Input step to display a customized pop-up either for a quick confirmation (simple decisions input form) or for input prompts with custom form fields anywhere in the flow of the playbook. Based on the input or decision provided by the user, the playbook chooses one of the paths from the paths that you have defined in the playbook and continues to execute the playbook as per the specified automated workflow.

To execute a manual input playbook, you must be assigned a role with a minimum of `Read` permission on the module containing the record on which the manual input playbook requires to be executed, as well as the `Execute` permission on the `Playbooks` module.

The manual input step can be used with all types of playbook triggers, including Custom API Endpoint trigger and Referenced trigger.

In FortiSOAR release 7.3.1, Manual Input has been enhanced to make it more effective to use by dividing the manual input into various segments that define various configurations required for manual input. For example, the 'Context' segment defines the context of manual input, i.e., whether it runs independent of records or on the

selection of records.



Support has also been added for 'Slack' and 'Microsoft Teams' as a channel for delivery of manual inputs prompts and for seamless integration between Slack, Microsoft Teams and FortiSOAR. Before you can use Slack or Microsoft Teams as delivery channels, you must install and configure their respective solution packs, connectors and applications, FortiSOAR For Slack application or FortiSOAR For Microsoft Teams application and install the FortiSOAR For Slack Solution Pack. For information about the Slack and Microsoft Teams connectors, applications, and solution packs, see the FortiSOAR Content Hub Portal.

In the case of a Custom API Endpoint trigger, a Referenced trigger, or a Manual trigger that has been created with the **Run Without Selecting Any Record** option selected, you must specify the module on which the action has to be taken. The module specified will also be used to populate the "People" lookup and assign ownership to specific users or teams, as well as the record fields that require inputs.

An example of an "Input Form" prompt would be the enrichment of indicators associated with an alert record in FortiSOAR that has been generated from a SIEM. Enrichment of indicators would be done using threat intelligence tools, for example, VirusTotal. The results from VirusTotal state that there are 3 indicators, 2 of which are marked as suspicious based on their score received from VirusTotal and 1 is marked as malicious based on their score received from VirusTotal. The Manual Input step would list these 3 indicators and prompt SOC analysts for an evaluation of the indicators and select the ones that they think should be marked as malicious. Based on the analyst's evaluation, further action will be taken on the alert record and the associated indicators. You can add visibility conditions to the fields displayed in the user input form, i.e., fields in the user form are visible based on the specified conditions. For an example of using the visibility conditions in a user prompt, see the User Input Prompt - Visibility Conditions section.

Dynamic list is supported as a '**Custom**' input type in both the Manual Trigger step and the Manual Input step. A 'Dynamic list' is a list with dynamic values that are set using a playbook, i.e., the options of the list are defined using JSON or comma-separated values are displayed as a list in a user input prompt. Dynamic Lists also provide an option that allows users to select multiple options from the input prompt. For an example of how to use dynamic lists in a user prompt, see the User Input Prompt - Dynamic Lists section.
From release 7.3.0 onwards, you can also select **Email Template Field**, as a field Type. For more information, see User Input Prompt - Using the 'Email Template' Field topic.

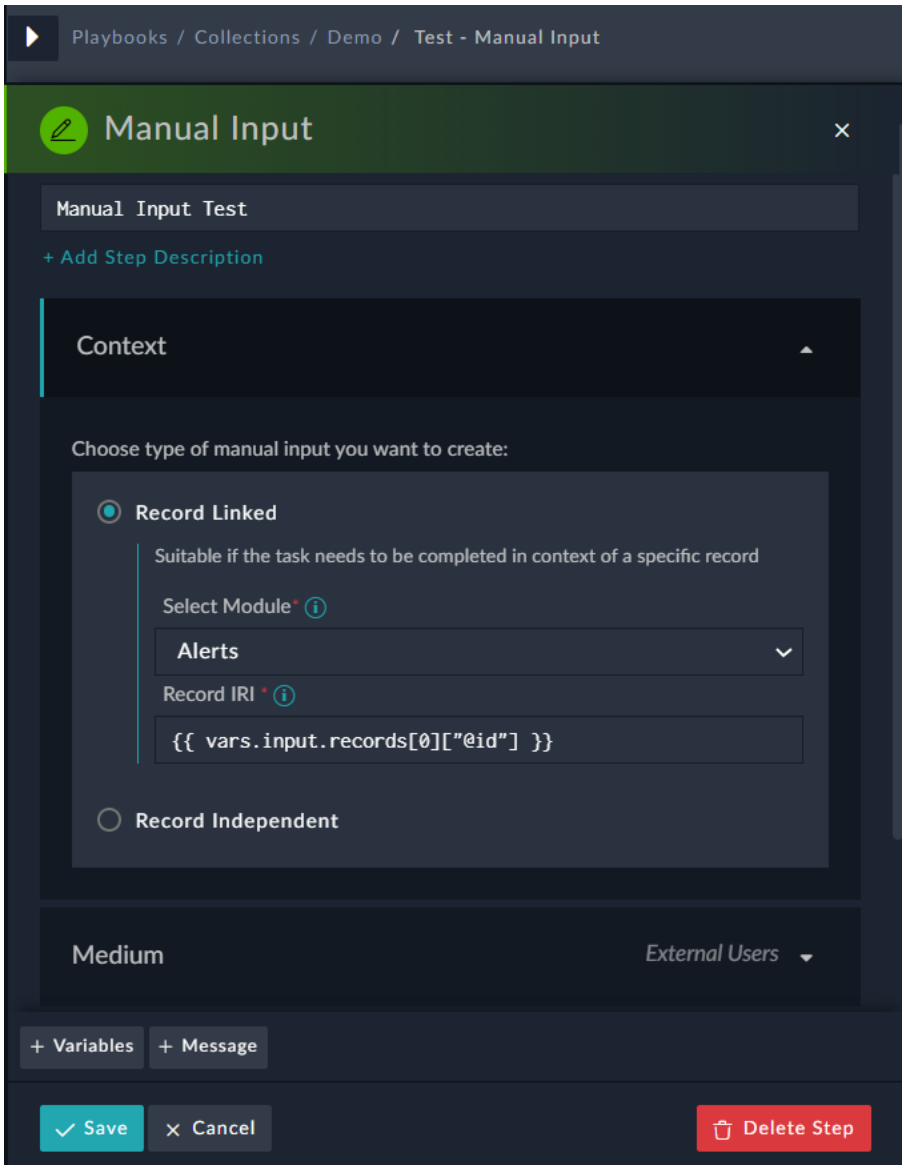
An example of a "Simple Decision" prompt would be similar to the above except that there would generally be a question in the prompt based on which the SOC analyst would be required to make a decision. For example, 'Is

the following Indicator Malicious?' The analyst then just has to choose either "Yes - Block the Indicator" or "No - Do not block the indicator". Based on the SOC analyst's decision, further action will be taken on the alert record and the associated indicators. You can also retrieve the reputation of the indicator from various threat intelligence tools such as VirusTotal using the Connector step and display this information to the analysts to enable them to make a more informed decision.

Building a Simple Decisions input prompt

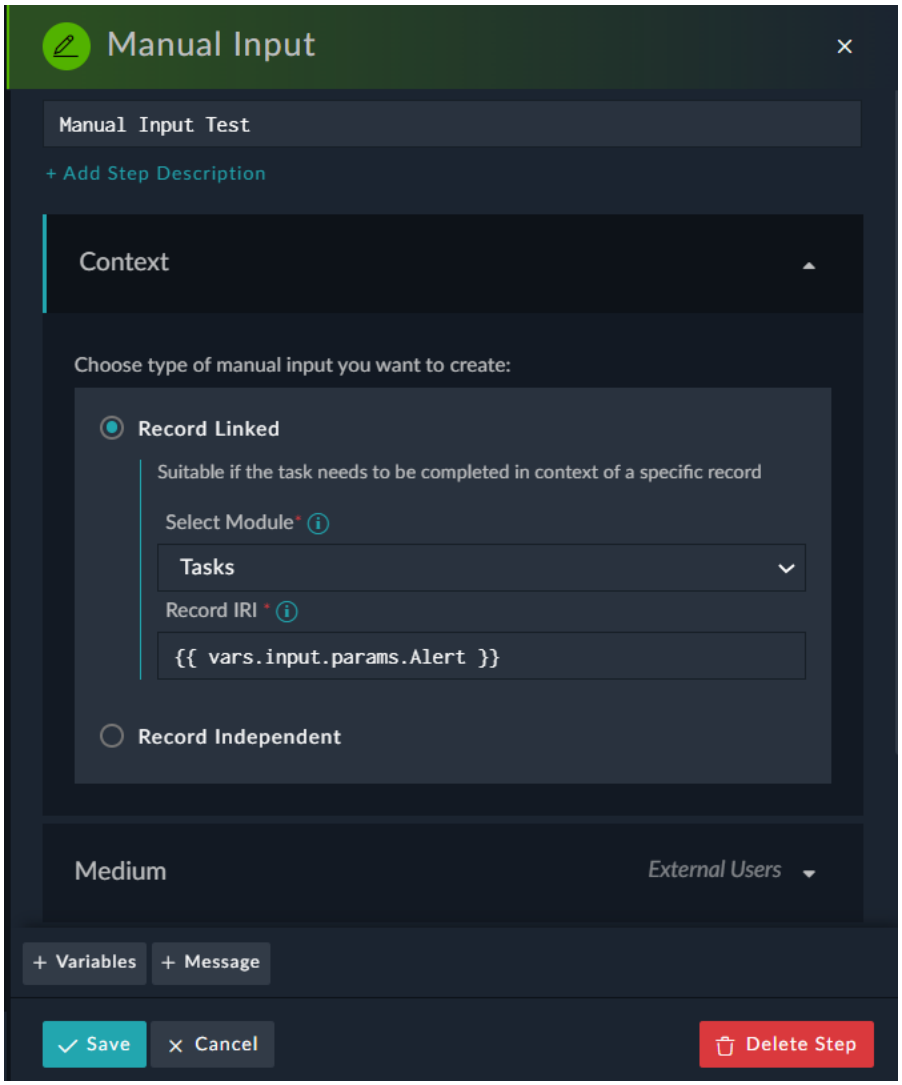
Perform the following steps to create a playbook with a Manual Input playbook based on prompting SOC analysts for an evaluation of indicators that are associated with an alert generated in FortiSOAR and confirm whether they are malicious or not. Based on the analysts' evaluation, further action is taken on the alert record.

1. Open FortiSOAR and click **Automation** > **Playbooks** in the left navigation bar.
2. On the `Playbook Collections` page, click on an existing playbook collection.
   This opens the `Playbook` page. Click on the playbook in which you want to add the **Manual Input** step, or add a new playbook, which opens the playbook in the `Playbook Designer`. For our example, create a new playbook named `Test - Manual Input` and ensure that the **Active** checkbox is clicked, then click **Create**. This opens the Test - Manual Input playbook in the Playbook Designer.
3. In the Trigger step, select **Manual Trigger** and define the following parameters:
   a. In the **Step Name** field, enter the name of the playbook. For example, `Test Manual Input`.
   b. In the **Trigger Button Label** field, type the playbook name as `Indicator Evaluation`.
   c. Ensure that the **Run once for all selected records** option is selected.
   d. In the **Choose record modules on which the playbook would be available on** field select the **Alerts** module.
   e. Click **Save**.
4. To get the indicators associated with the record, you can add a **Find Records** step with the **Indicator** module selected, and in the `Build Search Query` section, select Alerts in the Related Module, and then using Dynamic Values, add the condition as `ID Equals {{vars.input.records[0].id}}` and click **Save** to save the step.
5. Add the steps that you want to add as the response actions to evaluate the inputs provided by the analysts. For our example, configure the **Block Indicator** and **Send Email Notification** steps as per your requirements.
6. Select **Manual Input** from the `Evaluate` section, and define the following parameters:
   a. In the **Step Name** field, enter the name of the playbook. For example, `Manual Input Test`.
   b. Click the **Context** menu to choose the context in which you want to run the manual input from the **Choose type of manual input you want to create** field.
   Choose the **Record Independent** option to create a Global Manual Input , i.e., this type of manual input does not depend on any record. Choose the **Record linked** option if you need the task to be completed in the context of a specific record. For our example, we will retain the selection of **Record linked** with its default options for module and triggered record IRI:



NOTE: When you choose the the **Record Independent** option, then the **Analyst working on the record** and **All owners of the record** options are not available in the Medium section.

You can also choose to display the manual input in a record that is different from the record that triggered the manual input playbook. For example, if a manual input playbook is triggered on a related task module in an alert record, you can still choose to display the manual alert on the alert record. To do this, select the module on which you want to run the manual input playbook from the **Select Module** drop-down list and update the triggered record IRI value in the **Record IRI** field of the record on which the manual input will be prompted, as shown in the following image:

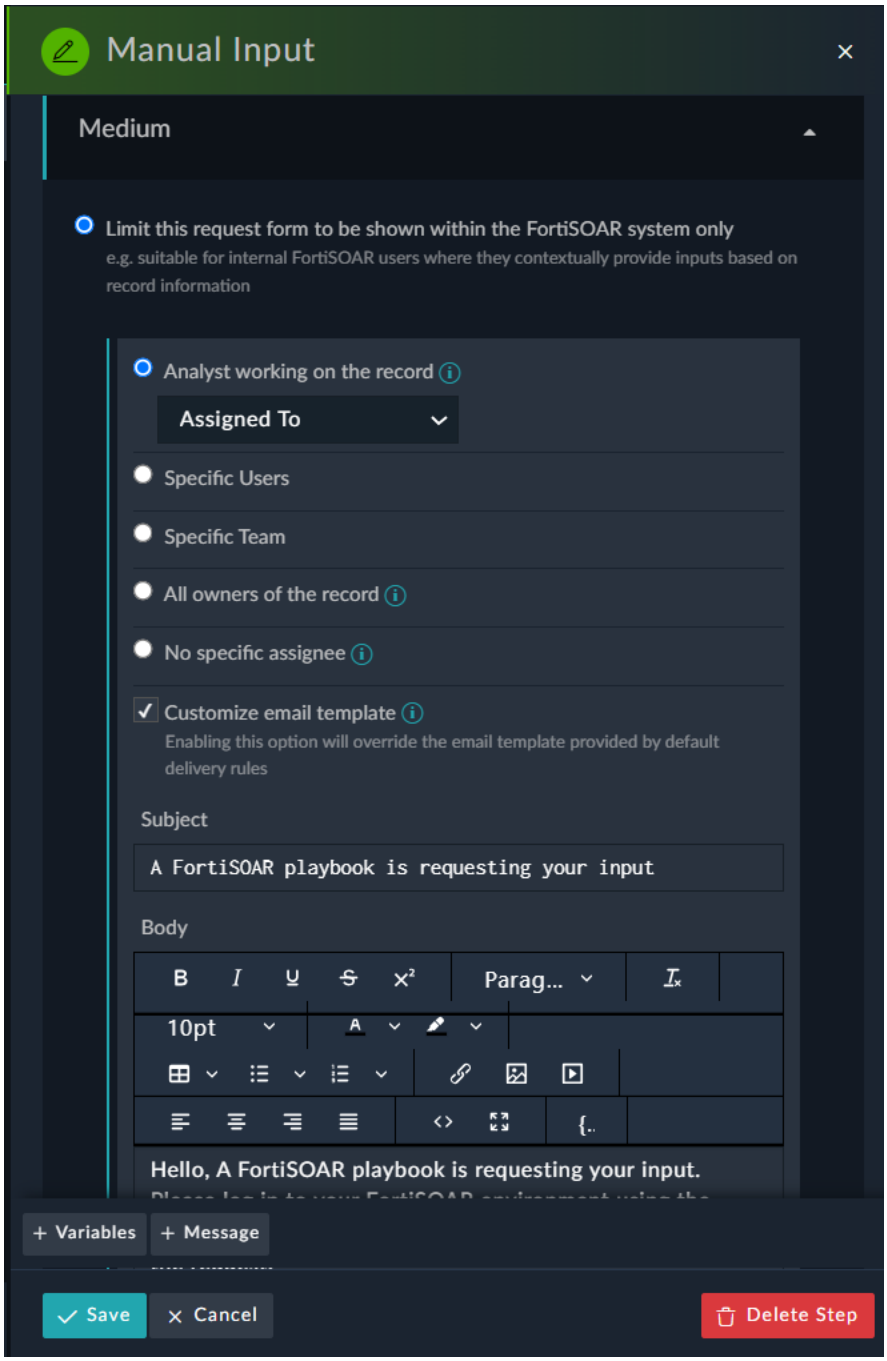


c. Click the **Medium** menu to choose the medium of delivery for the manual input, and also whether this manual input will be limited to users within FortiSOAR or also be accessible to non-FortiSOAR users to provide inputs.

- Select the **Limit this request form to be shown from within the FortiSOAR system only** option if you want the manual input to be accessible to only FortiSOAR users. Once you select this option, then you must select one of the following options to determine who is responsible for responding to the input prompt:
  - **Analyst working on the record**: The manual input is visible and actionable by the analyst who is working on the record. In this case, select the field that is used to assign the user corresponding to the specific module, i.e., the `People` lookup for that module. For our example, since we are working with the `Alerts` module, select **Assigned To**. You can also choose the **Created By** or **Modified By** options.
    The record assignee is defined as the user mapped to the assignment field in the record, at the time of playbook execution. The manual input stays assigned to this user, even if the record assignee is changed at a later time.
  - **Specific Users**: The manual input is visible and actionable by users, other than the user who is assigned to the record, who need to provide the input. When you select this option, then the

People multi-select list appears from which you can select users who require to take the decision. You can also add a custom expression in this field.

- **Specific Team**: The manual input is visible and actionable by team(s) who requires to provide the input. This means that any user who is part of the selected team(s) will be able to provide the input. When you select this option, then the **Team** multi-select list appears from which you can select specific teams(s) that can provide their input. You can also add a custom expression in this field.
- **All owners of the record**: The manual input is visible and actionable by all the owners of the record, i.e., users who have permissions on the record, at the time of execution of the playbook.
- **No specific assignee**: The manual input is visible and actionable to everyone in the FortiSOAR instance.



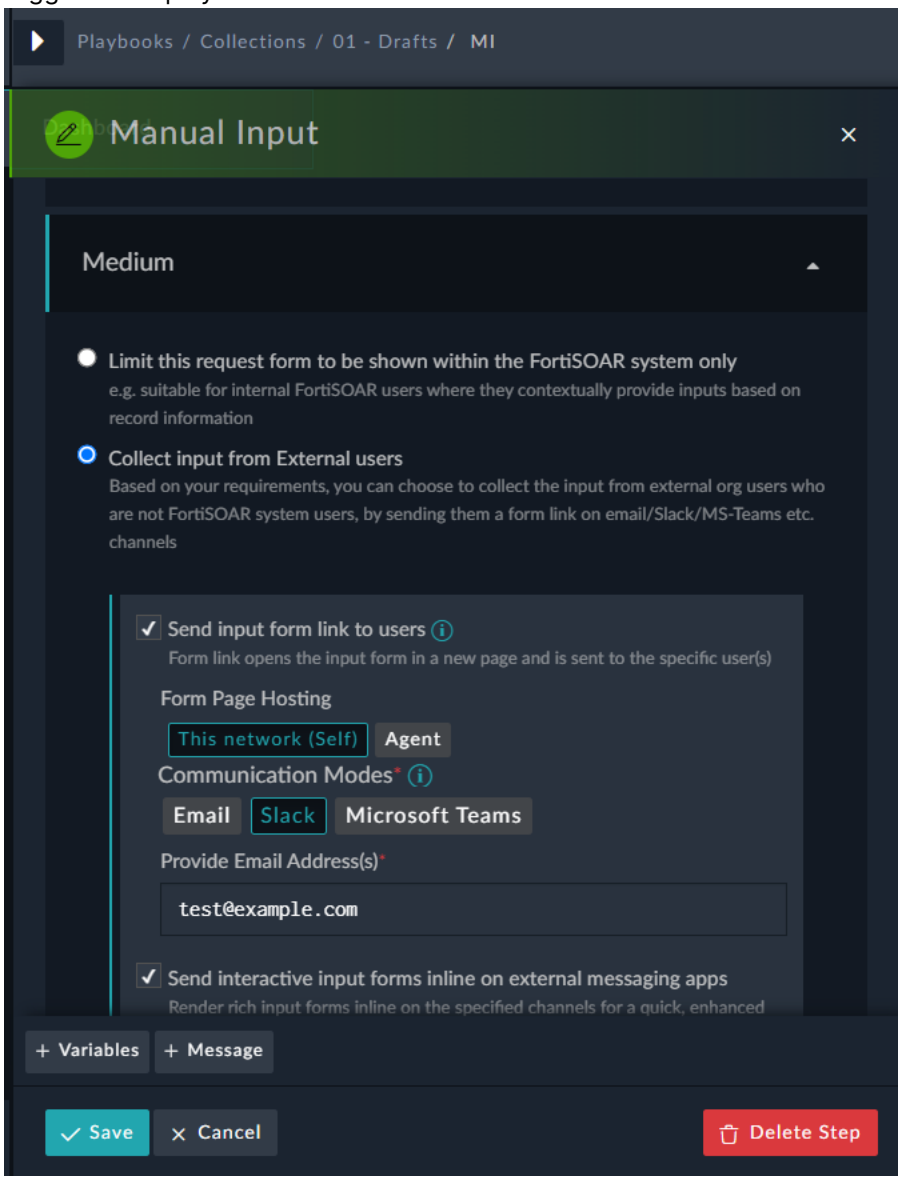
Note: The teams or users who are specified as owners, i.e., to whom this task is assigned, must have access to the record and appropriate permissions to perform the steps required to complete the task.

Additionally, starting with release 7.4.2, select the **Customize email template** checkbox to define a custom email body using a rich text field rather than sending the default template in order to provide context for the manual input or approval. Selecting this checkbox allows users to personalize manual input notifications for each record and include playbook variables using custom Jinja input in the email body.

Additionally, you can specify a list of record IRIs or file IDs for attachments that you want to add to the email in the **Attachment IRI List** field.

- Select the **Collect input from external users** option, if you want to non-FortiSOAR users to provide decisions or inputs. In this case, either the actual input form or a link to a page that contains the input form is sent to the delivery medium for users to provide their responses.
  - Select the **Send input form link to users** option, to send a link to a page that contains the input form to users. Clicking the input prompt link opens the form in a new page, where users can provide their responses.
    - From the **Form Page Hosting** options, choose from where you want the input form page to be available. Select **This network (Self)** to host the input form on the same network. Select **Agent** (if applicable) to host the input form on the agent's network. For more information see, the Running unauthenticated manual inputs in segmented networks using FSR Agents topic.
      Important: If you have deleted an existing Agent and then added it again with the same name, then you will encounter a "`Server Error`" when you run the manual where you have chosen to host the manual input form link on the Agent network and chosen Email as the delivery medium. Therefore, when you are adding an Agent again (after deleting the existing one) ensure that you give the re-added Agent a new name that is distinct from the old one.
    - From the **Communication Modes** list, choose the communication mode to be used to deliver the input prompt link. The available options are **Email**, and **Slack**, **Microsoft Teams**. It is recommended that you choose only one communication mode for delivery of manual inputs.
      Note: 'Slack' is displayed only if you have installed and configured the Slack connector, FortiSOAR For Slack Application, and FortiSOAR For Slack Solution Pack. Similarly, 'Microsoft Teams' is displayed only if you have installed and configured the Microsoft Teams connector, FortiSOAR For Microsoft Teams Application, and FortiSOAR For Microsoft Teams Solution Pack.
      For information about configuring Slack or Microsoft Teams and getting them ready for integration with FortiSOAR, see the connector, application, and solution pack documentation on the FortiSOAR Content Hub Portal.
      In the **Provide Email Address** field, add email addresses, in as a list (JSON or comma-separated), of non-FortiSOAR users, who should provide responses in the input form.
      Note: If the server address for the manual input endpoints sent in the email is incorrect, then see the `Manual Input Email - Incorrect Link Error` topic in the Optimizing and Troubleshooting chapter.
      In the case of Slack, add the email address of the user from whom you want the response and who belongs to the same Slack workspace in which the FortiSOAR For Slack app is registered, or use `vars.bot_context.user_id`, for the current users' ID, and `vars.bot_context.channel_id` for the ID of the Slack channel that has triggered the playbook.
      In the case of Microsoft Teams, add the email addresses of users, teams, or channel paths in Microsoft Teams that has triggered the playbook. For more information on supported input types for the **Provide Email Address** field, see the FortiSOAR For Microsoft Teams application document.
  - Select the **Send interactive input forms inline on external messaging apps** option to render the rich input form inline on the selected channels for a quick, enhanced input experience. 'Slack' and 'Microsoft Teams' are the options available in the **Available Channels** list. It is recommended that you choose either Slack or Microsoft Teams to deliver the manual input.
    Note: 'Slack' is displayed only if you have installed and configured the Slack connector, FortiSOAR For Slack Application, and FortiSOAR For Slack Solution Pack. Similarly, 'Microsoft Teams' is displayed only if you have installed and configured the Microsoft Teams connector,

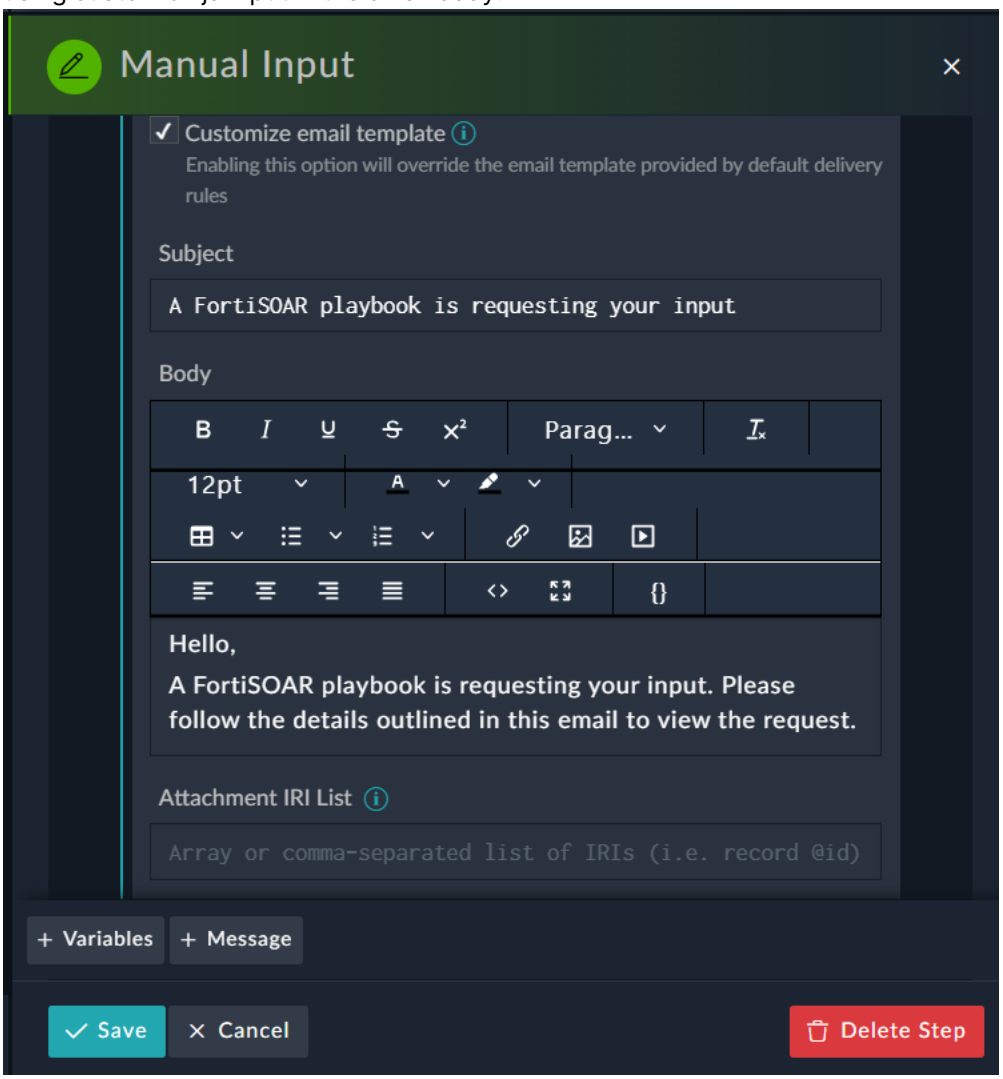
FortiSOAR For Microsoft Teams Application, and FortiSOAR For Microsoft Teams Solution Pack.
For information about configuring Slack or Microsoft Teams and getting them ready for integration with FortiSOAR, see the connector, application, and solution pack documentation on the FortiSOAR Content Hub Portal.
In the **Provide Email Address** field, in the case of Slack, add the email address of the user from whom you want the response and who belongs to the same Slack workspace in which the FortiSOAR For Slack app is registered, or use `vars.bot_context.user_id`, for the current users' ID, and `vars.bot_context.channel_id` for the ID of the Slack channel that has triggered the playbook:



In the case of Microsoft Teams, add the email addresses of users, teams, or channel paths in Microsoft Teams that has triggered the playbook. For more information on supported input types for the **Provide Email Address** field, see the FortiSOAR For Microsoft Teams application document.

Additionally, starting with release 7.4.2, select the **Customize email template** checkbox to

define a custom email body using a rich text field rather than sending the default template in order to provide context for the manual input or approval. Selecting this checkbox allows users to personalize manual input notifications for each record and include playbook variables using custom Jinja input in the email body:



Additionally, you can specify a list of record IRIs or file IDs for attachments that you want to add to the email in the **Attachment IRI List** field

NOTE: By default, an access link to the external manual input is created when the **Customize email template** option is selected.

d. Click the **Input Prompt Design** menu, and configure design your input prompt:

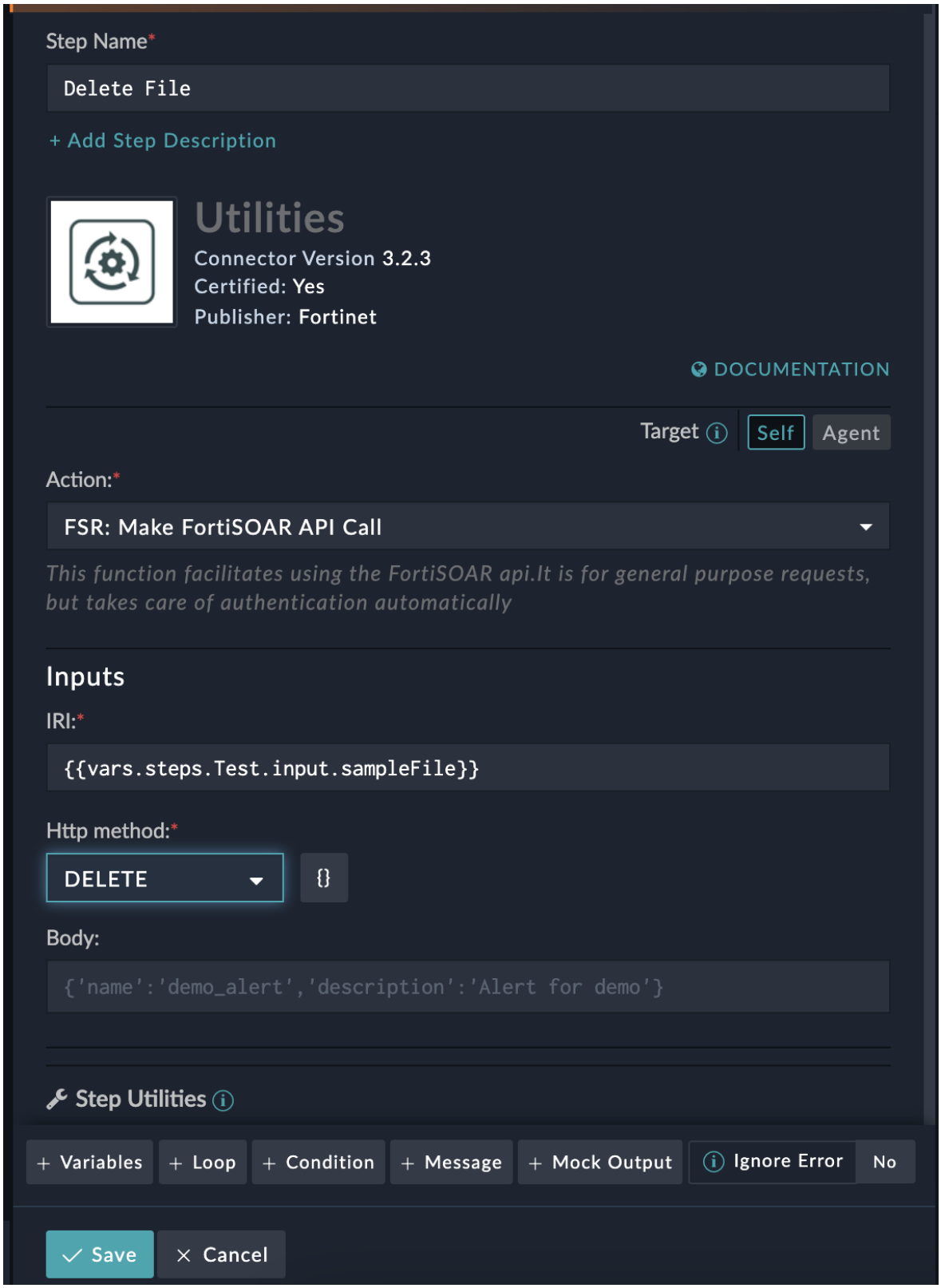
i. In the **Title** field, enter the title for the prompt. For example, `Indicator_Malicious or not`.

ii. In the **Description** field, enter the description for the input prompt. For example, `Should the indicator be blocked?` and add then add the jinja to retrieve the indicators associated with the alert in the format: `{{vars.steps.<nameOfFindRecodsStep[0].value}}`. For example, `{{vars.steps.Find_Associated_Indicators[0].value}}`

To add Jinja in playbooks, use **Dynamic Values** . For more information, see the Dynamic Values chapter.

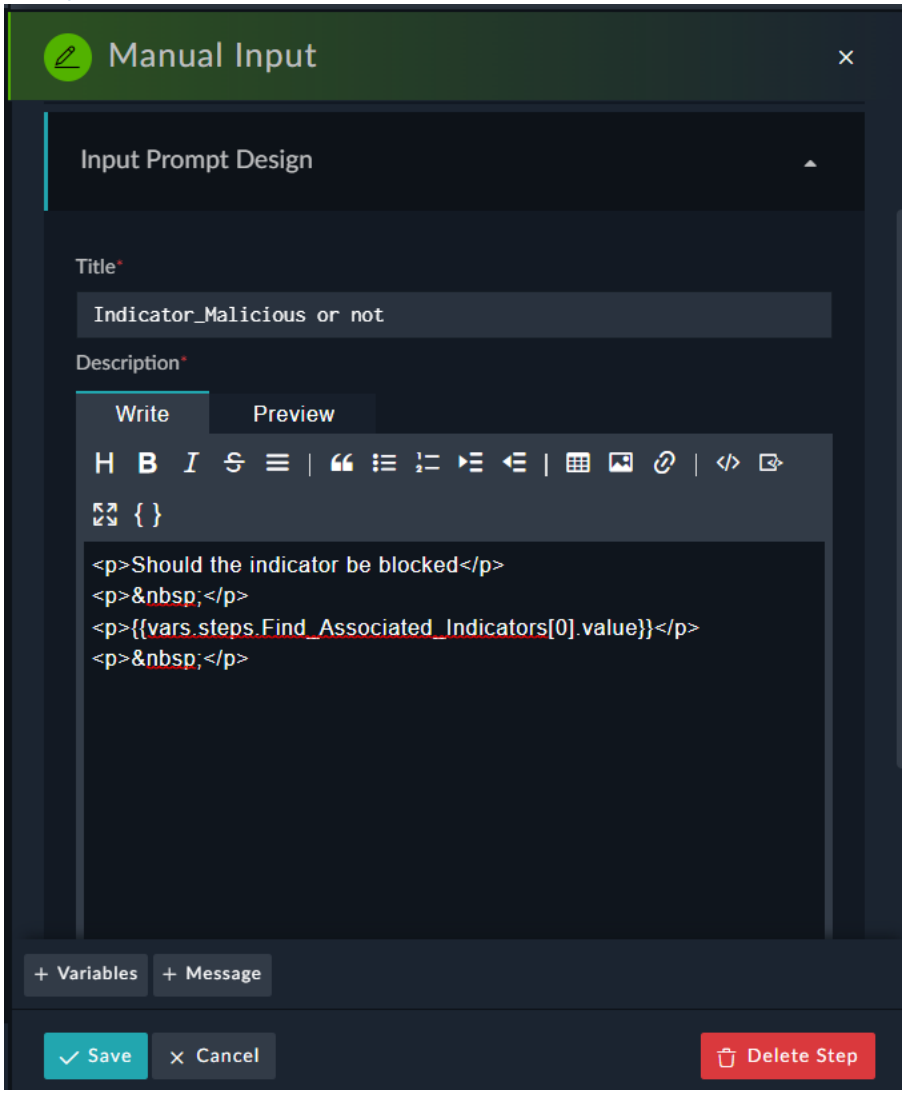
iii. In the `Build Input Prompt` section, build a user prompt to take inputs from users. You can build a customized user prompt form by adding multiple types of input fields of standard field format

within the UI such as Text, Picklist, Lookup, File, Date/Time, Dynamic list, Checkbox, Email Field, etc., or you can just define a simple yes/no type of prompt to get quick confirmations from users or create a prompt with custom form fields.
Note: If you use "File" as part of inputs in the manual input step, i.e., if users can upload files as part of the input prompt, then you must ensure that once the file has been processed in the playbook, it is deleted. You can delete the file using the "Utilities" connector's "`FSR: Make FortiSOAR API Call`" step. In this step, specify the `DELETE` method and the IRI of the file to be deleted.

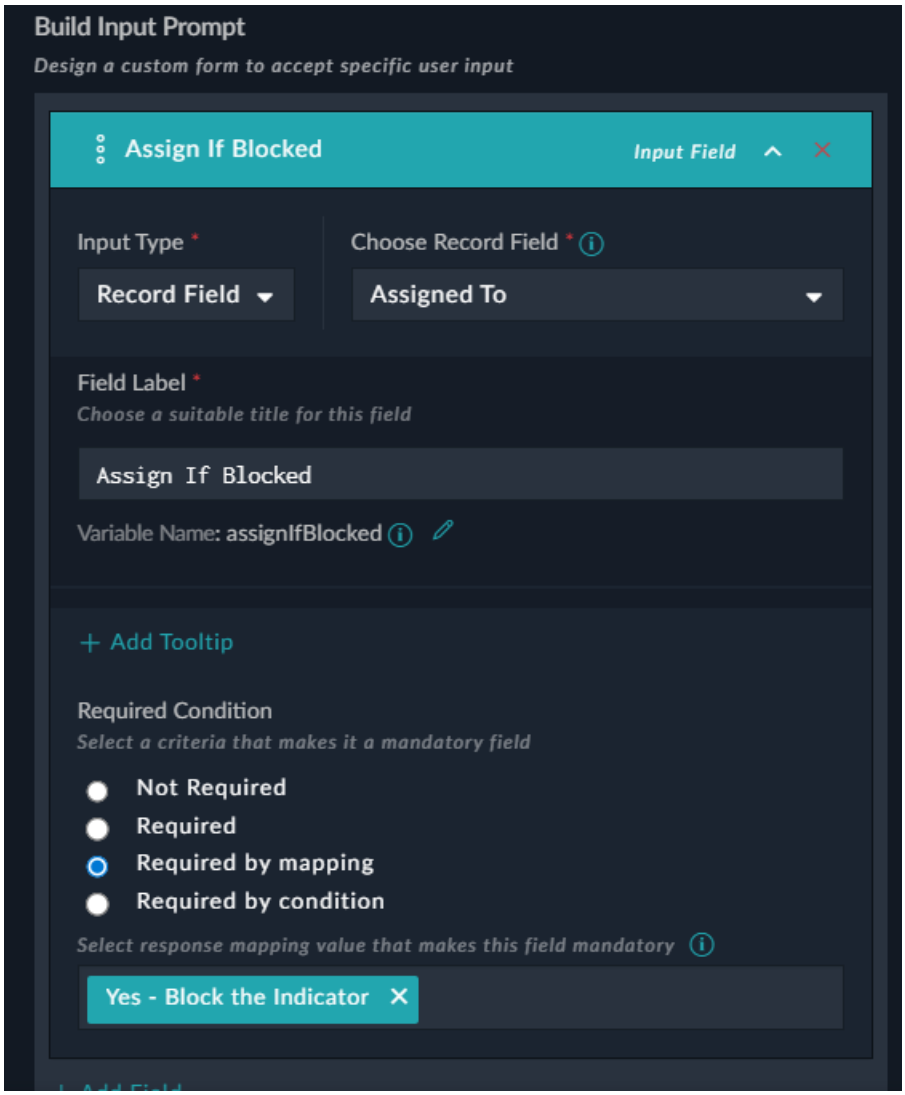


Important: If you choose 'Slack' or 'Microsoft Teams' as the medium to deliver Manual Inputs and get responses from users, then the 'File' and 'Email Template Field' fields are not supported in an

input prompt. If users select any of these fields, then they are not displayed in the message displayed in Slack or Microsoft Teams.
An example of building a user prompt is described in the Manual Trigger section - Building a User Prompt.



Starting with release 7.6.5, a manual input prompt can be designed to allow users to securely provide passwords to FortiSOAR. For details, see User Input Prompt - Using the Password field.
If you build a input prompt you can click the **Add Tooltip** link and enter the description in the **Tooltip** field.
Starting with release 7.6.5, a new option, **+Add Required Condition** is added to the Build Input Prompt, which allows users to specify criteria when a field is made mandatory during step execution. The following options are available:
Not Required: The field is optional.
Required:The field is always mandatory.
Required by mapping: The field becomes mandatory only when a specific response mapping is selected (for example, *Yes, Block the Indicator*):

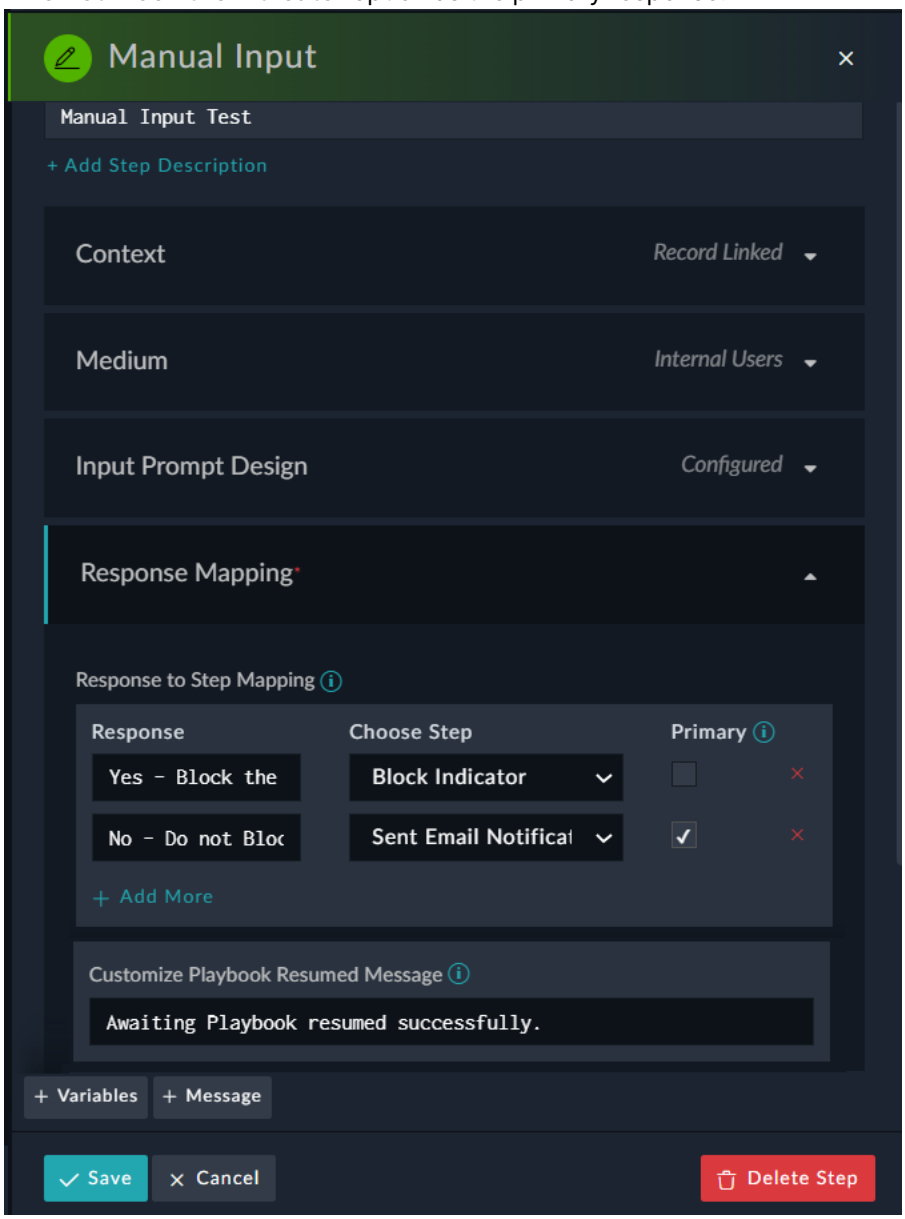


Required by condition: The field becomes mandatory when defined conditions are met.
Prior to this release, users could mark a field as required only by selecting the **Mark as Required Field in Prompt** checkbox.
With the new options, the system can interpret functional differences between steps and selectively apply or ignore required-field status as appropriate.

iv. Click **Save** to save the step.

e. Click the **Response Mapping** menu to add the custom response options that the user can choose from when presented with the decision. You should map each custom response option to a corresponding playbook step.
You can add the custom response for the decision first so that you can define the complete workflow and then create the corresponding playbook steps.
For our example, in the `Response to Step Mapping` section, click the **Add More** link and in the **Response** field, type `Yes - Block the Indicator` and corresponding to this response, in the **Choose Step** field, select the **Block Indicator** step to block the indicator. Then click **Add More** and type `No - Do not Block the Indicator` and corresponding to this response, in the **Choose Step** field, select the **Send Email Notification** step to send a notification to the SOC team (admin team) so that they can be informed that this indicator is not blocked and they can take further steps if required.
You can also select the response that you want to consider as a primary response by selecting the

Primary checkbox. Selecting a response as primary will add a distinct visual style to that option button, making it more prominent when compared to the other buttons. In our example, we have marked the **No - Do not Block the Indicator** option as the primary response.



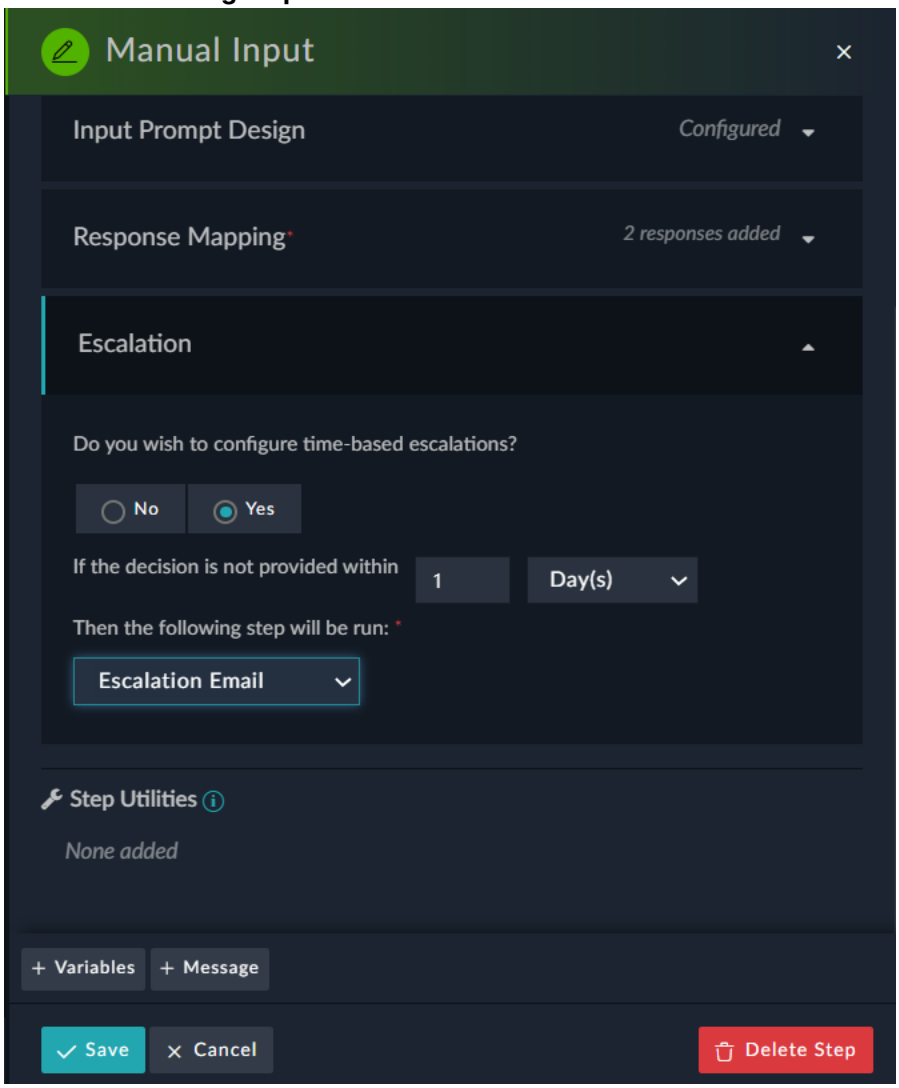
Additionally, you can customize the message that is displayed when a manual input playbook is resumed by typing a custom message in the **Customize Playbook Resumed Message** field. The default system message "`Awaiting Playbook resumed successfully`" is displayed when the manual input playbook is resumed if you do not set any custom message. The custom message can have a maximum of 255 characters.

f. Click the **Escalation** menu to define actions that should be taken in case a decision is not taken within the specified time frame. From the `Do you wish to configure e-based escalation?` section, choose **No** or **Yes**.

If you choose **No**, then there is no time-based escalation.

If you choose **Yes**, then you must specify the following:

- The time within which the action (input or decision) must be taken, in the **If the decision is not provided within** field. You can specify the time in Days, Hours, or Minutes. The minutes option has been added for cases where responses from analysts are quickly required, such as 15-20 minutes. **NOTE**: The maximum duration for an escalation that you can set is 7 days. If no approval or input is provided within this time, the playbook will be terminated, ensuring efficient workflow management.
- The Escalation step must be selected from the **Then the following step will be run** field. For example, if you want to send an email notification to the managers, then you can define that step as `Escalation Email` and connect it to the Manual Input step and choose that option from the **Then the following step will be run** field:



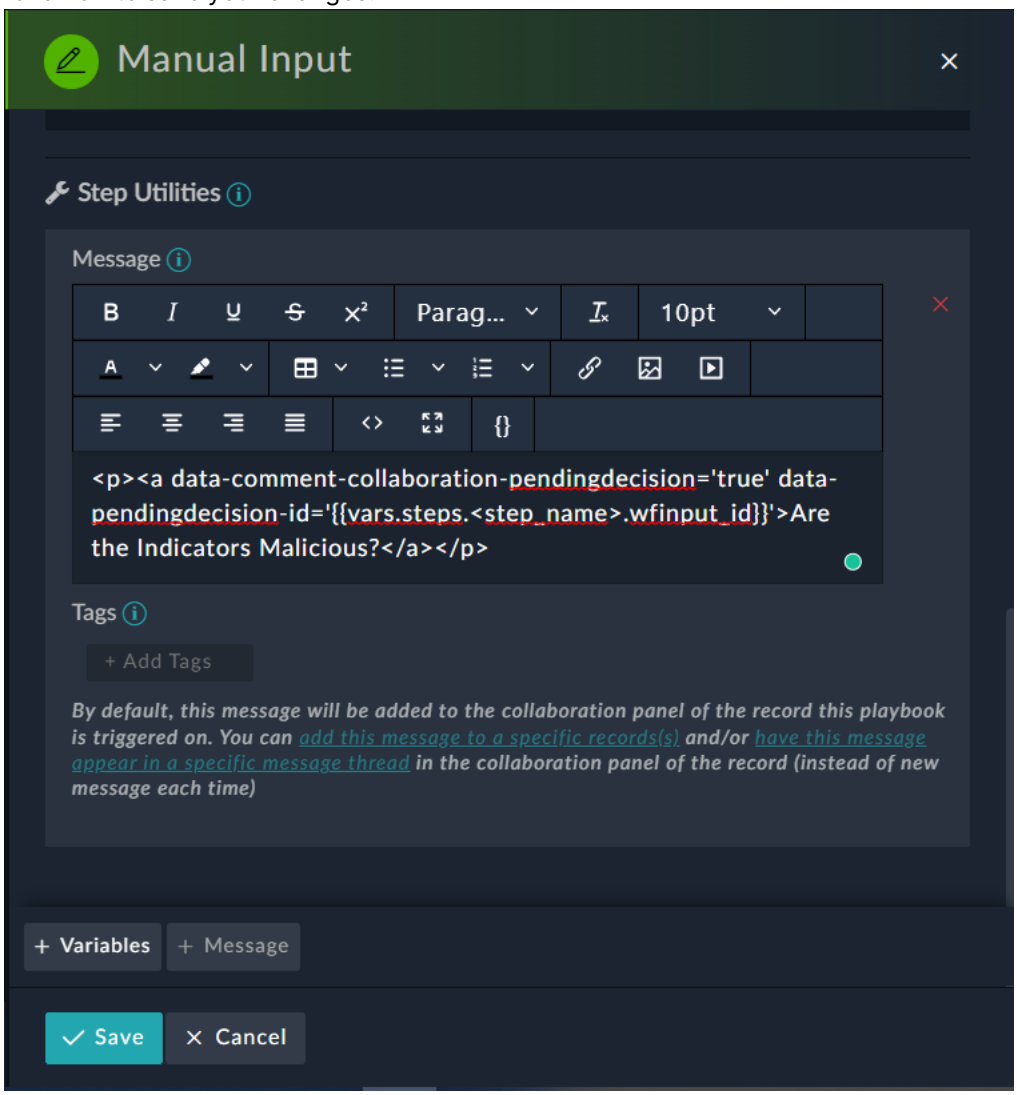
If you are requesting decisions or inputs from non-FortiSOAR users via email, then you can use the escalation settings to define when the links provided in the email will expire. For example, if you select **Yes**, and specify 4 hours in the **If the decision is not provided within** field, this would mean that the links in the email that has been sent for the decision or input would expire in 4 hours. **Note**: FortiSOAR runs a system schedule to resume the workflows that have timed out, such as running the escalation step when the decision is not taken within the specified time. This schedule, by default, is set to run every minute. The cron expression for this system schedule is present in the `/opt/cyops-workflow/sealab/sealab/config.ini` file, and is as follows:

```
MANUAL_INPUT_ESCALATION_SCHEDULE: {'minute': '*', 'hour': '*', 'day_of_week': '*',
```

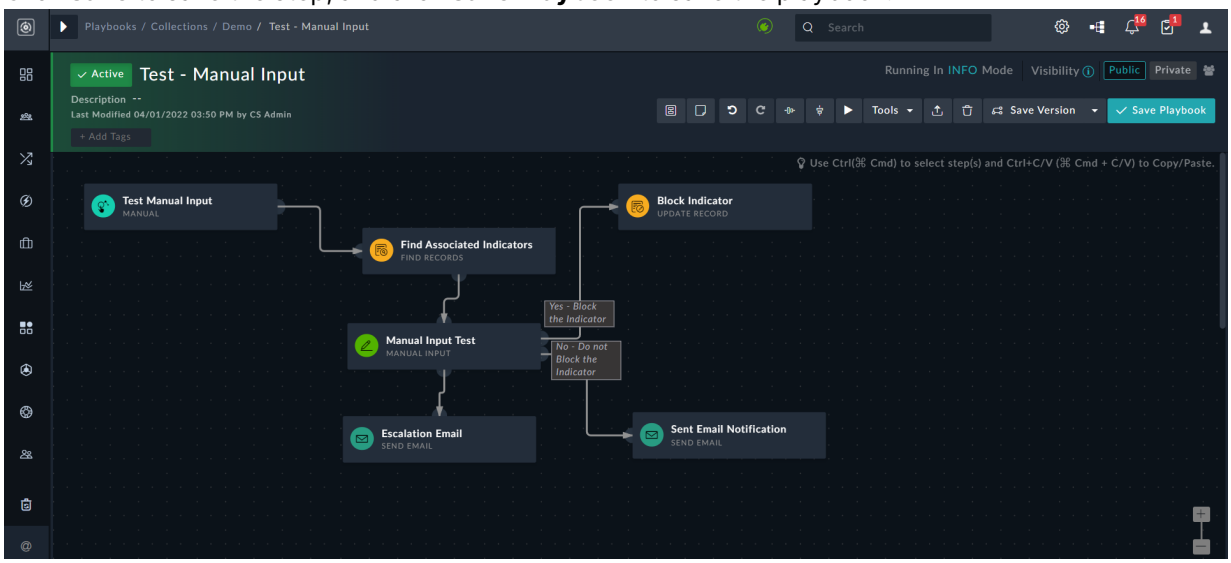
`'day_of_month': '*', 'month_of_year': '*'}`
NOTE: The days of the week are represented by integers from 0 to 6, with 0 representing Sunday and 6 representing Saturday. Therefore the `day_of_week` property accepts `*` or [`0-6`] characters. You can update this cron expression if you want to change the default schedule timing window of 1 minute, and then run the following command:
`$ sudo -u nginx /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py default_schedules`
Also, note that if the `celerybeatd` service is down then the system schedule to resume the manual input in case of an escalation step will not run. You can check the status of the celerybeatd services using the `sudo csadm services --status` command, or by viewing the System Health Dashboard.

g. (Optional) If you want to add the 'Manual Input' dialog link in the Pending Tasks Panel, then click the **Message** link that is present in the footer of the playbook step that displays the `Message` text box. In the `Message` text box, add the following inline code:
Inline Code Snippet: `<p><a data-comment-collaboration-pendingdecision='true' data-pendingdecision-id='{{vars.steps.<step_name>.wfinput_id}}'>Manual Input Link</a></p>`
Important: The format of the inline code that you require to add for adding the link to the Manual Input dialog in the Pending Tasks Panel has changed in version 6.4.0. Therefore, if you have upgraded to a 6.4.0 or later version from a version earlier than 6.4.0, you will need to change the format of the older code snippet to match that of the new code snippet.
You can type the text that you want to display as the link text in the Collaboration Panel, which by default is set to Manual Input Link within `<a data..></a>`.
For example:
Inline Code Snippet: `<p><a data-comment-collaboration-pendingdecision='true' data-pendingdecision-id='{{vars.steps.<step_name>.wfinput_id}}'>Are the Indicators Malicious?</a></p>`

Click **Ok** to save your changes.



h. Click **Save** to save the step, and click **Save Playbook** to save the playbook:



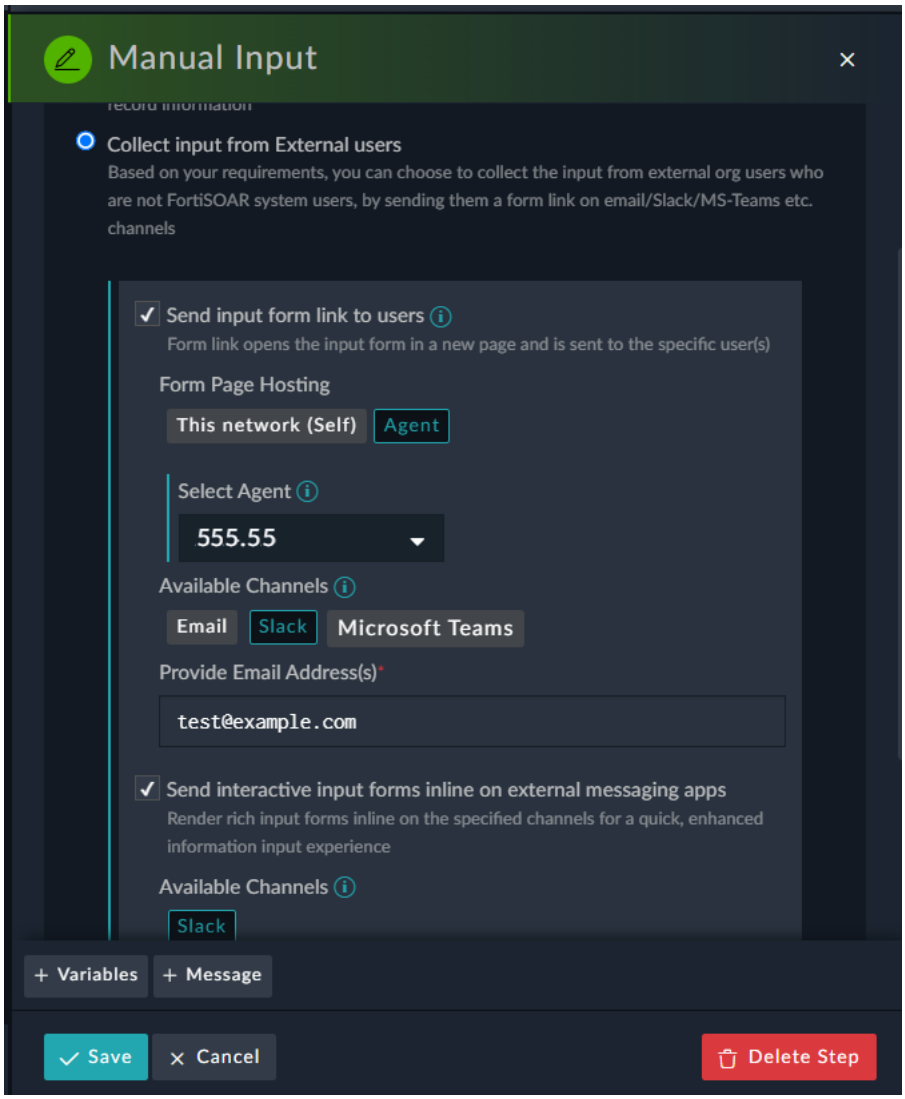
Running unauthenticated manual inputs in segmented networks using FSR agents

Prior to FortiSOAR release 7.3.0, when inputs were required from users who were outside your FortiSOAR network, an email containing a link to provide inputs was sent to the user. The URL link created for the manual input was from the originating instance, i.e., the instance where the playbook is running. Due to this, organizations were required to add their FortiSOAR instance for external IP's to their 'allowlist' of their firewall or proxy servers, which could have some implications for organization policies. To overcome these issues, FortiSOAR release 7.3.0 introduces the ability to run unauthenticated manual inputs in segmented networks using FSR agents.

To run manual inputs using FSR agents, ensure that FortiSOAR Agent role is assigned the 'Execute' permission on the 'Playbook' module.

When you select the **Send a input form link to users** option in the Manual Input step (see the Manual Input section) FortiSOAR temporarily hosts your custom input form on a page, and then renders the actual input form on the specified delivery channel, such as 'Slack' or 'Microsoft Teams', or sends a link to a page that contains the input form to the specified delivery channel, such as 'Slack', Microsoft Teams, or Email, for users to provide their responses. The input page can be made available on the same network (**This network (Self)**) or from your agent's network (**Agent**). If you select **This network (Self)** then the input form is hosted in your FortiSOAR network, and if you select **Agent** then the input form is hosted on your agent's network. It is recommended that if you require inputs from users in your FortiSOAR network, you select the **This network (Self)** option, and for users outside your FortiSOAR network, you select the **Agent** option. If **network (Self)** is selected, then the input prompt page is hosted on the same network either the actual input form is rendered on the specified delivery channel, or a link to a page that contains the input form is sent using the specified delivery channel to the external users whose email addresses are specified in the playbook. The corresponding user actions are as described in the User actions when non-FortiSOAR users are providing inputs topic.

To select the **Agent** option, click **Agent**, then from the **Select Agent** drop-down list, select the agent on whose network you want the input prompt page to be available.



The **Select Agent** drop-down list displays only those agents that are configured and reachable from the FortiSOAR node, and also have their '**Agent Input Bridge**' enabled.

If you have deleted an existing Agent and then added it again with the same name, then you will encounter a "`Server Error`" when you run the manual where you have chosen to host the manual input form link on the Agent network and chosen Email as the delivery medium. Therefore, when you are adding an Agent again (after deleting the existing one) ensure that you give the re-added Agent a new name that is distinct from the old one.

For more information on agents, see the FortiSOAR Agent Setup and Configuration chapter in the "Administration Guide.", and for steps on enabling the agent input bridge, see the FSR Agent Communication Bridge connector document.

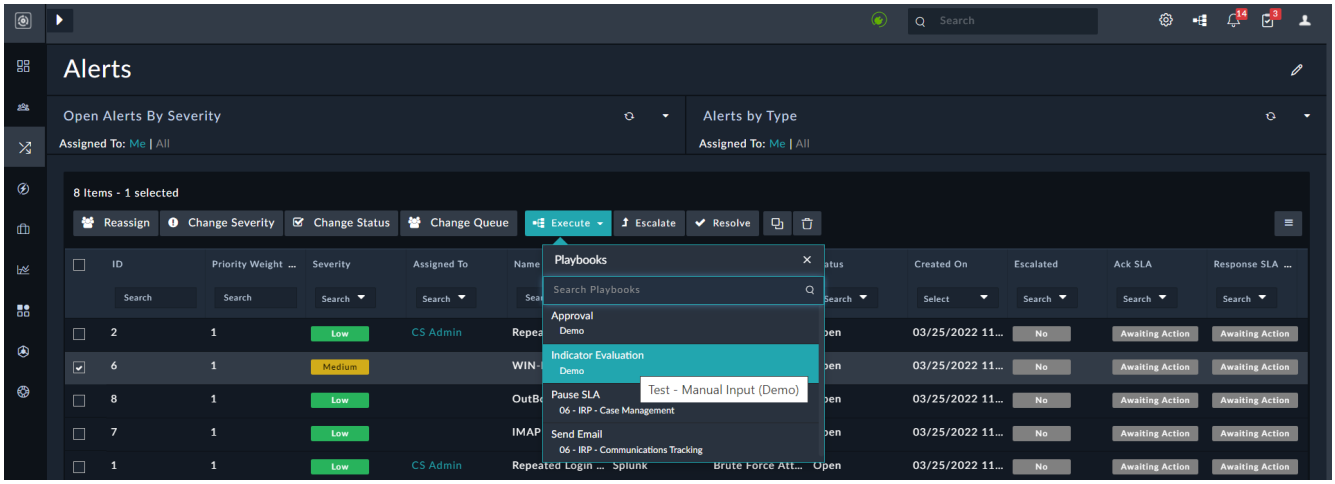
If you select an agent dynamically based on the configuration name in the playbook connector step, and the configuration name is incorrect, the connector step will be executed using the connector's default configuration on the system instead of the agent.

Once you have configured the FSR Agent Communication Bridge connector and you trigger a FortiSOAR playbook that contains a manual input, which requires inputs from users outside of your FortiSOAR network and therefore requires the custom input page to be hosted on the agent's network, the following events occur:

- If you have chosen to send then 'input form link' to users, and specified 'Email' is as the delivery mechanism, then an email containing a link of the host IP of the agent is sent to the email addresses of the external users specified in the playbook. However, if external messaging apps, such as 'Slack' or 'Microsoft Teams' is chosen as the delivery mechanism, then a link to a page that contains the input form is displayed on the external messaging app.
- When users clicks on the link, the request goes to the agent, and then the agent populates the required manual input form that requires input from the user.
- The input provided by the users is forwarded to the FortiSOAR node, which resumes the playbook based on the user input.

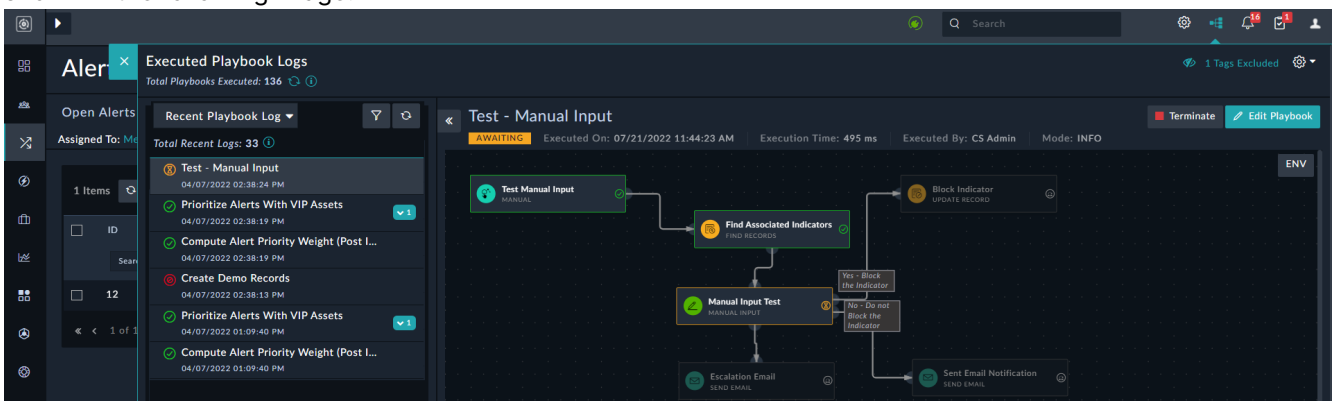
User Actions corresponding to Manual Input

The Manual input playbook gets triggered based on the type of trigger and trigger conditions defined in the playbook. For our example, we have created the 'Indicator Evaluation' playbook to be triggered on the `Alerts` module. Navigate to the **Alerts** module, and then click the record for which you want to run the Indicator Evaluation playbook, and then from the **Execute** drop-down list, select the **Indicator Evaluation** action to trigger the Indicator Evaluation playbook as shown in the following image:



Once you trigger the Indicator Evaluation playbook, FortiSOAR displays a message such as `Triggered action "Indicator Evaluation" on 1 record` and halts the further execution of the Test - Manual Input Playbook. You can open the **Executed Playbook Logs** by clicking the **Executed Playbook Logs** icon in the upper right corner of the FortiSOAR. You will see that the status of the `Test - Manual Input` Playbook is set to **Awaiting** as

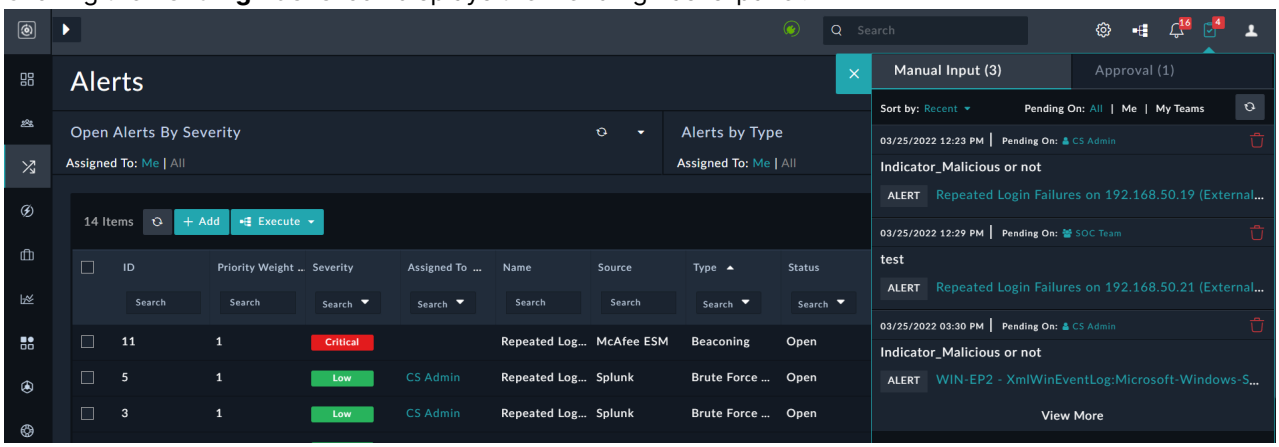
shown in the following image:



User actions when users are providing inputs using system notifications or emails

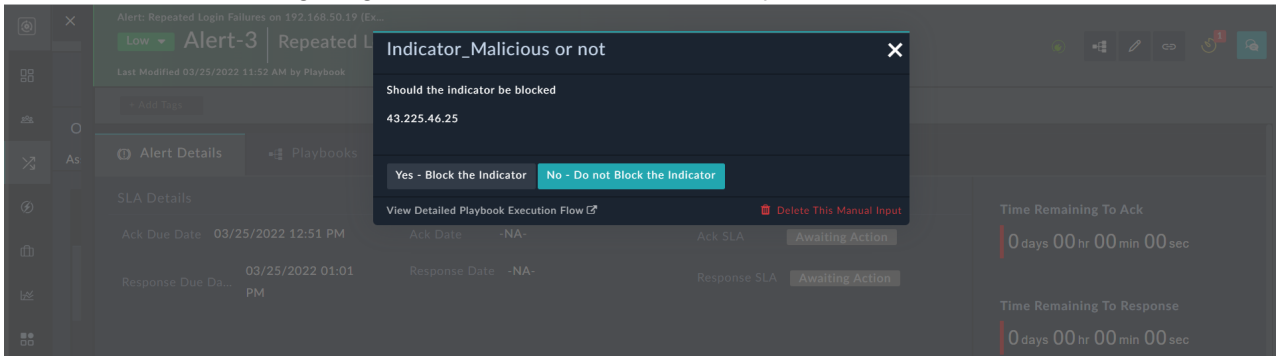
When the decision or inputs are required to be provided by a user using system notifications, then users have to do the following:

- Click the **Pending Tasks** icon that appears on the top-right corner in FortiSOAR when an action is pending. The **Pending Tasks** icon contains a number in red color that mentions the number of pending tasks, both approvals and manual inputs.
  Clicking the **Pending Tasks** icon displays the 'Pending Tasks' panel:



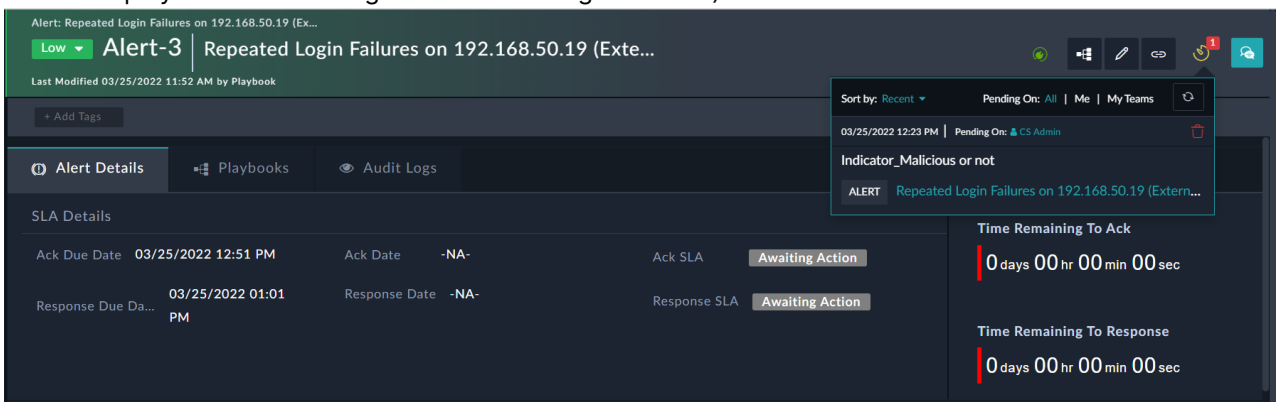
  In the **Pending Tasks**, click the item on which you want to provide input. This will open the record that is associated with the manual input as well as the `Pending Tasks` pop-up that the users can use to provide their input, which would then resume the playbook workflow. You can also click on the **Pending Decision** icon in the detailed view of the record to open the `Pending Tasks` pop-up that the users can use to provide their input, which would then resume the playbook workflow, which has been described later in this topic.
  The manual input prompt of the Pending Tasks panel contains a Pending Tasks list which displays details such as created date, the person or team the decision is assigned to, the title of the manual input step, type of record on which the action is pending, for example, the `Alert` record, as shown in the above image, and due date till when the decision should be taken are displayed. Users can also *sort* the pending tasks by **Recent**, i.e., based on its created date or on the **Due By**, which is the date by which a decision requires to be given. You can *filter* the list of pending tasks by **All**, which displays all the pending items, **Me**, which displays the pending tasks that have been assigned to the **current user**, or **My Teams**, which displays the pending tasks that have been assigned to the team(s) of the current user.
  In the **Pending Tasks List**, click the item on which you want to provide input. This opens the **Pending Tasks** pop-up that users can use to provide their input, which would then resume the playbook workflow, which

has been described later in this section.
For example, click the **Pending Tasks** icon and in the **Pending Tasks List**, click the item to provide the input, which opens the record that is associated with the manual input as well as the `Pending Tasks` pop-up (as shown in the following image) in which users can add their input:



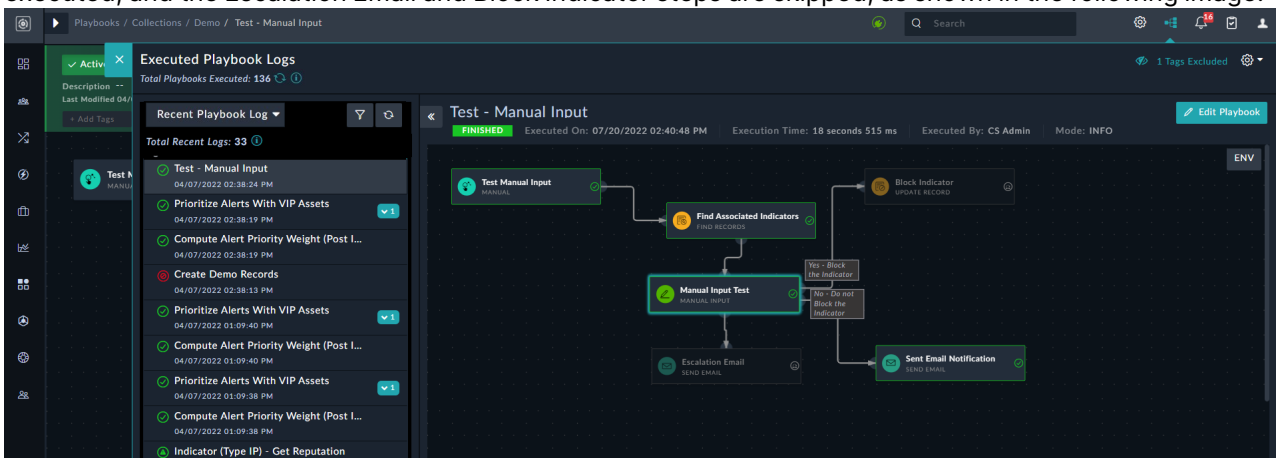
NOTE: The width of the Manual Input form is 45% of your screen size to make it easier to see the contents of the manual input, especially when it contains a lot of text. However, it stays at 600px on small displays.
If you click the **View Detailed Playbook Execution Flow** link, a new window opens that displays the execution of the playbook based on the input or decision received. Also, as displayed in the above image, since in the playbook selected **No - Do not Block the Indicator** has been specified as the *Primary* action, that option gets highlighted in the pop-up.
Users, who are assigned a role containing a minimum of '`Security Update`' permission (apart from the other required permissions) can use the **Delete This Manual Input** link to discard the manual input and remove this input from playbook workflows or queues. Use the **Delete This Manual Input** link to completely discard manual inputs without references in cases such as, removal of an executed log entry without addressing the open manual input requests, or the deletion of a record that requires manual input.
Note: Users must use the **Message** action to add a message to their playbook to add a link to the 'Manual Input' dialog as described in the Building a decision-based input prompt procedure.
For more information on the user tasks associated with the Notifications panel and Pending Tasks panel, see the Viewing Notifications and Pending Tasks topic in the "User Guide."

- You can also see the **Pending Decisions** icon in the detail view of the alert on which the playbook is triggered and depending on the ownership you have defined in the playbook. For example, if you have provided the ownership of **Analyst working on the record**, and if you are not assigned to that record, then you will not see the Pending Decisions button. The details of the Pending Decisions list are the same as the details displayed in the Pending Tasks list at the global level) :

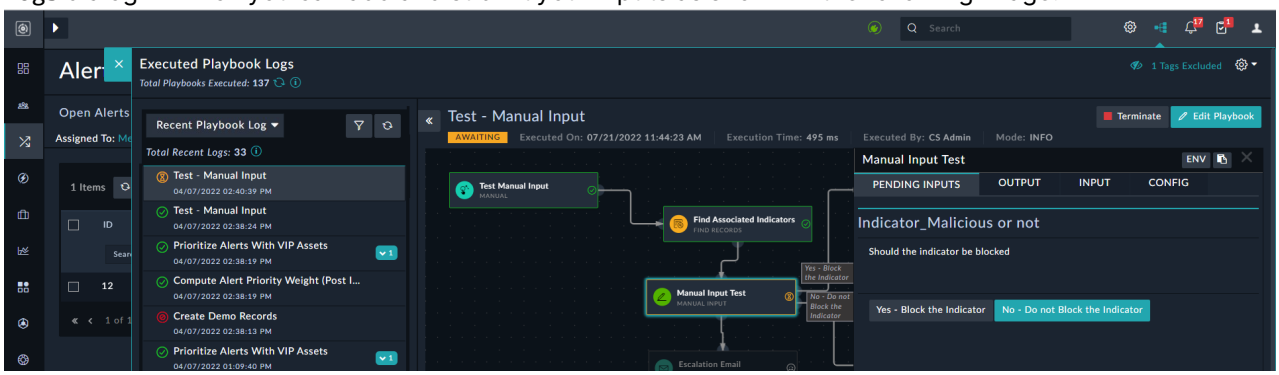


Clicking the item in the **Pending Decisions** list displays the `Pending Tasks` decision box as explained earlier.

- If, for example, the user selects the **No - Do not Block the Indicator** option in the `Pending Tasks` decision box, then FortiSOAR displays a message such as `Awaiting playbook resumed successfully`, and then, based on the user's decision, FortiSOAR continues the execution of the playbook. For our example, the 'Block Indicators' step will be run, which will block the indicator. Users can open the **Executed Playbook Logs** by clicking the **Executed Playbook Logs** icon in the upper right corner of the FortiSOAR, and there they will see that the status of the "Test - Manual Input Playbook" is set to **Finished**, the Send Email step is executed, and the Escalation Email and Block Indicator steps are skipped, as shown in the following image:



An analyst or user on whom the action is awaiting can also provide the input from the **Executed Playbook Logs**. Click the **Executed Playbook Logs** icon in the upper right corner of the FortiSOAR to open the Executed Playbook Logs and click the playbook whose status is **Awaiting**. Clicking the awaiting playbook opens the **Test - Manual Input** playbook > **Pending Inputs** tab on the right side of the `Executed Playbook Logs` dialog in which you can add and submit your inputs as shown in the following image:



System users can also provide their decision or inputs using email and it works the same way as described in the following section. Note that, you can customize the text of the email body by editing the delivery rules on the Notifications page. In this case, you have to edit the "Notify on Pending Internal Manual Input Notification" rule. For more information, see the 'Notifications' topic in the *System Configuration and Management* chapter of the "Administration Guide."

User actions when non-FortiSOAR users are providing inputs

Using an email

If the decisions or inputs are required to be provided by a non-FortiSOAR user, using 'Email' as the delivery channel, then users have to do the following:

- Once the playbook is triggered and the playbook is set to **Awaiting**, an email containing a link to a page that contains the input form gets sent to the email addresses mentioned in the playbook. The email body contains text such as, "`A FortiSOAR Playbook is requesting your input...`" and a link such as, "`Open input form`".
  You can customize the text of the email body by editing the delivery rules on the Notifications page. In this case, you have to edit the "Notify on Pending External Manual Input Notification" rule. Also, if you have upgraded your system to release 7.2.0 or later, and you have used customized email templates for external manual inputs, then you must update the "Notify on Pending External Manual Input Notification" rule. For more information, see the 'Notifications' topic in the *System Configuration and Management* chapter of the "Administration Guide."
  Note: Some examples of Jinja expressions that can be used while creating or adding notifications are included in the `Usage examples of Jinja Expressions in Notifications` section in the 'Notifications' topic in the *System Configuration and Management* chapter of the "Administration Guide."
  By default, email notifications are sent using SMTP. However, you can choose to send email notifications using a different email server, such as Exchange. To do this, you can either update the 'Email Notification' channel or create a new custom channel and use this channel in the "Notify on Pending External Manual Input Notification" rule. For more information, see the 'Notifications' topic in the *System Configuration and Management* chapter of the "Administration Guide."
- Clicking the link opens the browser and displays a page in which users are required to provide their inputs. The contents of this page depend on the title, description, and design of the 'Custom Input Form' that you have added in the playbook, along with the configured responses, such as a button for submitting the response, or buttons for acceptance or rejection of the decision. Users should provide the necessary inputs and then click the appropriate button to submit the form. Once the form is submitted, it cannot be re-opened and its contents cannot be changed.
  In the case of our example, the user will see "Should the indicator be blocked?" followed by the indicator value and then two buttons: "Yes - Block the indicator", or "No - Do not block the indicator" which displays a page in which they are required to provide their inputs. Users should evaluate the indicator and then choose Yes or No to submit their response.

Once the user provides the required inputs and submits their response, the playbook continues its execution as per the defined workflow.

Using an external app, such as Slack or Microsoft Teams

If the decisions or inputs are required to be provided by a non-FortiSOAR user using Slack or Microsoft Teams, then users have to do the following:

- Once the playbook is triggered and the playbook is set to **Awaiting**, either the actual input form or a link to a page that contains the input form gets sent to the configured Slack or Microsoft Teams channel based on the email addresses are specified in the playbook. For information about Slack or Microsoft Teams configuration and integration with FortiSOAR, see the documentation on the Content Hub Portal.
- Users, on receiving either the actual input form or a link to a page that contains the input form on the configured Slack or Microsoft Teams channel can do the following:
  - If a link to a page that contains the input form is received on the Slack or Microsoft Teams channel, then users can click the link to open a browser, which displays a page in which they are required to provide their inputs.
    The contents of this page depend on the title, description, and design of the 'Custom Input Form' that you have added in the playbook, along with the configured responses, such as a button for submitting the response, or buttons for acceptance or rejection of the decision. Users should provide the necessary inputs and then click the appropriate button to submit the form. Once the form is submitted, it cannot be re-opened and its contents cannot be changed.
    In the case of our example, the user will see "Should the indicator be blocked?" followed by the

indicator value and then two buttons: "Yes - Block the indicator", or "No - Do not block the indicator" which displays a page in which they are required to provide their inputs. Users should evaluate the indicator and then choose Yes or No to submit their response.

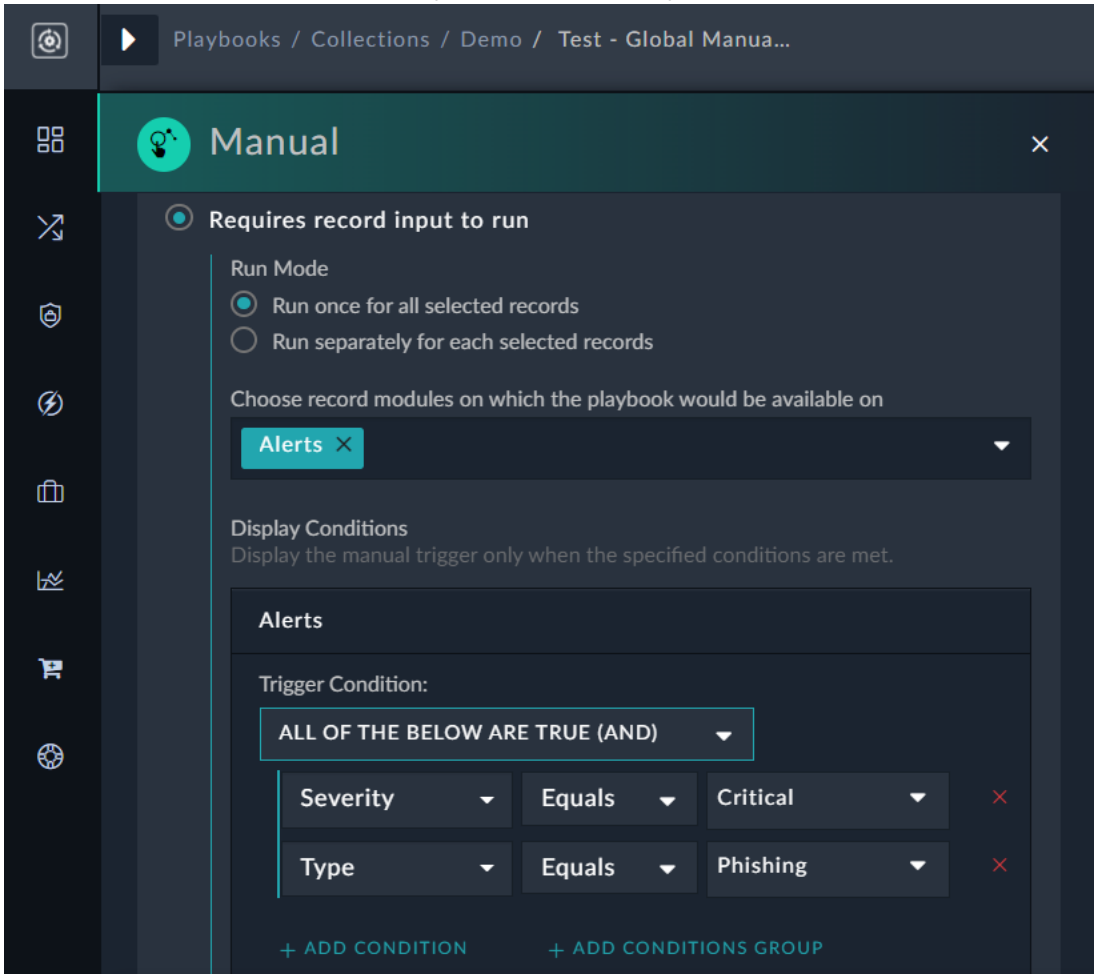
- If the actual 'Custom Input Form' is rendered on the Slack or Microsoft Teams channel, then users will see the 'Input Form' containing fields that have been defined in the playbook along with the configured responses, such as a button for submitting the response, or buttons for acceptance or rejection of the decision. Users should provide the necessary inputs on Slack itself and then submit the form. Once the form is submitted, it cannot be re-opened and its contents cannot be changed.

Once the user provides the required inputs and submits their response, the playbook continues its execution as per the defined workflow.

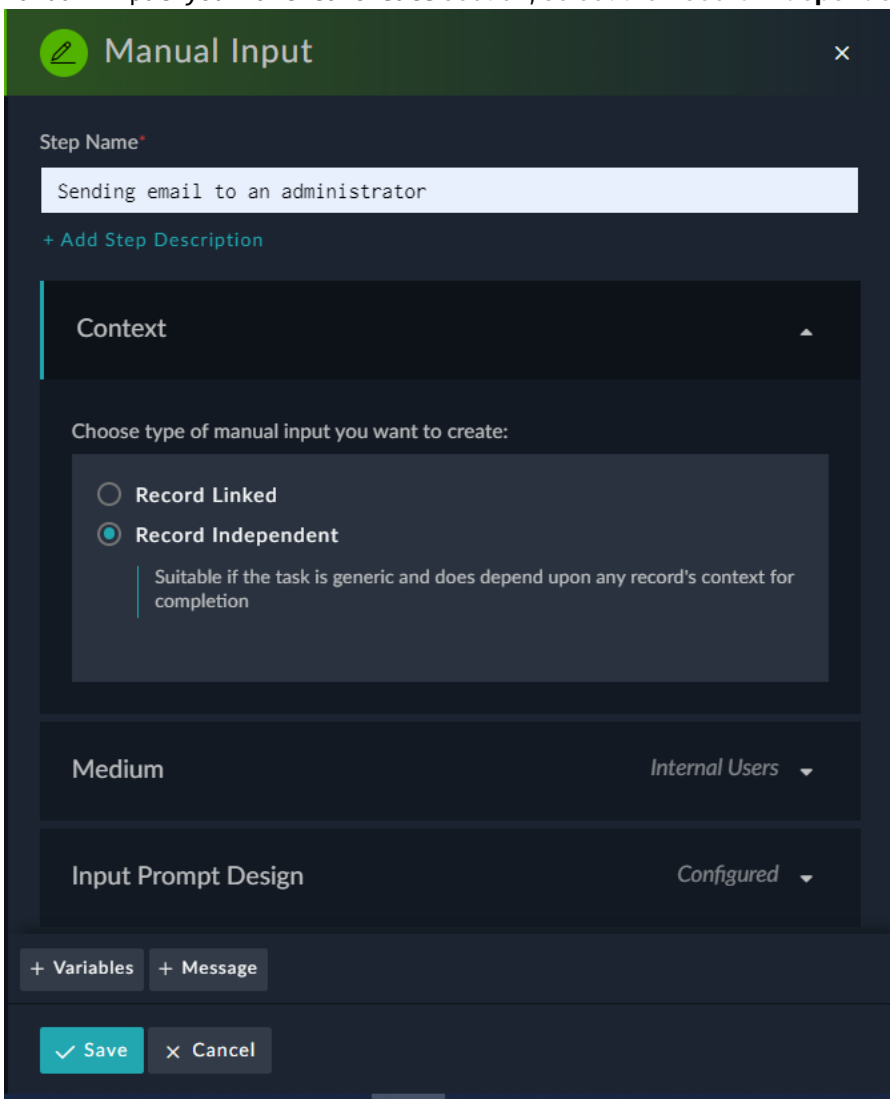
Global Manual Input

From release 7.2.0 onwards, you can create a manual input that is independent of records, and which could be acted on by users anywhere in FortiSOAR. Global Manual inputs are suitable if the tasks to be performed are generic, and which do not require the context of a record for its completion.

An example, of a global manual input, could be a requirement of sending an email to an administrator in case of a "Critical Alert of type 'Phishing'". For this example, you can create a manual trigger playbook named "Send Email to Administrator", and then in which you can add visibility conditions as follows:



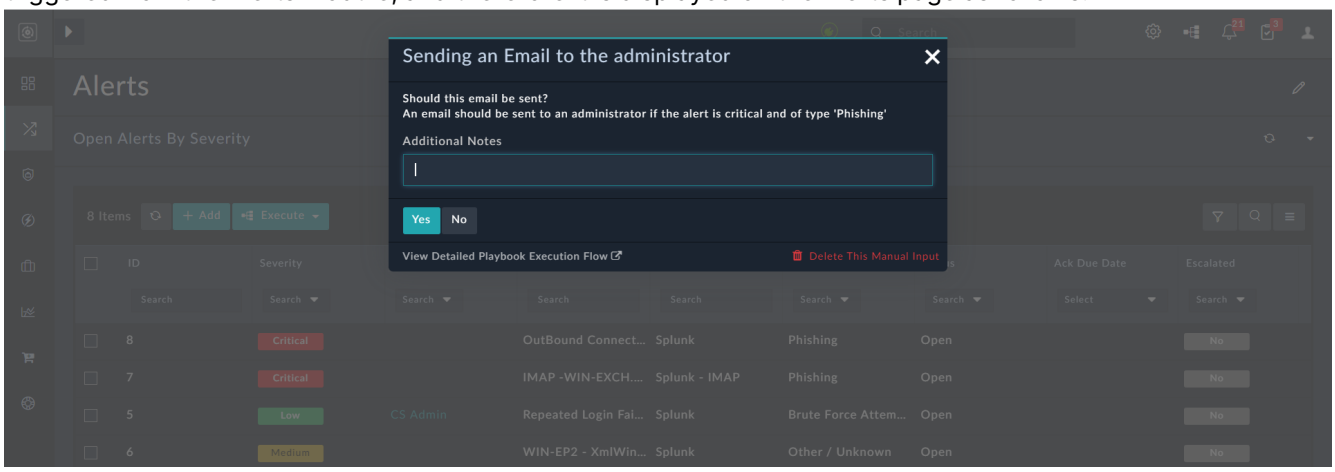
Next, add the Manual Input step to create a Global Manual input. In the **Context** menu from the `Choose type of manual input you want to create` section, select the **Record Independent** option:



You can then configure the manual input as per your requirements. For more information on setting up manual inputs, see the Manual Input topic.

Once you have created and saved the playbook, navigate to the Alerts page, and select a Critical alert whose type is Phishing. Click Execute and select the Send Email to Administrator playbook, FortiSOAR will display the Global Manual pop-up on the UI from the module on which it is triggered. In our example, this manual input was

triggered from the Alerts module, and therefore it is displayed on the Alerts page as follows:



If users are not on the Alerts module when this manual input is triggered, then users have to use the Notifications panel or the Pending Tasks panel to view the Global manual inputs. For more information on the Notification and Pending Tasks panel, see the User Actions corresponding to Manual Input topic.
Users can add additional notes in the pop up dialog and then click the **Yes** or **No** to resume the designed playbook workflow.

Considerations:

- **Customization**: Custom input forms can be created to collect decisions or data from users at specific points in the playbook. These forms can be simple confirmation (for quick decisions) or detailed (requiring additional data).
- **Context Definition**: The input form can either operate independently of records or be linked to records. For example, in a record-based context, user input may enrich indicators associated with an alert.
- **Visibility Conditions**: Conditions can be defined to control when certain fields appear or are hidden, based on previous user inputs or other variables. For instance,fields may only be displayed if specific criteria are met, streamlining the user experience.
- **Integration with External Channels**:Manual Input supports integration with external messaging platforms like Slack and Microsoft Teams, allowing input prompts to be triggered directly within these tools. However, these integrations require the installation and configuration of specific solution packs: FortiSOAR For Slack and FortiSOAR For Microsoft Teams.
- **Using Email Templates and Dynamic Inputs**: Email content can be dynamically included in the input form to assist analysts in determining necessary actions based on incoming email content. Dynamic Lists are also supported, allowing options in the input prompt to be defined dynamically using JSON or comma-separated values, making the input form adaptable to real-time changes in the playbook's context.

Execute Steps

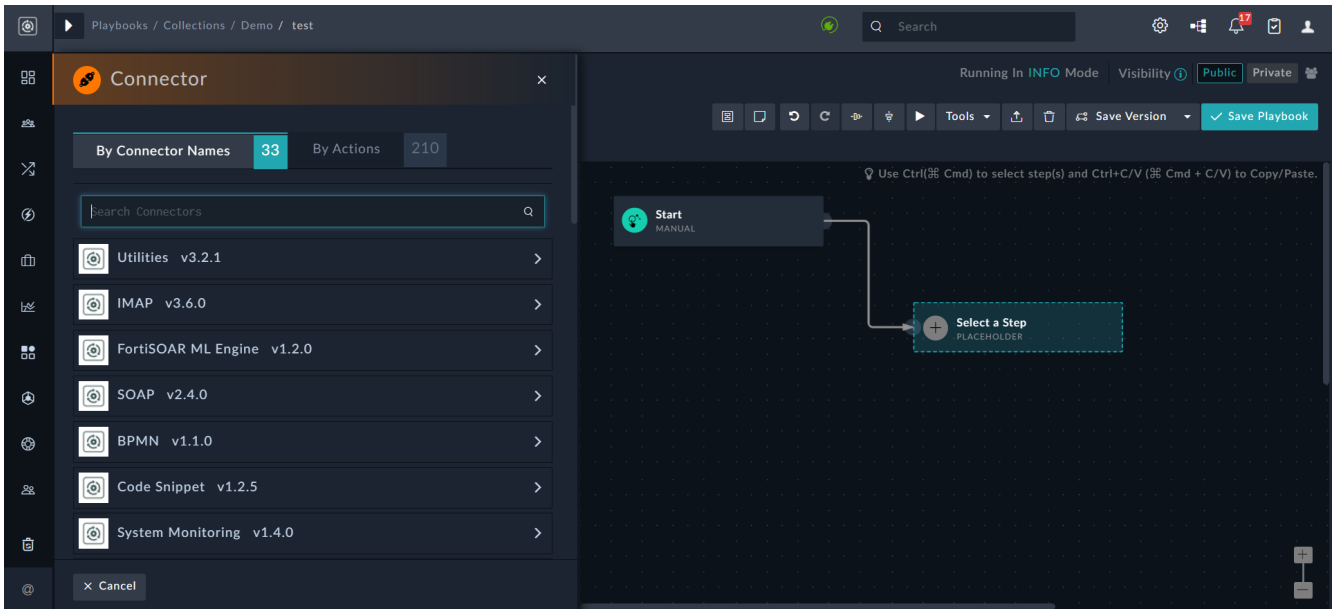
Used to perform actions such as executing connector functions or built-in utilities.

Connector

Use the **Connector** step to add connectors, including FortiSOAR Built-in connectors, to your playbook. Third-Party Connectors, such as connectors for Elastic, VirusTotal, or Splunk, can retrieve data from custom sources and perform automated operations. FortiSOAR Built-in connectors, such as the Database connector, the IMAP connector, and the SMTP, are all pre-installed connectors or built-ins that you can use within FortiSOAR playbook and perform automated operations. For more information see the FortiSOAR Connectors page.

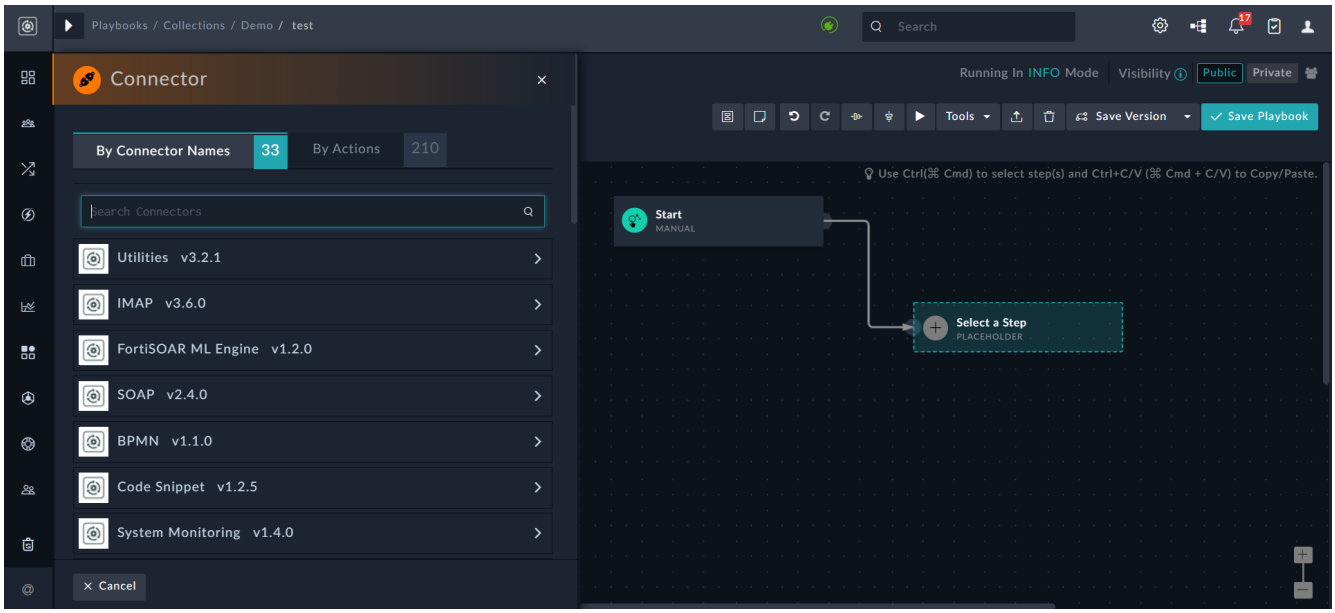
Use the **By Connector Name** tab to first choose a specific connector and then choose the operation that you want that connector to perform or use the **By Actions** tab to first choose the action (annotation) that you want to perform and then choose the connector that you want to use to perform the selected action.

Once you click the **Connectors** step, the `Connectors` step page is displayed that contains the connectors (**By Connector Names** tab) that are configured in your system and the automated actions that you can perform (**By Actions** tab).



By Connector Names tab

After selecting the **Connector** step in the playbook designer, the **By Connector Names** tab is displayed. The **By Connector Names** tab displays all the connectors that are configured in your system. Use this tab if you want to use a particular connector to perform a particular action.

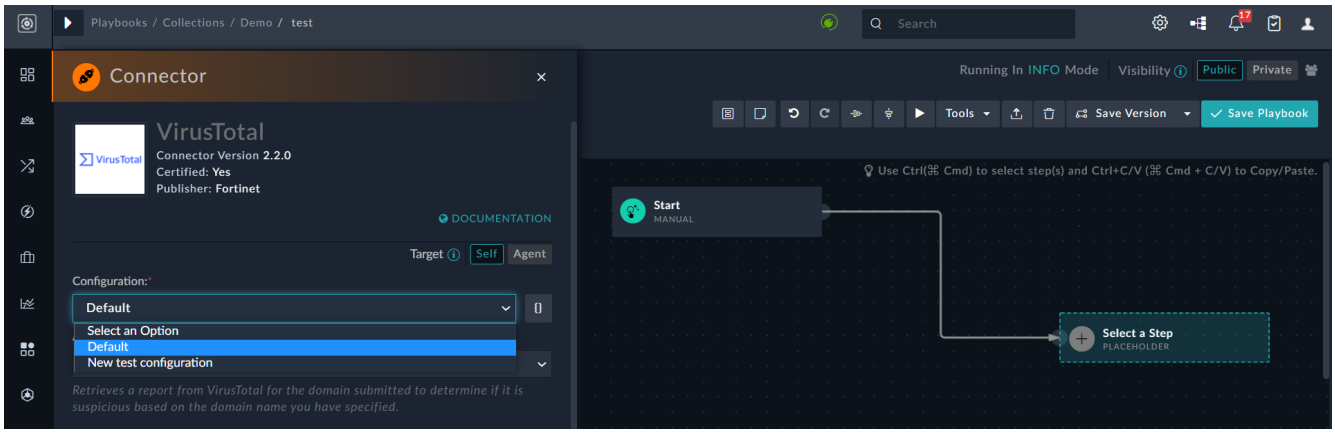


Use the `Search Connectors` section to search for connectors by name.

Click the connector that you want to include in your playbook, for example, **VirusTotal v1.0.1** and then type the **Step Name**. You can also specify whether you want to run the action on the current FortiSOAR node or remotely on the agent node by clicking the **Self** or **Agent** buttons besides `Target`. By default, **Self** is selected, which means that the action will run on the current FortiSOAR node, then you must select the configuration by clicking the **Configurations** drop-down list using which you want to run the action since the FortiSOAR node can have multiple configurations. Configurations are based on the configuration names that you specify when you are configuring the connector (see notes below). If you click **Agent**, then you can select the agent on which you want to run the action and you must also select the configuration using which you want to run the action since agents can have multiple configurations. For more information on agents and how to run remote actions using agents, see the FortiSOAR Agent Setup and Configuration chapter in the "Administration Guide." You can also specify the connector configuration by clicking the **{}** icon and either typing the connector configuration name or specifying a Jinja variable that contains the connector configuration name. If you have only one configuration for the connector or have specified a default configuration, then that configuration automatically gets selected.

Users can see only those connector configurations to which they have access. For example, if a VirusTotal connector is configured with configuration name as 'Demo1' and with visibility set to 'Private' with assignment given to 'Team 1' (see Ownership, Roles, and Permissions for details), then the 'Demo 1' configuration is not visible to users belonging to teams other than 'Team 1', though they can execute playbooks created by 'Team 1' users.

Next, from the **Action** drop-down list, select the action that you want the connector to perform and then in the `Inputs` section, specify the inputs required. Click **Save** to add the connector as a playbook step.

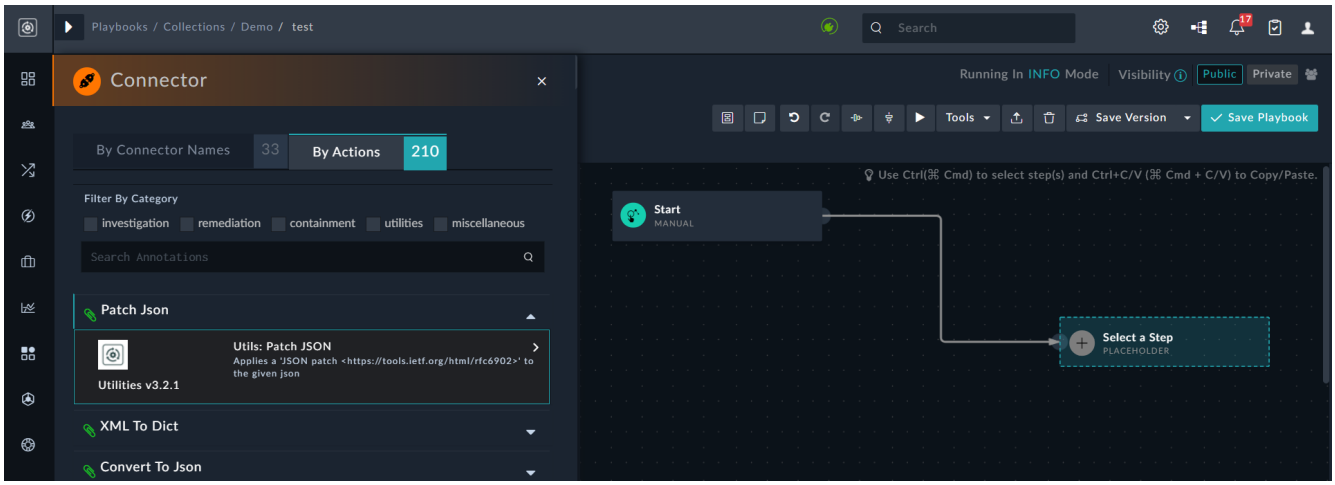


Notes:

- You can install different versions of a connector, and while adding a connector operation, you can specify a specific version of a connector within a playbook. For example, you can have VirusTotal connector versions 1.0.0 and 1.0.1. The version of the connector must be in the **x.y.z** format, for example, **1.0.0**. Version must consist of valid integers, for example, "1.15.125" is a valid version.
  In case you have installed multiple connectors, and if the version of the connector specified in the playbook is not found, then the playbook by default uses the latest version. FortiSOAR checks for the latest version of the of the connector in the format "major version.minor version.patch version". For example, version 2.0.1 is a later version than 1.0.1.
- Upgraded versions of your connector are displayed on the Connectors page and you can upgrade the version of your connector. The upgrade process replaces your existing connector version with the upgraded version. For more information, see the *Introduction to connectors* chapter in the "Connectors Guide."
- You can install different versions of a connector, enabling you to reference a specific version of a connector from a playbook. If you want to replace all previous versions of the connector, ensure that you click the **Delete all existing versions** checkbox while importing the new version of the connector. If you do not click the **Delete all existing versions** checkbox, then a new version of the connector is added. You must ensure that your playbooks reference a correct and existing version of the connector.
- You can add multiple configurations for your connector if you have more than one instance of your third-party server in your environment. You must, therefore, add a unique `Name` for each configuration.
  If you have previous versions of a connector and you are configuring a newer version of that connector with the same configuration parameters, then FortiSOAR fetches the configuration and input parameters of the latest available version of that connector. For example, if you have 1.0.0 and 1.0.1 versions of the VirusTotal connector and you are configuring the 1.0.1 version of the VirusTotal connector, then while configuring the 1.0.1 version, FortiSOAR will fetch the configuration and input parameters from the 1.0.0 version of the VirusTotal connector. You can review the configuration and input parameters, and then decide to change them or leave them unchanged.
- You can check the **Mark As Default Configuration** option to make the selected configuration, the default configuration of this connector, on the particular FortiSOAR instance. This connector will point to this configuration by default.
- The `password` type fields include encryption and decryption. All configuration fields of type `password` are encrypted before they are saved in the database.

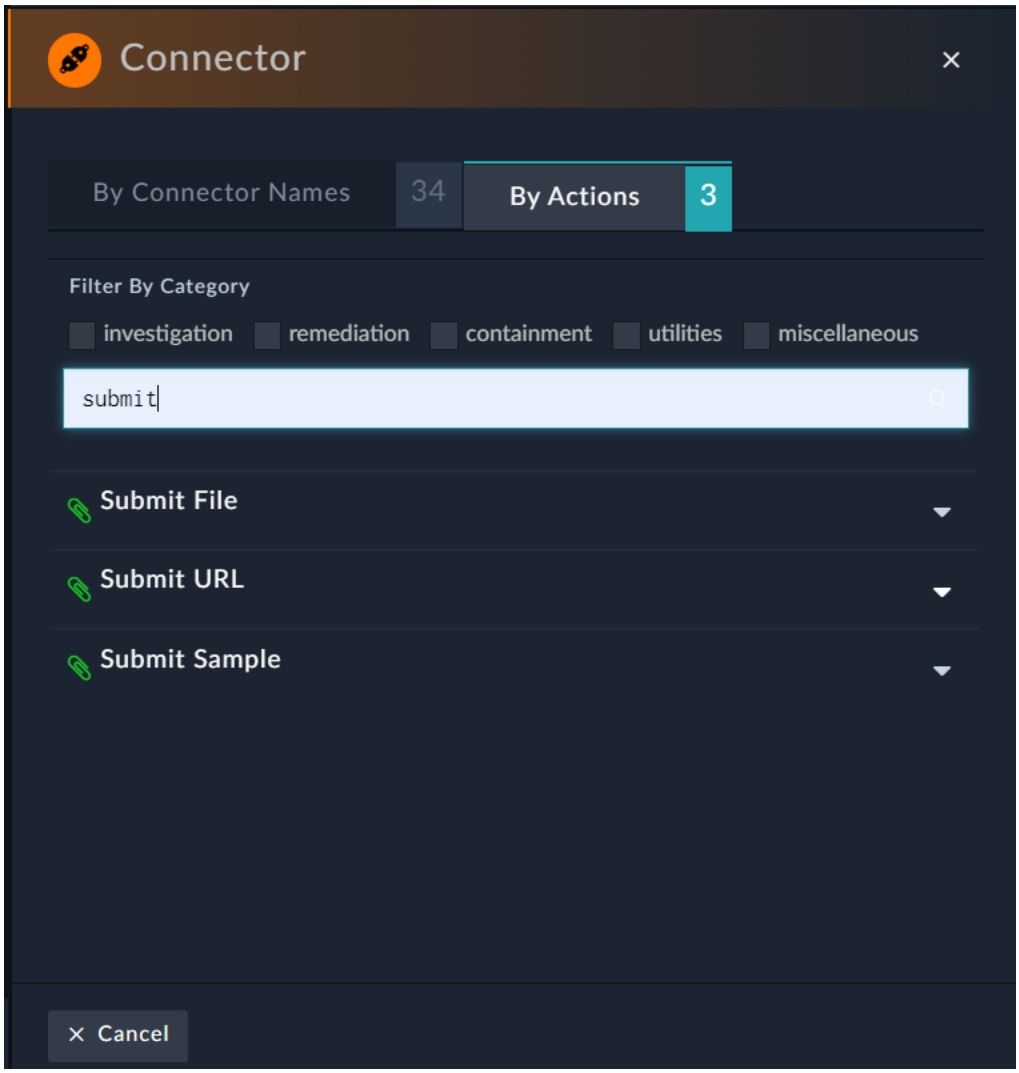
By Actions tab

After selecting the **Connector** step in the playbook designer, if you want to see the available connectors configured in your system for a particular action, then click the **By Actions** tab. Click the **down** arrow to view which connector is providing that action and the description of the action. The **By Action** tab displays the connectors grouped by actions.

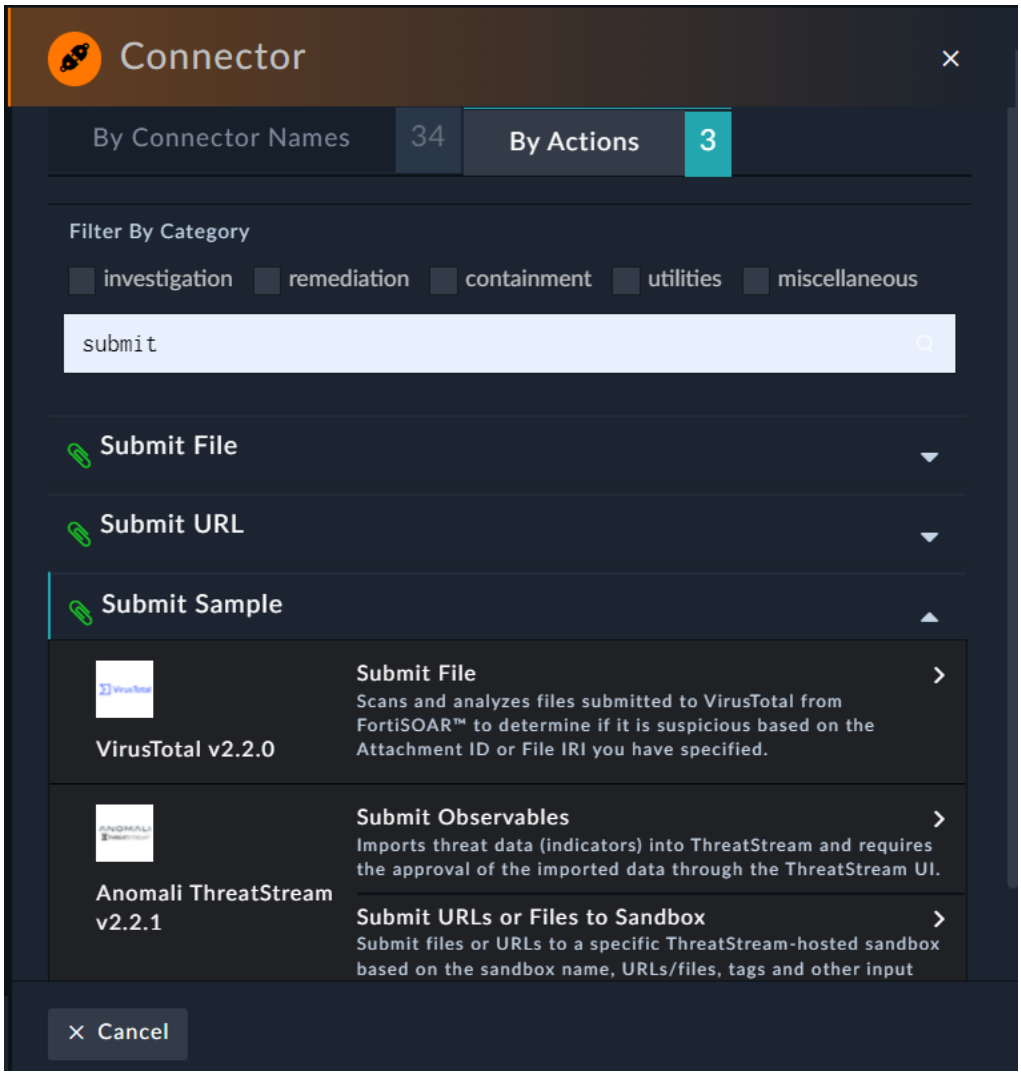


Use the `Filter By Category` section to filter the actions on the basis of the type of operation they will perform. The types of operations are currently categorized into **Investigation**, **Remediation**, **Containment**, **Utilities**, and **Miscellaneous** categories.

To search for a specific action that you want to perform, type the search keyword in the **Search Annotations** search box.



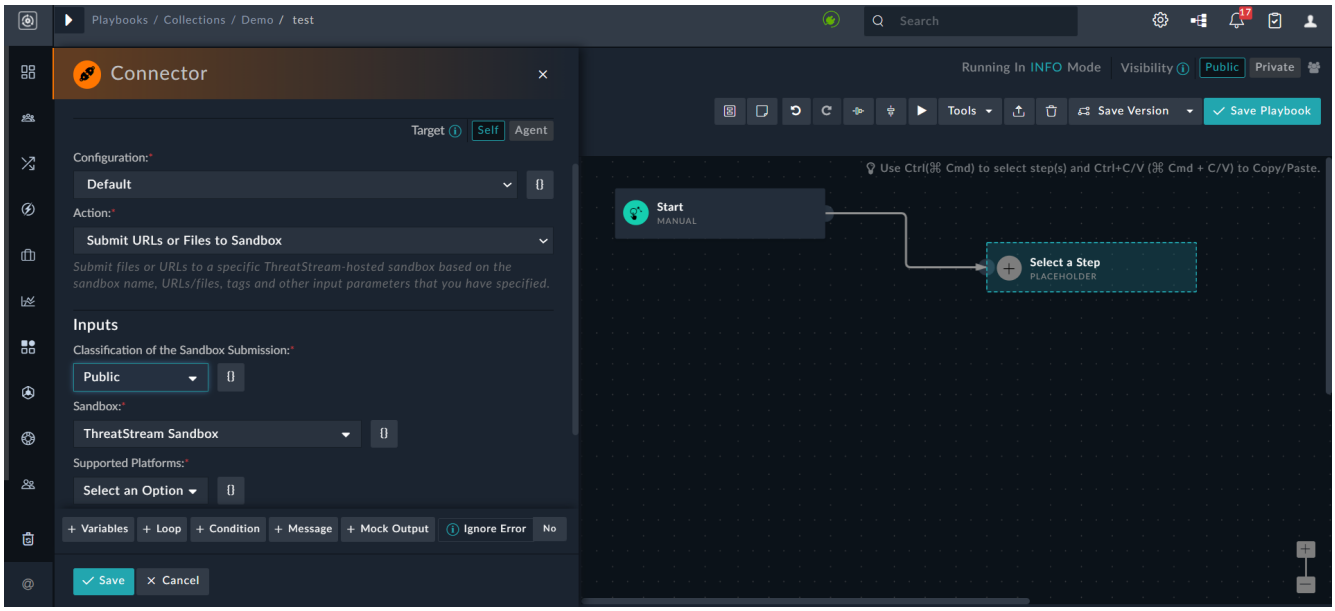
Click the name of the action that you want to perform. For example, if you want to submit a sample for analysis to a website or a sandbox click the **Submit Sample** action. Once you select the action, then a list of configured connectors that can perform that operation is displayed as shown in the following image:



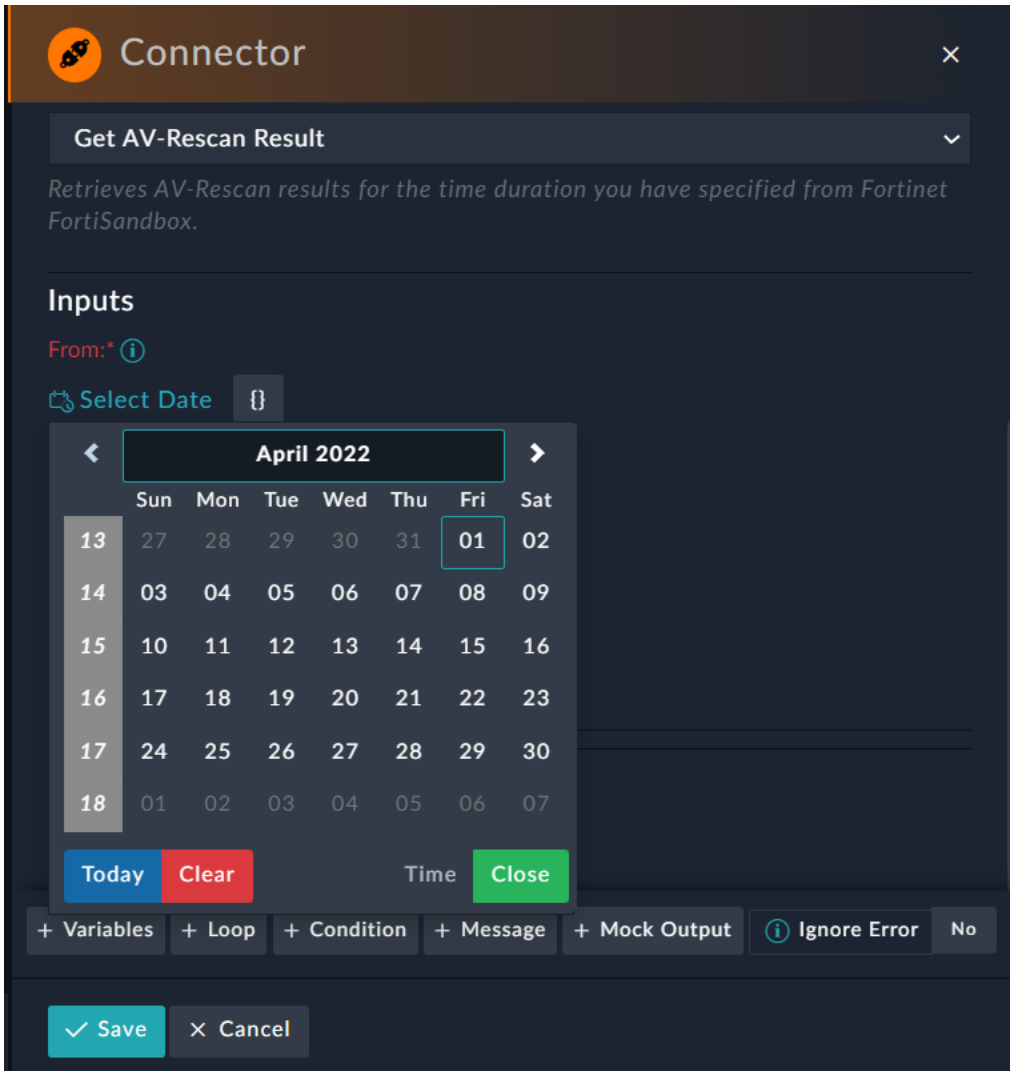
An annotation can have multiple connectors configured to perform that action, and if more than one connector can perform the same action, then a list of connectors will be displayed when you click the name of the action. As in our example, we want to submit a sample for analysis click the **Get Sample** action, and you will see that multiple connectors, such as `VirusTotal` and `Anomali ThreatStream` are tagged with this annotation.

Select the connector and the exact operation that you want to perform and then type the **Step Name**. Next, in the `Inputs` section specify the necessary input parameters to run the operation. Click **Save** to add the connector as a playbook step.

For example, to submit a sample for analysis click the **Submit Sample** action and you will see the connectors associated with this action. Select the connector, for example, the `Anomali Threatstream` connector, and you will see multiple functions, such as **Submit Observables** and **Submit URLs or Files to Sandbox**, associated with the desired action as shown in the above image. Click the exact operation that you want to perform, for example, if you want to submit files or URLs to a specific ThreatStream-hosted sandbox, then click **Submit URLs or Files to Sandbox**. Next, type the **Step Name**, and in the `Inputs` section, enter the input parameters, such as the sandbox name and sample type that you want to submit for analysis to Threatstream, and then click **Save** to add the connector as a playbook step.



In case of connector actions that have the `Datetime` field whose sub type is set as `Date`, you can use the Date picker to choose the date (such fields do not have the time picker. For `Datetime` field whose sub type is set as `Date/Time`, you can use the Date and Time picker to choose the date and time as shown in the following image:



You can also add custom expressions in the jinja format in the `Datetime` field. Click the {} icon to enter Jinja for this field. Click the icon to toggle back to the original `Datetime` field.

Key Points:

- **Multiple Versions of a Connector**: FortiSOAR supports multiple versions of a connector. When adding a connector action to a playbook, the specific version to use can be specified (e.g., 1.0.0, 2.1.5). If a specific version is not selected, FortiSOAR will default to the latest available version.
- **Connector Configuration**: Before use, connectors must be configured with necessary details such as API keys, IP addresses, and specific permissions or profiles on the external system. The connector's *Health Check* status should be marked as 'Available' before it is used.
- **Multiple Configurations per Connector**: If multiple instances of a third-party service exist (e.g., separate API keys or environments for VirusTotal), multiple configurations for the connector can be set up in FortiSOAR.
- **Visibility Settings**: Ensure that visibility settings are configured such that only authorized teams can access and use the connector configurations.

- **Verify Connector Installation**: Playbooks can be imported without the referenced connector installed, which creates a generic step. However, for the connector step to function correctly, the corresponding connector must be installed on the FortiSOAR instance.
- **Regular Connector Updates**: Although updates are not always explicitly required, regularly updating connectors ensures they remain functional and performant.
- **Password management**: Configuration fields related to passwords are encrypted before saving in the database. These fields should be securely handled to maintain confidentiality.
- **Connector and Action Associations**: An action (annotation) may be supported by multiple connectors (e.g., the "Submit Sample" action can be performed by both VirusTotal and Anomali ThreatStream). The most appropriate connector should be selected based on operational needs.
- **Scheduling for Data Ingestion Connectors**: Connectors designed for data ingestion (like the FortiSIEM connector) can be scheduled to run automatically at configured intervals to pull data into FortiSOAR. For details, see the Data Ingestion chapter in the "Connectors Guide."

Utilities

Use the **Utilities** step to run various utility functions and scripts that come built-in with FortiSOAR. For details, see the Utilities Connector document.

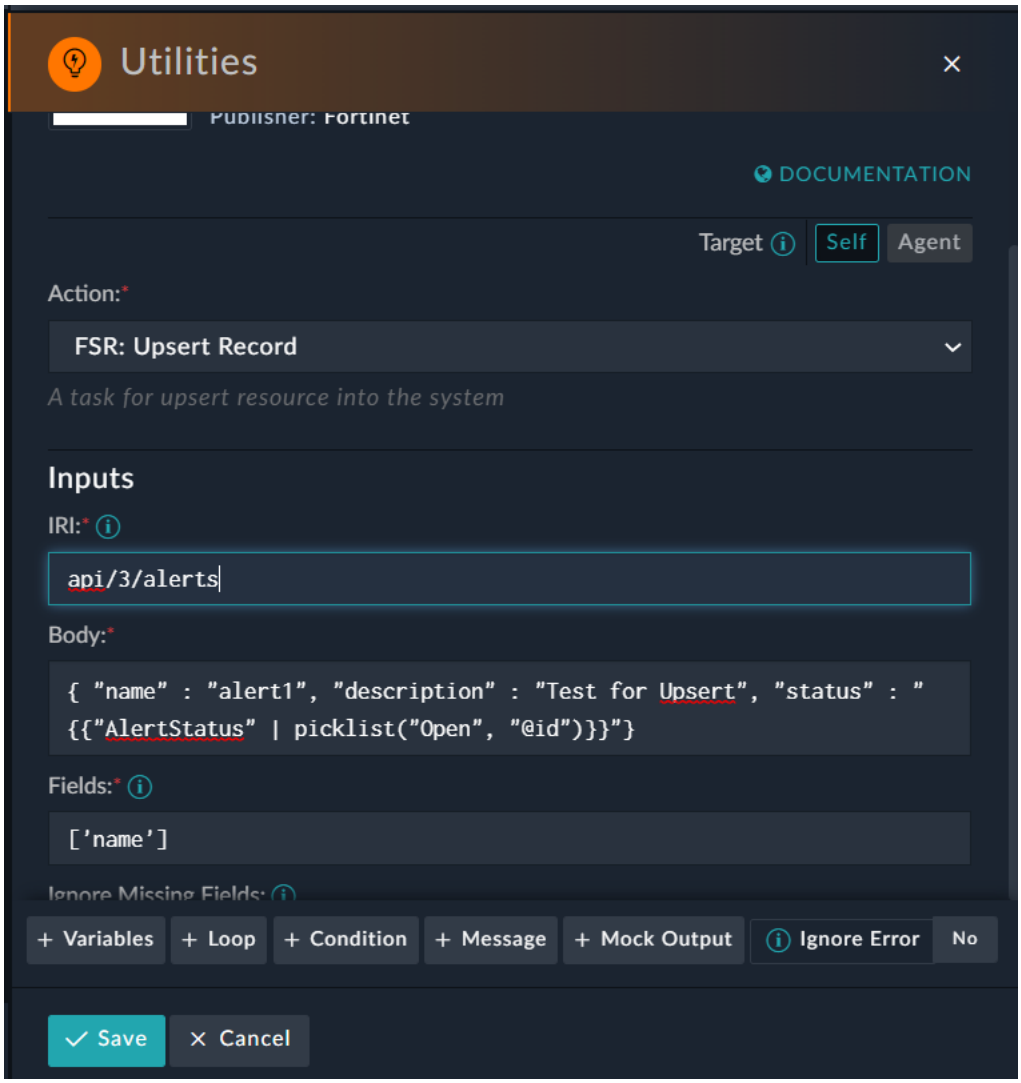
Utility functions include functions such as, the **Utils: Make REST API Call** action to make a RESTful API call to any valid URL endpoint, the **FSR: Create Record** action to insert a new record in FortiSOAR, the **Email: Extracts email's metadata from email file** action to extract the metadata of the specified eml/msg file that is present in the `/tmp` directory on your system, and the **File: Zip** action to zip and password protect a file.

Example of using the FSR: Upsert Record option in the Utilities step

The **FSR: Upsert Record** step either updates an existing record, if any record matches the unique list of fields you have specified, in the database, or inserts a new record in the database based on the parameters you have specified.

Upsert behavior for uniqueness will not work for fields that are marked as encrypted.

In the Playbook Designer, click the **Utilities** step and add the step name in the **Step Name** field. From the **Action** drop-down list, select **FSR: Upsert Record**. In the `IRI` field add the name of the module in which you want to upsert data in the format `api/3/alerts`. In the `Body` field, add the fields that you want to add or update in the database in the *dynamic values* (Jinja variables) format. For example, `{ "name" : "alert1", "description" : "Test for Upsert", "status" : "{{"AlertStatus" | picklist("Open", "@id")}}"}`. In the `Fields` field, add the list of fields to check for uniqueness. For example, if you want to check for records in the database based on the `Name` of the record in the database, add `['name']` in the `Fields` field. If you want to search the database based on multiple items, you can add more than one item in the `Fields` field, for example, `['name', 'status']`. The `Ignore Missing Fields` field is used to determine whether or not to raise an exception if you specify a field in the `Fields` field that is not in the record. The `Ignore Missing Fields` defaults to `False`, which means that an exception will be raised if you specify a field in the `Fields` field that is not in the record. Click **Save** to save the step.



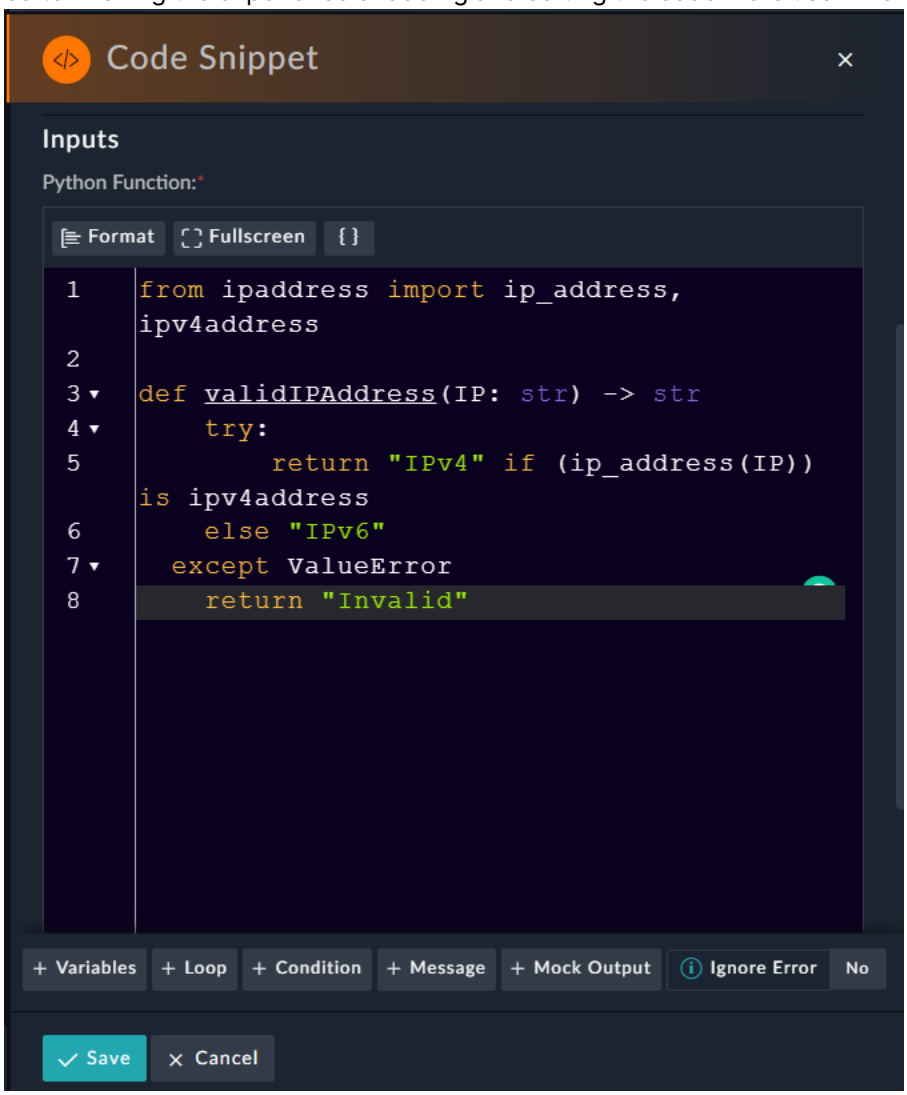
Once the step is run, the database record will either be updated with the parameter you have specified, if any record matches the list of fields you have specified in the `Fields` field, or a new record will be inserted in the database based on the parameters you have specified.

Code Snippet

Use the `Code Snippet` step to add and run custom python scripts within a playbook. For details, see the Code Snippet Connector document.

Select the `Code Snippet` step and in the **Execute Python Code** action, enter the python function that you want to run as part of the playbook, and click **Save** to save the step.

The **Python Function** field's interface that is part of the **Execute Python Code** action is enhanced to a code text editor making the experience of adding and editing the code more user-friendly:

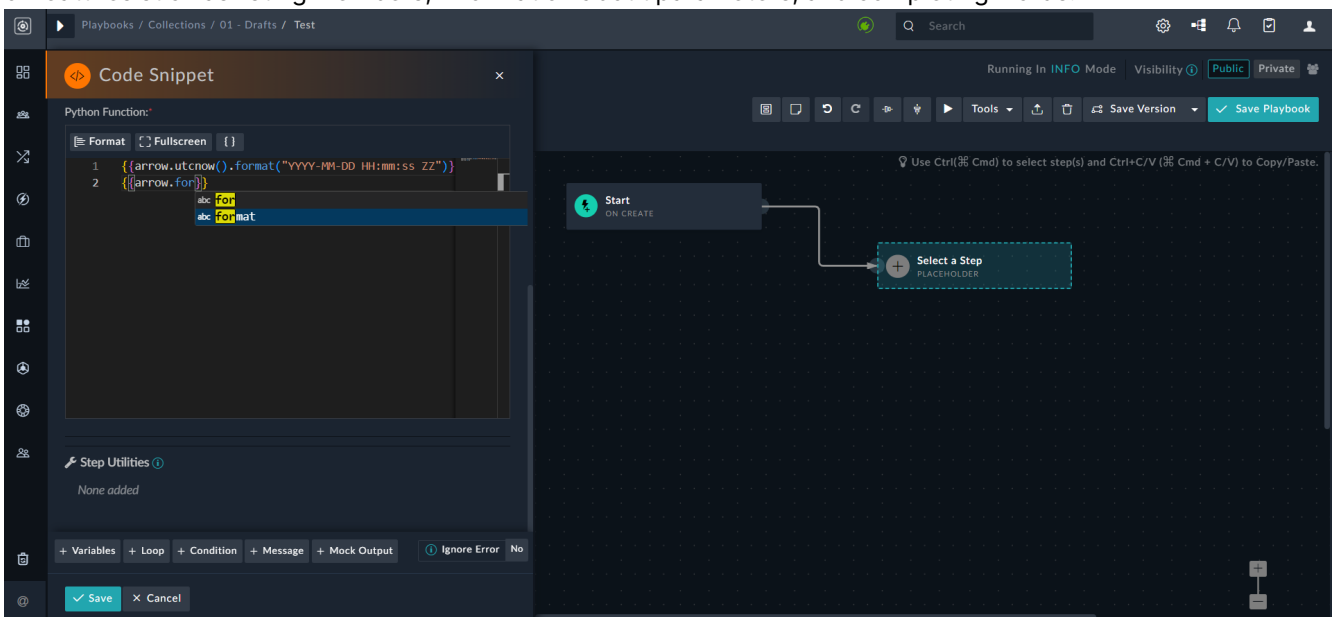


You can also perform the following operations on the code editor interface:

- To lint your code automatically and make the code more human-readable and error-free (syntax and programming errors), select the entire code in the editor and click the **Format** button.
- To get a better working view and make the editor go full-screen, click the **Fullscreen** button. To exit the full screen, press ESC.
- To add dynamic values (Jinja) or variables, or access values of objects, or perform lookups, click the **Dynamic Values** button to display the 'Dynamic Values' dialog. For more information, see the Dynamic Values chapter.

From release 7.6.0 onwards, IntelliSense support is added for playbooks steps while using Jinja variables, functions, etc. IntelliSense serves as an intelligent tool for code or a text completion aid that includes a number

of features such as listing members, information about parameters, and completing words:



The **Python Function** field used to be a text box, which is still available if you select the **Execute Python Code (Deprecated)** action. It is not recommended to use this action.

For practical tips and examples for using the Code Snippet connector in playbooks, see the Code Snippet Connector: Usage Tips and Examples topic.

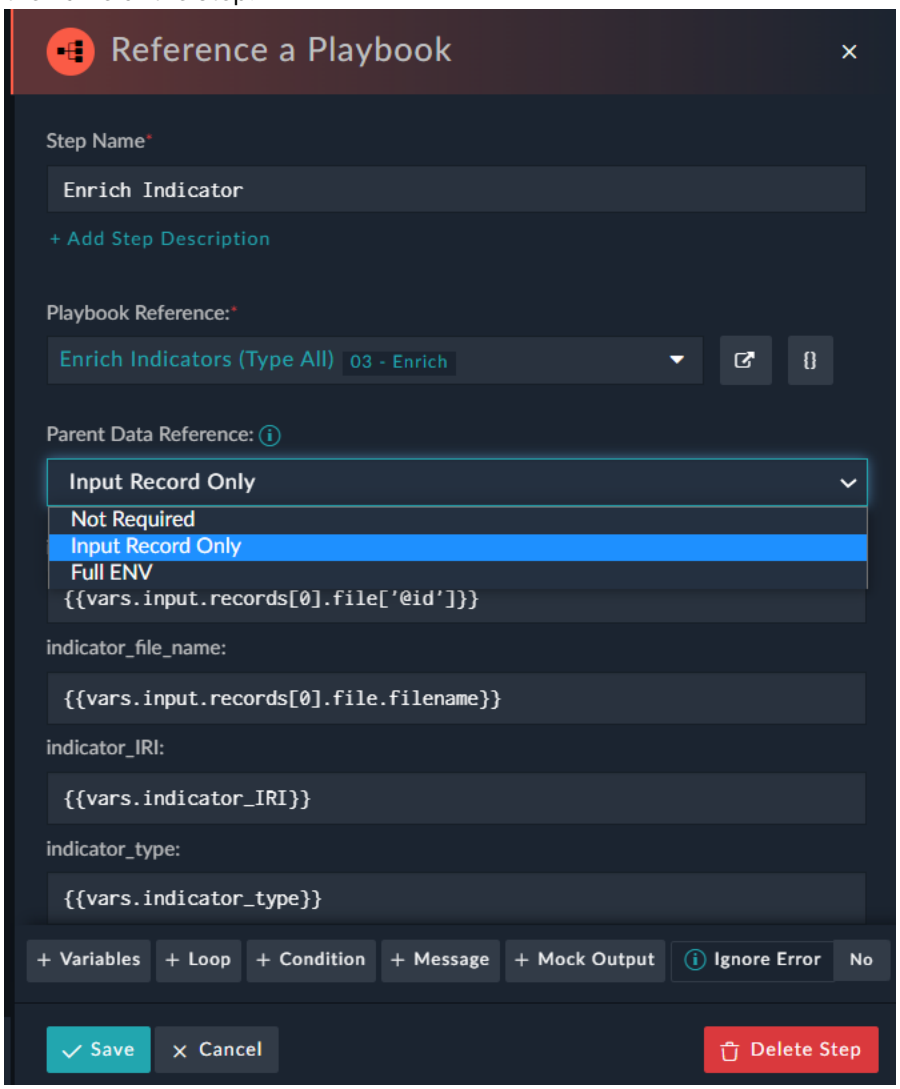
References Steps

Used to call child playbooks from a parent playbook.

Reference a Playbook

Use the **Reference a Playbook** step to call any playbook within the system, whether Active or Inactive, by name. To add a reference to a playbook, click the **Reference a Playbook** step, and in the **Step Name** field, type

the name of the step.



The **Playbook Reference** field displays a list of all the available playbooks across the playbook collections, from which you can select the playbook that you want to reference. Therefore, from the **Playbook Reference** drop-down list, search for and select the playbook you want to reference. You can also click the **Dynamic Values** ( ) button beside **Select Playbook Reference** to specify the Jinja variable that contains the IRI value of the playbook you want to reference. Once you have specified the IRI value or selected the playbook you want to reference, you can click the **Playbook In New Tab** button to open the referenced playbook in a new tab.

The parent playbook can pass data to the reference playbook based on the option you have selected from the **Parent Data Reference** drop-down list:

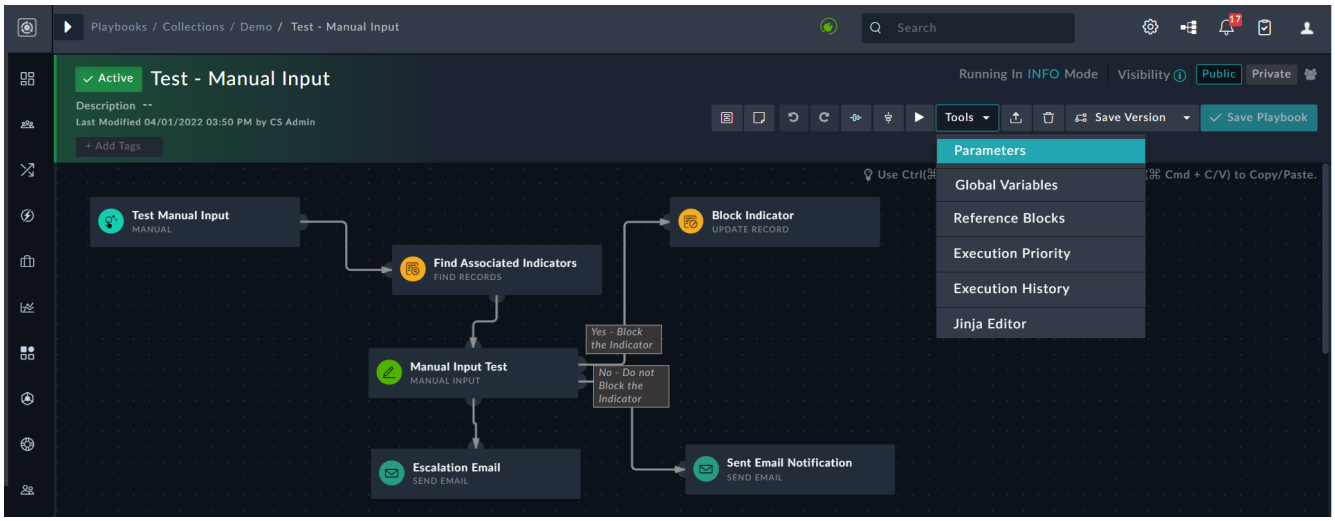
- To pass the complete environment data of the parent playbook to the reference playbook, select the **Full ENV** option. Selecting this option enables the reference playbook to reference any dynamic variable of the parent playbook and use it just as it is being used in the parent playbook.
- To pass the record inputs available under `vars.input.records` of the parent playbook to the reference playbook, select the **Pass Input Record Only** option.
- To prevent any data of the parent playbook from being passed to the reference playbook, select the **Not Required** (default) option. For example, in cases where the referenced step loops on lot of items creating

unnecessary data in memory while is running, and also in the database if the playbook is run in the debug mode, you might not want the parent playbook to any data.

You can use the **Loop** option to iterate over a playbook step as per your requirements.

If you have migrated a Map Playbook to a Reference a Playbook (using Loop), you will observe a change in behavior. In the case of the Map Playbook, any changes done to the environment variables by the Map Playbook made directly reflected in the Main Playbook. However, in the case of the Reference a Playbook, you must explicitly set the returned values from the referenced (child) playbooks in its last step. This ensures that the child playbook does not change the behavior of the main playbook in an unexpected manner.

The reference playbook steps' output varies depending on the called playbook parameters. You can define parameters using **Tools** > **Parameters** in the playbook designer.



If you update any of the parameters in a child playbook, then you must review and make the necessary updates in the Reference a Playbook step in the parent playbook. For example, if delete a parameter from a child playbook, the parameter will still pass from the parent playbook to the child playbook since the input values are saved in the reference playbook step. These inputs are cleared only when you open and save the Reference a Playbook step in the parent playbook.

If you want to use a variable from a playbook that you are referencing (A) in the calling playbook (B), then define that parameter in the referenced playbook (A) using **Tools** > **Parameters**. This is the recommended method of passing environment variables from the referencing (parent) playbook to the referenced (child) playbook. It is *not* recommended to directly use the environment variable (since the parent environment is available in the child workflow as well) without explicitly defining child playbook input. You can turn this feature (passing of parent environment variables) **on** or **off** by updating the following entry in the `celeryd` section of the `/opt/cyops-workflow/sealab/sealab/config.ini` file:
`COPY_ENV_FOR_REFERENCE_WORKFLOW : false`

Restart the FortiSOAR services once you have updated the entry in the `config.ini` file.

By default, the `COPY_ENV_FOR_REFERENCE_WORKFLOW` is set to `false`.

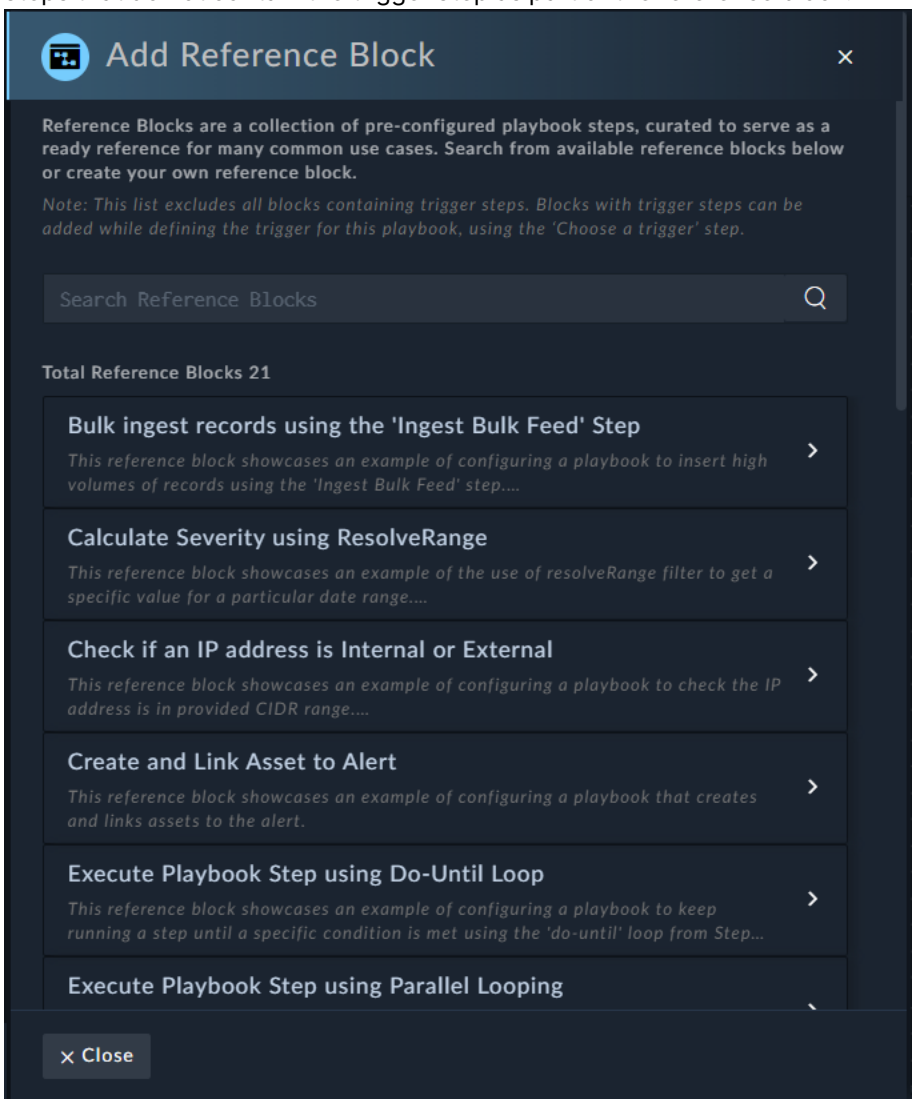
Key Points:

- **Use the Full Env Option with Caution**: When passing data from the parent playbook to the referenced playbook, use the Full Env option cautiously. This passes the complete environment data from the parent playbook to the referenced playbook, granting access to dynamic variables from the parent. This can lead to potential storage and performance issues.
- **Avoid Using Direct Environment Variables**:It is recommended to avoid directly using environment variables from the parent playbook in the referenced playbook without explicitly defining them as parameters. Use the **Tools > Parameters** method to ensure clear and predictable data passing.
- **Updating Parameters**: When parameters in the referenced (child) playbook are updated, review and update the Reference a Playbook step in the parent playbook accordingly. For instance, if a parameter is deleted in the child playbook, it may still be passed from the parent unless the reference step is updated.
- **Configuring the Passing of Environment Variables**: To pass environment variables from the parent playbook to the referenced playbook, update the `celeryd` section of the `/opt/cyops-workflow/sealab/sealab/config.ini` file. Set the value of `COPY_ENV_FOR_REFERENCE_WORKFLOW` to '`true`' and restart the FortiSOARservices.

Add Reference Block

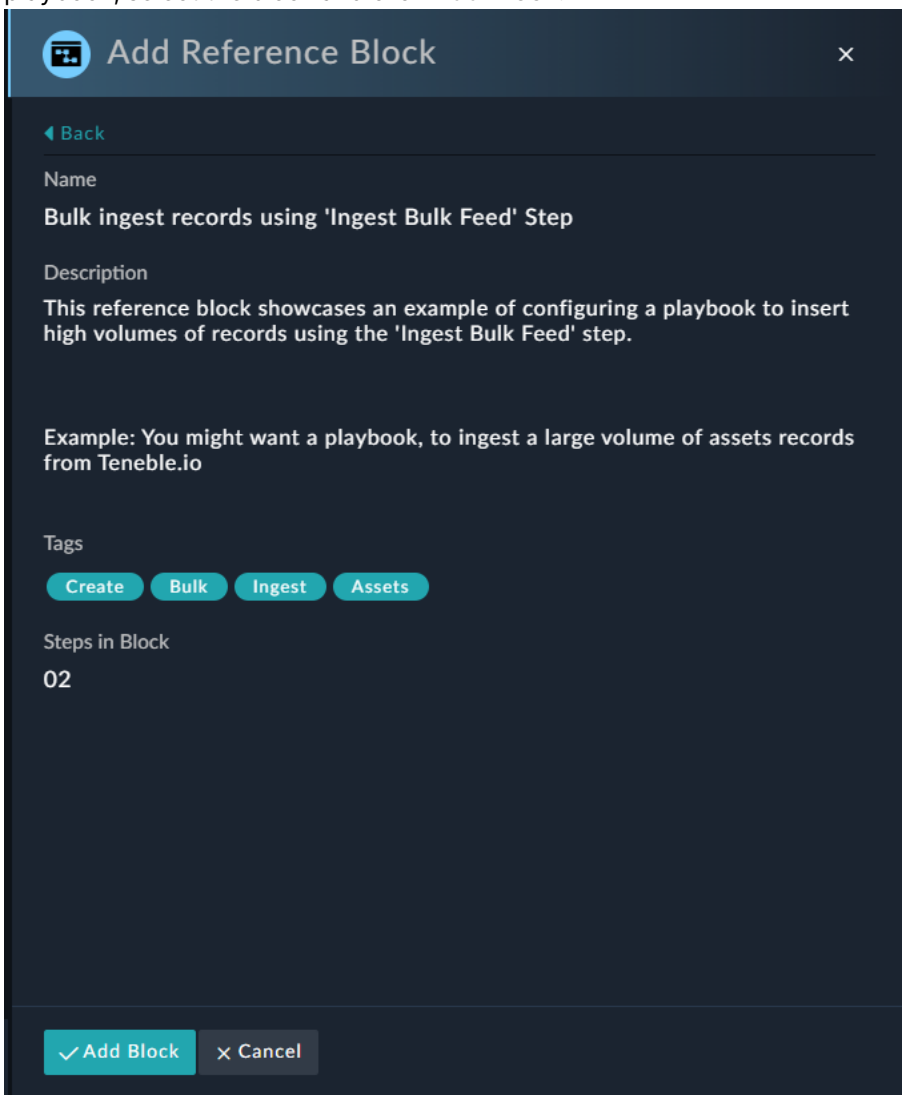
For commonly used playbook blocks that do not contain the trigger step, use the **Add Reference Block** step. The **Add Reference Block** step contains ready-made blocks (steps) included with the SOAR Framework solution pack, or blocks that you have created based on the steps mentioned in the *Adding Blocks and Notes*section, which makes it easier for users to get relevant help reference, samples, and more context around building a playbook. Blocks that do not include the start (or trigger) step get added as 'Reference Blocks'.

To add a block, click **Add Reference Block** in the Playbook Designer, which displays a list of all the reference steps that do not contain the trigger step as part of the reference block:



Use the **Search Reference Blocks** field to search for reference blocks using tags (exact match supported), the name of the block, or its description.
Click **>** or click anywhere in the block row to expand a particular block to get information about the block; information includes the description, tags, and number of steps in that particular block. To add a block to your

playbook, select the block and click **Add Block**:



Clicking **Add Block** displays the `Add Block Details` dialog where you can edit the name and description for the block and then click **Add**, which adds the block in your playbook designer. The block is added to the playbook designer with the name and description, i.e., this is the content that will be displayed when you hover on the **Info** icon that you have specified in the `Add Block Details` dialog.

You can edit the added reference block; however, note that editing the block only updates the step(s) in the current playbook and does not update the step(s) in the reference (original) block. Clicking the **Save as Reference Block** icon in the current playbook displays the `Save as Reference Block` dialog, where you can make changes to the block and save the block with a unique name. To make any changes to the name or description of the reference (original) block, or update its tags, you must use **Tools** > **Reference Blocks** option.

You can export and import playbook blocks using the Export and Import Wizards; see the *Application Configuration and Customization* chapter in the "Administration Guide" for more information. Reference Blocks can also be included in Solution Packs, see the Install and Manage Solution Packs chapter in the "User Guide" for more information.

Use Reference Blocks to:

- Reuse commonly executed logic (e.g., parsing emails, enriching indicators).
- Standardize procedures across multiple playbooks.
- Improve maintainability by modularizing your workflows.

Trigger Tenant Playbook

The **Trigger Tenant Playbook** step is used in the case of MSSP setups. Add the **Trigger Tenant Playbook** step in the playbook that you want to trigger from a master node and remotely execute and retrieve details from a tenant node. For details, see the "MSSP Guide."

Email Step

Send Email Step

Use the **Send Email** step to create a step that will prompt the executed playbook to automatically send an email to the user(s) identified in the step with either specific static criteria or record-relevant data using dynamic values.

If the email needs to reflect data specific to the entity that triggered the playbook, then use *dynamic values* in the fields.

Following are some examples of how you can send an email with jinja content In case of On Create or On Update triggers:

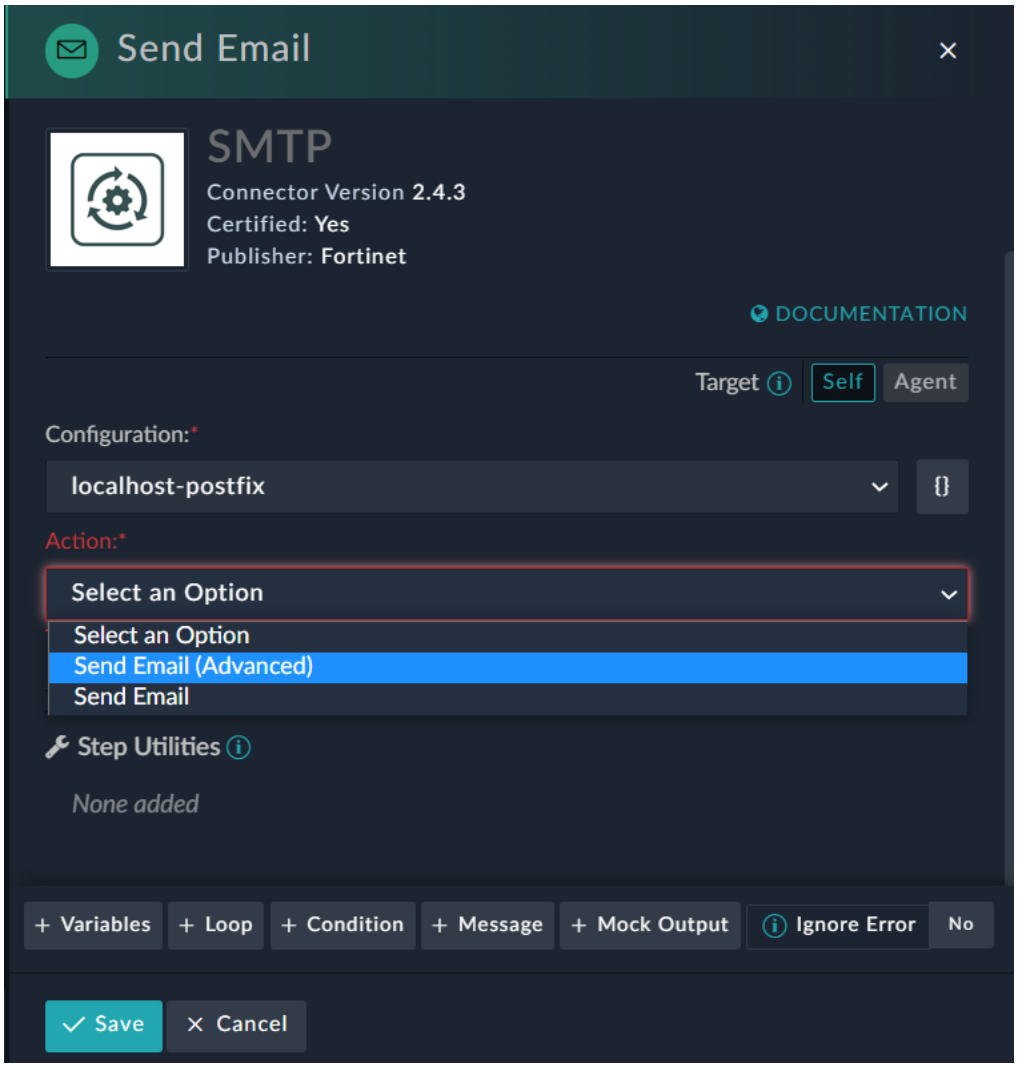
- To send an email to the user who is assigned to a Task, enter the following in the *TO* field: `{{vars.input.records[0].assignedTo.email}}`.
- To set the email subject line as the name of the Task/Incident, enter the following in the *Subject* field: `{{vars.input.records[0].name}}`

If the email will always have the same recipient/content/etc., then enter the text in the corresponding fields and click **Save**.

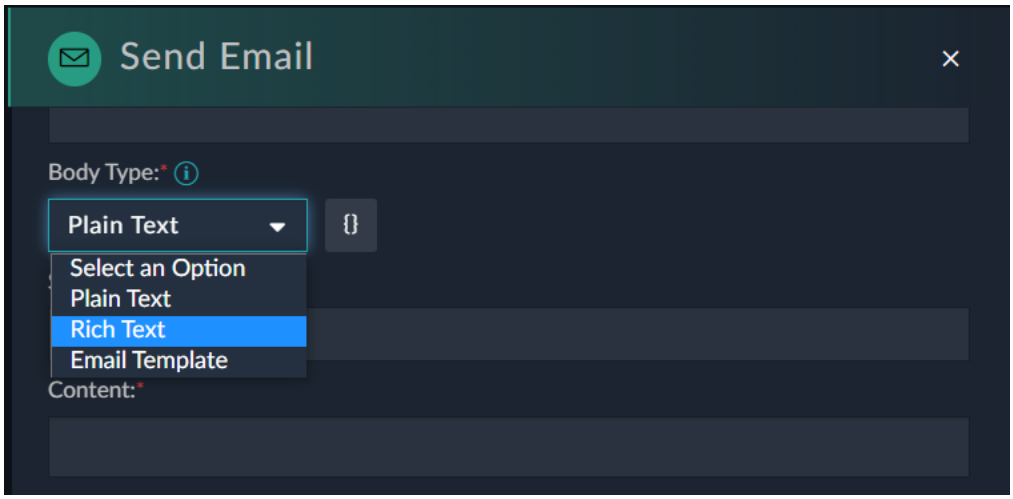
If you have stored a comma-separated list of multiple email addresses in any 'Set Variable' step and you use that jinja variable in the 'TO' field in the 'Send Email' playbook step then the email is not sent to all the email addresses. If you require to send the email to multiple email addresses, you must use the FortiSOAR provided SMTP built-in connector to add multiple email addresses in the 'TO' field. The SMTP connector has a `Send Email` function that supports multiple addresses both in the jinja variable and string formats.

The **Send Email** step uses the SMTP built-in connector and you can send emails to existing FortiSOAR teams or users by selecting teams or users from pre-populated multi-select fields. For more information, see the SMTP Connector document.

The **Send Email** step has renamed to **Send Email (Advanced**). Use the **Send Email (Advanced)** step to send a rich text email with jinja and email template support. Use the **Send Email** step to send a rich text email with Dynamic Values support but no support for email templates:

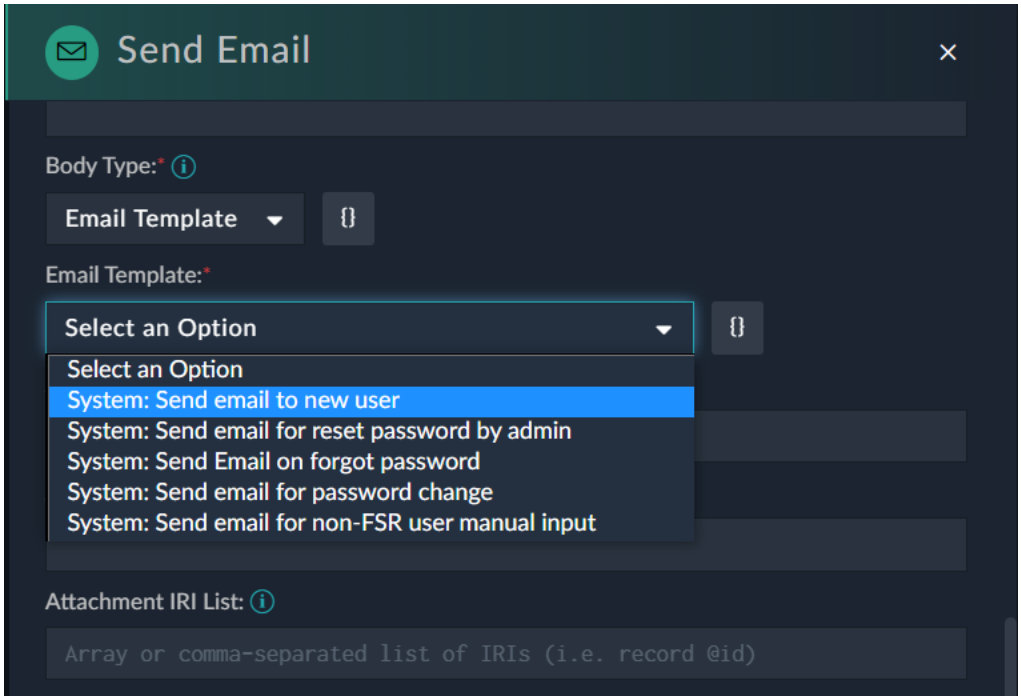


The `Send Email(Advanced)` step provides user with the ability to pass an existing email template as an input for the email subject and body (content), thereby, enabling users to leverage an existing template and build upon it, and therefore, avoid re-work and ensuring consistency. The **Send Email (Advanced)** step contains a **Body Type** drop-down list from which you can choose whether you want to send a plain text email (**Plain Text**), rich text email (**Rich Text**), or an email based on a template (**Email Template**):



If you select **Rich Text** from the **Body Type** drop-down list, then in the **Content** field, you can add formatted content, images, and even custom jinja expressions using Dynamic Values.

If you select **Email Template** from the **Body Type** drop-down list, the **Email Template** drop-down list gets displayed, using which you can select the template that you want to use to send the email:



Tips:

- **Use Meaningful Step Names**: Examples such as 'Notify Assigned User' or 'Escalate to Tier'.
- **Set Up Email Components**:
  - **Email Recipients (TO, CC, BCC)**:
    - Use dynamic values in the "To" field: `{{ vars.input.records[0].assignedTo.email }}`
    - For multiple recipients: `{{ my_email_list | join(', ') }}`
      Avoid using raw lists; always `join()` them into a string.

    - Use team or role-based emails for scalability.
  - **Email Subject**:
    - Use dynamic subjects.
      Example, for the name of the record (Incident/Task/Alert), use:
      `{{ vars.input.records[0].name }}`
    - Make the subject meaningful and easy to search.
    - Include relevant context, such as severity or incident ID.
  - **Choose Appropriate Email Body Type:**:
    - Use **Rich Text** for clarity and professionalism.
    - Use **Email Template** for standardized messages.
  - **Email Content**: Use Jinja to insert record data dynamically, and avoid using overly complex logic in the email content.
    Example of using Rich Text:

```
<p><strong>Incident:</strong> {{ vars.input.records[0].name }}</p>
<p><strong>Severity:</strong> {{ vars.input.records[0].severity or 'Not Set' }}</p>
<p>Please investigate the incident as soon as possible.</p>
```

Authentication Step

Set API Keys Step

You can change the context of the user, i.e., override the default appliance keys using the `Set API Keys` step. For a particular playbook if you wish to run the API steps with less or more privileges than that of the default Playbook appliance, you can do so by adding the `Set API Key` step before the concerned steps in the playbook. In this case the privileges of the specified API key will be used; and this will apply to all steps in the playbook after the `Set API Key` step.

You can also use the `Set API Keys` step to create a playbook using the no authentication webhook (No Authentication trigger) in case of the Custom API Endpoint Trigger. In such a case, to successfully perform any operation, such as creating a record in FortiSOAR, you will require to use the Set API Keys step and provide the appliance keys for authentication.

To use the `Set API Keys` step, open the playbook and click the **Set API Keys** step and in the **Step Name** field, type the name of the step. Next, enter the **Public Key** and **Private Key** values and click **Save**. For details on generating a public and private key, or retrieving the details of a public key, see the Access Keys topic in the *Security and Access Management* chapter of the "Administration Guide."

The owner of the records created or updated by this playbook are the teams who own the appliance whose keys are specified in the playbook.

Considerations:

- **Position the Set API Keys Step**: Always place the Set API Keys step before any steps that require API calls using the specified keys.

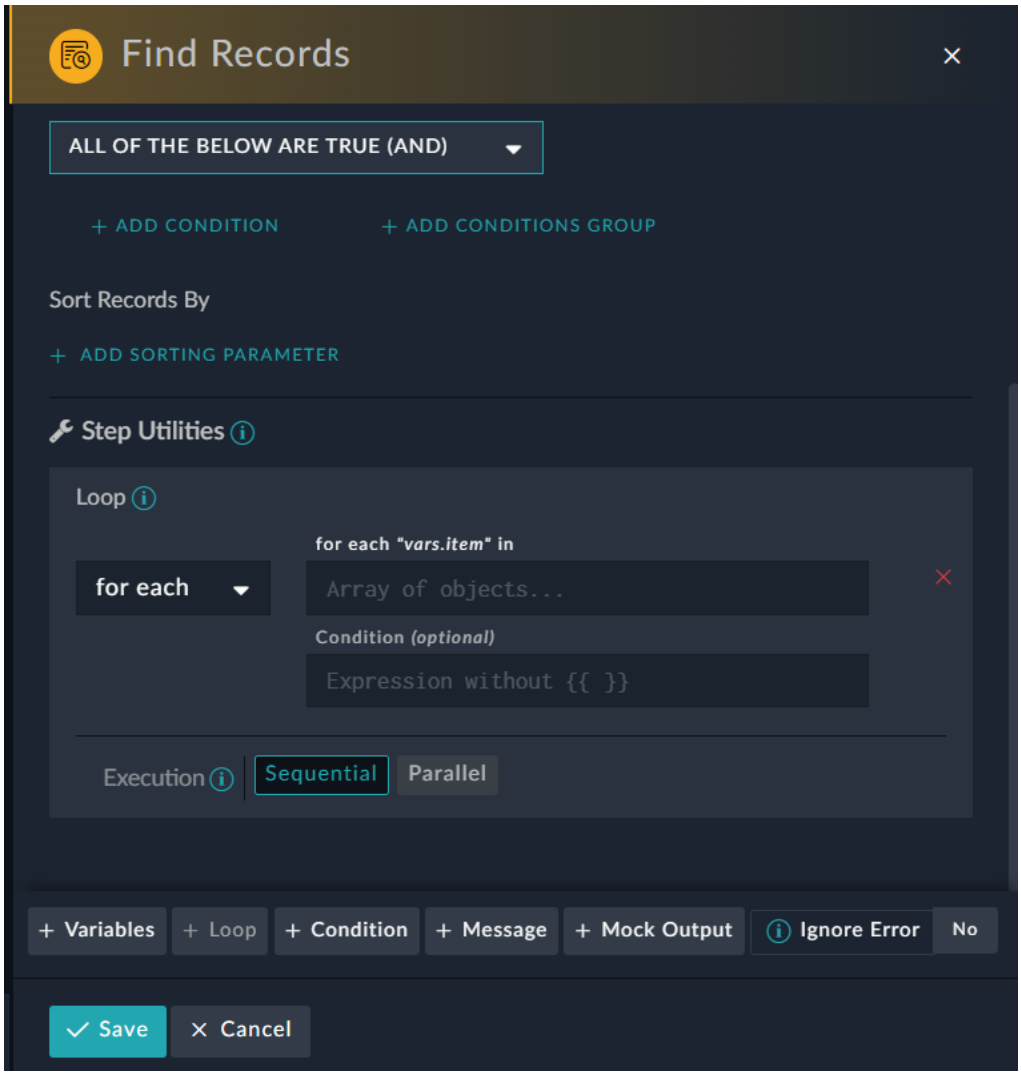
- **Secure Key Management**:
  - Store public and private Keys securely. Avoid hard-coding them in the playbook. FortiSOAR's Password Vault can be used for secure key management.
  - For custom API keys, only provide the necessary permissions for the tasks the playbook needs to perform.
  - Keep API key configurations to the minimum required privileges to reduce security risks.
  - Limit access to API keys to the playbooks or users that require them.
- **Testing**: Always test with a restricted set of permissions to ensure that API calls are using the provided keys without exceeding their privileges.
- **Key Rotation**: Implement a key rotation policy and regularly update API keys to mitigate the risk of using compromised keys.
- **Error Handling and Auditing**:
  - Ensure API keys are correctly configured to access external systems, and set up mechanisms to handle authentication errors.
  - Regularly audit logs for unusual API access patterns or potential misuse.

Extending Playbook Steps with Actions

Condition

To add a condition to a step, click the **Condition** link that is present in the footer of the playbook step. Clicking the **Condition** link adds the **When** textbox, in which you add the expression (condition) based on which the decision to execute the playbook step is taken. If the condition is met, then the playbook step is executed. If the condition is not met, then the playbook step is skipped.

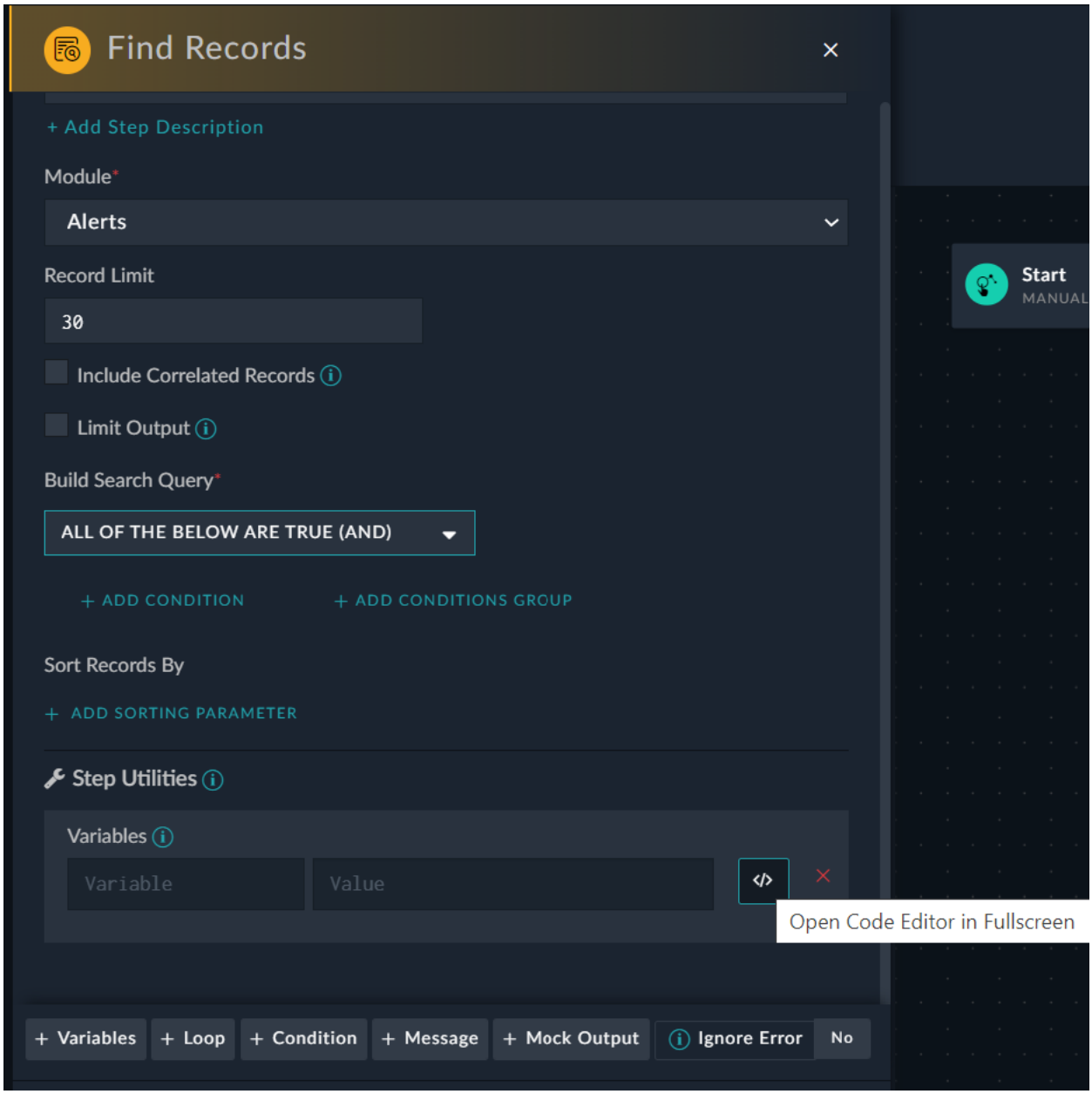
If you use `when` without the `for each` loop, then it applies to the step level and determines whether the playbook step will be executed or not and it is the first thing that is evaluated for the step. If you use `when` with the `for each` loop, then it applies inside the for loop for each item.



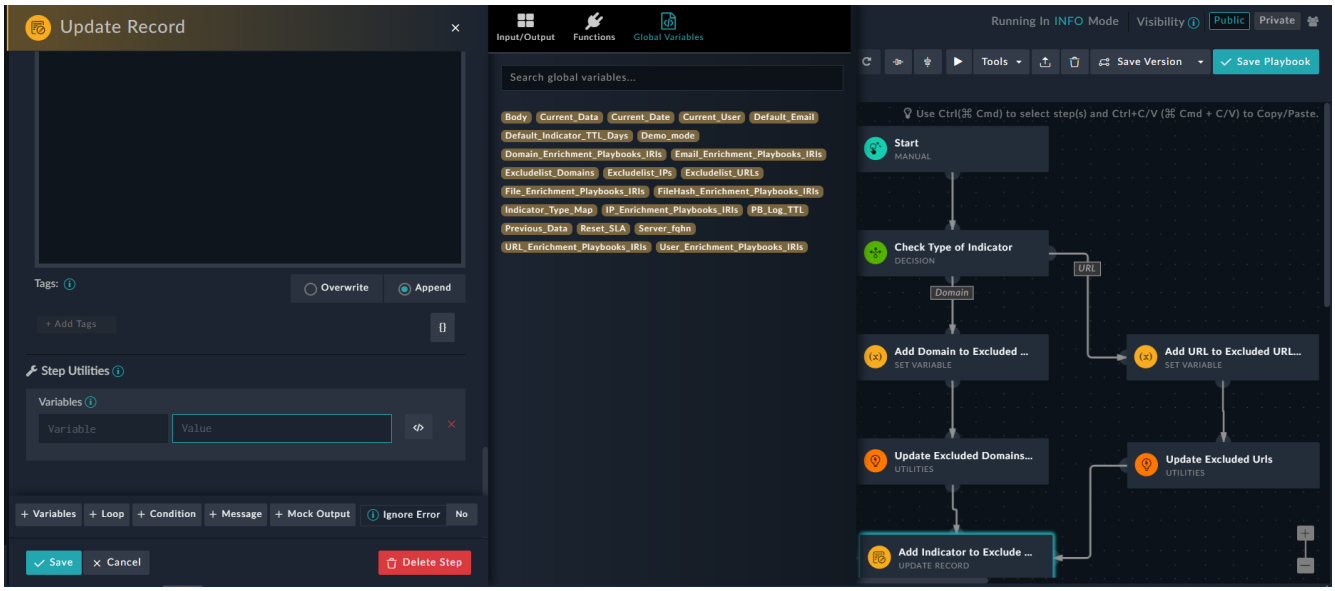
Variables

To add a variable to a step, click the **Variables** link that is present in the footer of the playbook step or add the variable in the `Variables` section of the step. Using **Variables** you can store the output of the step directly in the step itself. Therefore, instead of having to use the **Set Variable** step frequently within a playbook to collect specific response data and provide a contextual name to the output, you can use **Variables** in the step itself. You can also store custom expressions in variables, which can be accessed within the playbook.

Do not use reserved words, which are listed in the Reserved Keywords section as the variable name.



Click the **Open Code Editor in Fullscreen** button to open a code text editor making the experience of adding and editing the code more user-friendly. Clicking the **Open Code Editor in Fullscreen** button opens the code editor in the full-screen mode. To exit the full screen, press ESC or click **Exit Fullscreen**.
Use Dynamic Values to add or store the output of the current step directly in the step itself as shown in the following image:



For more information on Dynamic Values, see the Dynamic Values chapter.

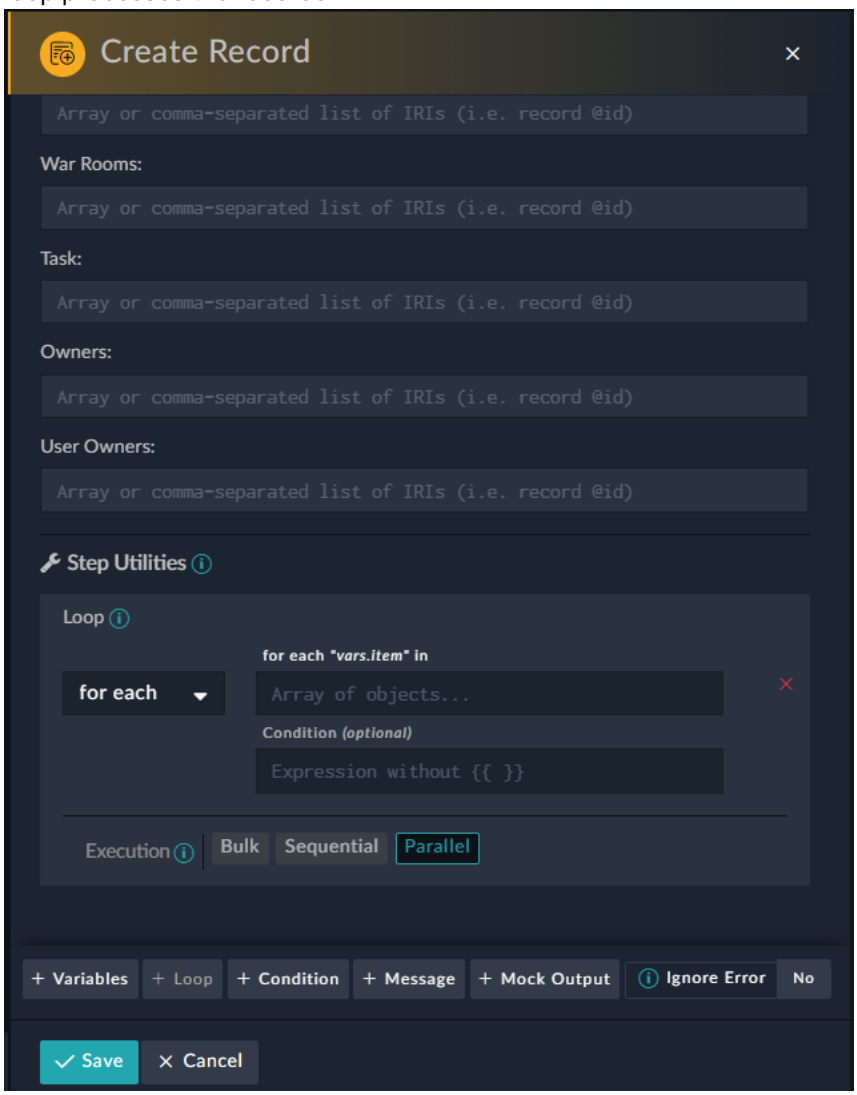
Loop

To iterate over a playbook step, click the **Loop** link that is located in the footer of the playbook step. There are two types of loops that you can add to a playbook step: the **for each** loop and the **do until** loop.

For Each loop

The **for each** loop allows you to iterate through an array of objects. It can only be added once in a playbook step. The input for the `for each` loop is an array of objects. The loop will run for each object in the array. You can access the current object using the reserved keyword `item`.
An example of an array of alerts objects is:
`[{"name":"Alert Name1"},{"name":"Alert Nam2"},{"name":"Alert Name3"}]`
To access the properties of the array items, use `vars.item.propertyName`. For example, to access the name of an alert, use `vars.item.name`.
You can optionally add a condition to the `for each` loop to control when the loop executes.
Loop Modes

The `for each` loop option supports three modes: **Bulk**, **Sequential**, or **Parallel**. These modes determine how the loop processes the records:



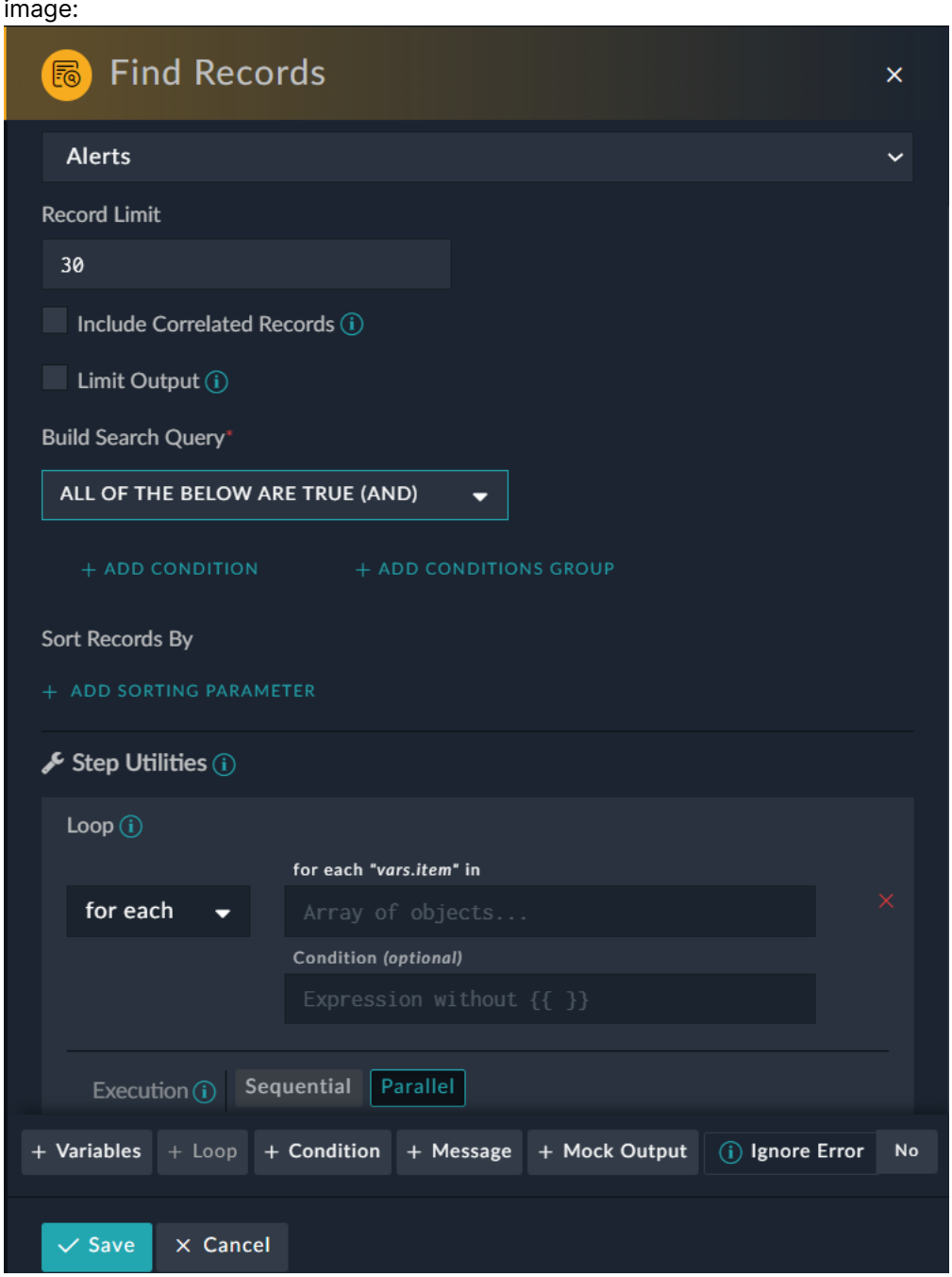
For Each Loop - Bulk Mode

In the **Bulk** mode, all records are processed in a single API request, making it the most efficient and recommended method for creating or updating records in bulk. This is the default mode when adding a new "Create Record" or "Update Record" step in a loop. If you are working with a large number of records that could cause an API call to timeout, consider using batching to insert records in smaller sets. For more information, see the Batching large datasets when using the 'Bulk' option section.

For Each Loop - Sequential Mode

In the **Sequential** mode, the loop processes records one at a time in a serial manner, whereas in the **Parallel** mode records are processed concurrently using multiple threads. This speeds up execution by running independent paths in parallel. You can choose your option using the **Execution** toggle as shown in the following

image:

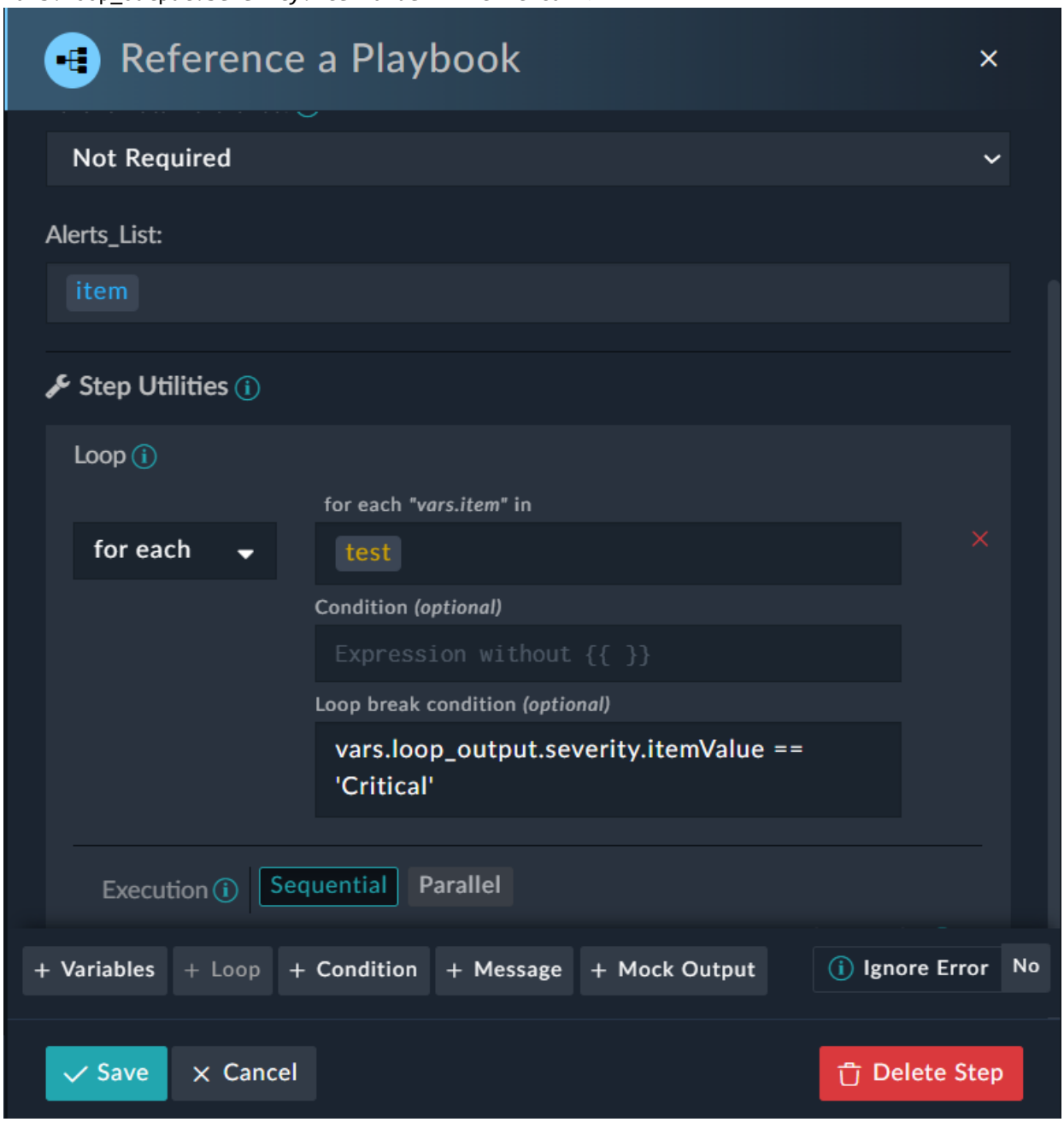


The **Sequential** mode sends the API, records separately for individual records one after another. If an error occurs with any record, the playbook stops, and no further records are processed. Starting from release 7.6.5, Sequential Mode (when the Run Asynchronously checkbox is not selected) includes the **Loop break condition** option for any step. This option allows you to specify a condition that will stop the loop and return the results up to that point. The 'Break Loop' condition can be applied using the reserved keyword "`loop_output`", which stores the result of that loop item's output.

For example, consider enriching a list of IP addresses from an alert with threat intelligence, and stopping the loop if a match is found in a critical threat feed. To do this, select the **Loop break condition** option on for example, a 'Reference a Playbook' step, where each loop item result is as follows:

```
{"id" : "1", "severity": {"itemValue" : "Critical", "@id" : "cac1612c-3005-4ede-8133-
```

`f1528c1b6c2a"}}`
In this case, you can break the loop using the following Jinja2 expression:
`vars.loop_output.severity.itemValue == 'Critical'`:



NOTE: When using a `for_each` loop with a 'Reference a Playbook' step, `vars.loop_output` contains the final result of the referenced playbook.
Additionally, ***do not use*** use Connector actions that run on an Agent in child playbooks when applying a loop break condition in a Reference Playbook. Doing so will result in the error: "`CS-WF-40: Terminated because a child or sibling workflow failed`". This occurs because the child playbook does not receive a response from the Connector action running on the Agent at the time the step is called. Therefore, running a playbook with a loop break condition is not supported when the child playbook includes an agent configuration.

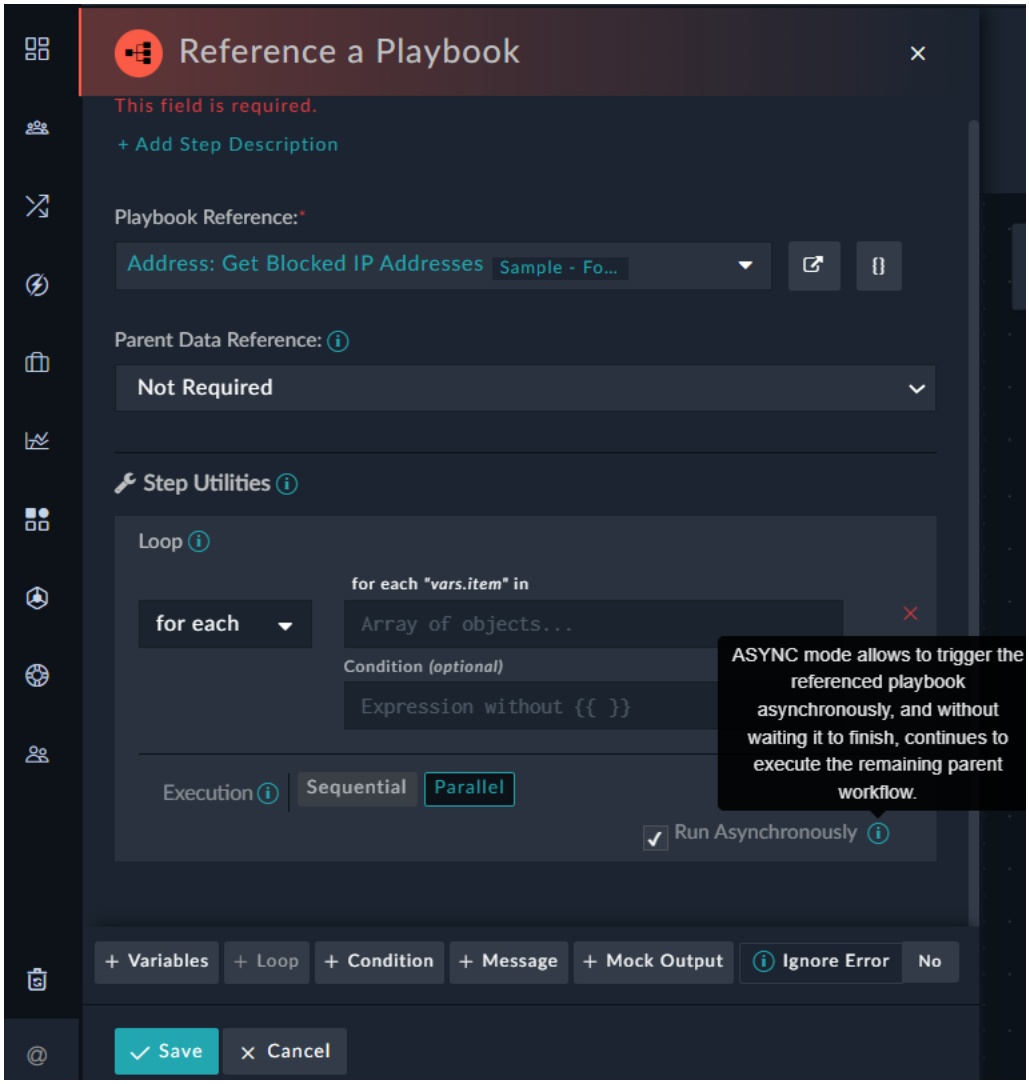
For Each Loop - Parallel Mode
The **Parallel** mode sends separate API requests concurrently using multiple threads. The workflow engine can

execute multiple independent paths in parallel threads. This speeds up execution as **Parallel** branch execution means that two or more paths execute the independent paths in parallel. This enhancement is transparent to the end-user, but in some cases, this could lead to a change in the behavior of certain playbooks when compared to the old sequential behavior as the order of execution could change.
Note: If any existing playbooks fail due to missing data from a prior step such as, previous step result not found, or similar reasons, you can test the playbook with parallel execution disabled to troubleshoot the issue.

To enable or disable parallel execution, modify the `PARALLEL_PATH` (true/false) variable in the `[Application]` section of the `/opt/cyops-workflow/sealab/sealab/config.ini` file. By default, this variable set to `true`.

Additionally, you can also configure the thread pool size and other parallel execution settings. For more information about settings that you can set for optimizing your playbooks, see the Optimizing and Troubleshooting chapter.

For Each Loop - Run Asynchronously
You can execute a referenced playbook asynchronously from the parent playbook by clicking the **Run Asynchronously** checkbox. In this case, the reference playbook can be triggered asynchronously and parent playbook continues to execute the remaining workflow, without waiting for the referenced playbook to finish.

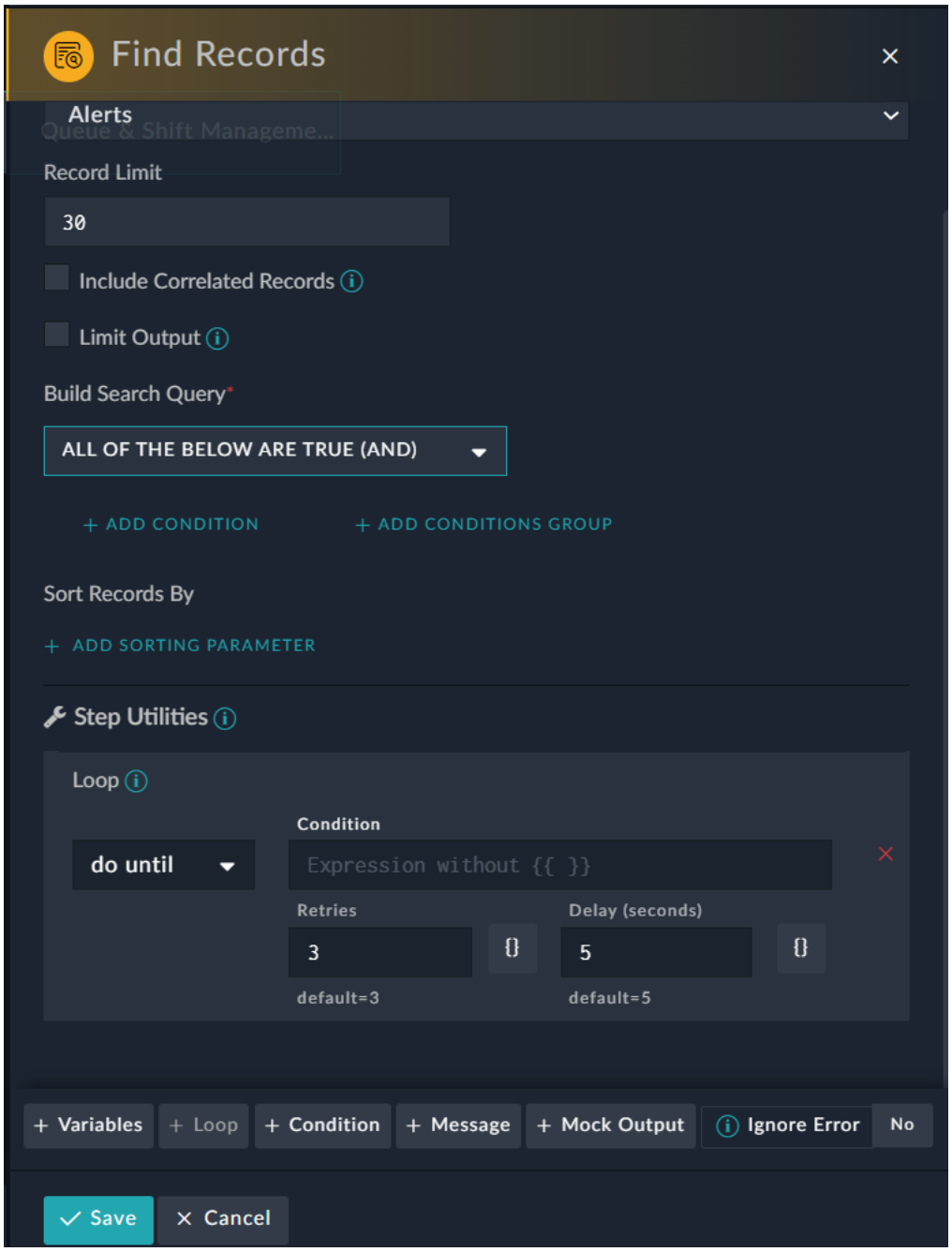


If you select a child playbook to be executed as asynchronously, then you will be unable to use the output of the child playbook in the parent playbook. This mode is ideal for situations where you want to process records without relying on the child playbook's output, such as when ingesting data.

Do Until Loop

The **do until** loop ensures that a step is executed at least once and continues running until a specified condition is met or the number of retries is reached. You can configure the number of retries the playbook step will execute to meet the condition and also the delay in seconds before the step gets re-executed in a loop. By default, the number of retries is set to `3` and delay is set to `5` seconds.
Within a do until loop, you can access the result of the current step using `vars.steps.<step_name>.keyname`. For example, to repeatedly attempt a connector action until it succeeds, use the condition: `vars.steps.<step_name>.message == 'Success'`. Additionally, to ensure the playbook does not stop if the step fails, check the **Ignore Errors** box.

Do not use the `do until` loop with `when` or `for_each` conditions.



Prevention of Infinite Looping

To prevents infinite looping in playbooks triggered by 'On Create' or 'On Update' triggers, FortiSOAR uses the `skip_recursive_playbook_execution` parameter with its value set to '`true`' in the `/opt/cyops-api/config/parameters_prod.yaml` file.This is useful for scenarios such as updating a record without a condition, which could otherwise result in endless recursion.

The default value for `skip_recursive_playbook_execution` parameter is '`true`', meaning the playbook will only run once in such cases.

If you want to change this behavior:

1. Edit the `/opt/cyops-api/config/parameters_prod.yaml` file and set the `skip_recursive_playbook_execution` parameter to '`false`.'
2. Run the following command:
   `sudo systemctl restart php-fpm && sudo -u nginx php /opt/cyops-api/bin/console cache:clear && sudo systemctl restart php-fpm`

Additionally, FortiSOAR limits recursion of self-references in playbooks to '10', using the '`REF_SELF_PB_LOOP_LEVEL`' parameter in the `/opt/cyops-workflow/sealab/sealab/config.ini` file. You can adjust this value based on your needs, but do not set it too high.
If you are on a release earlier than 7.3.1, you can add the '`REF_SELF_PB_LOOP_LEVEL`' parameter to the the '`[application]`' section in the `/opt/cyops-workflow/sealab/sealab/config.ini` file.

Accessing Iteration Results

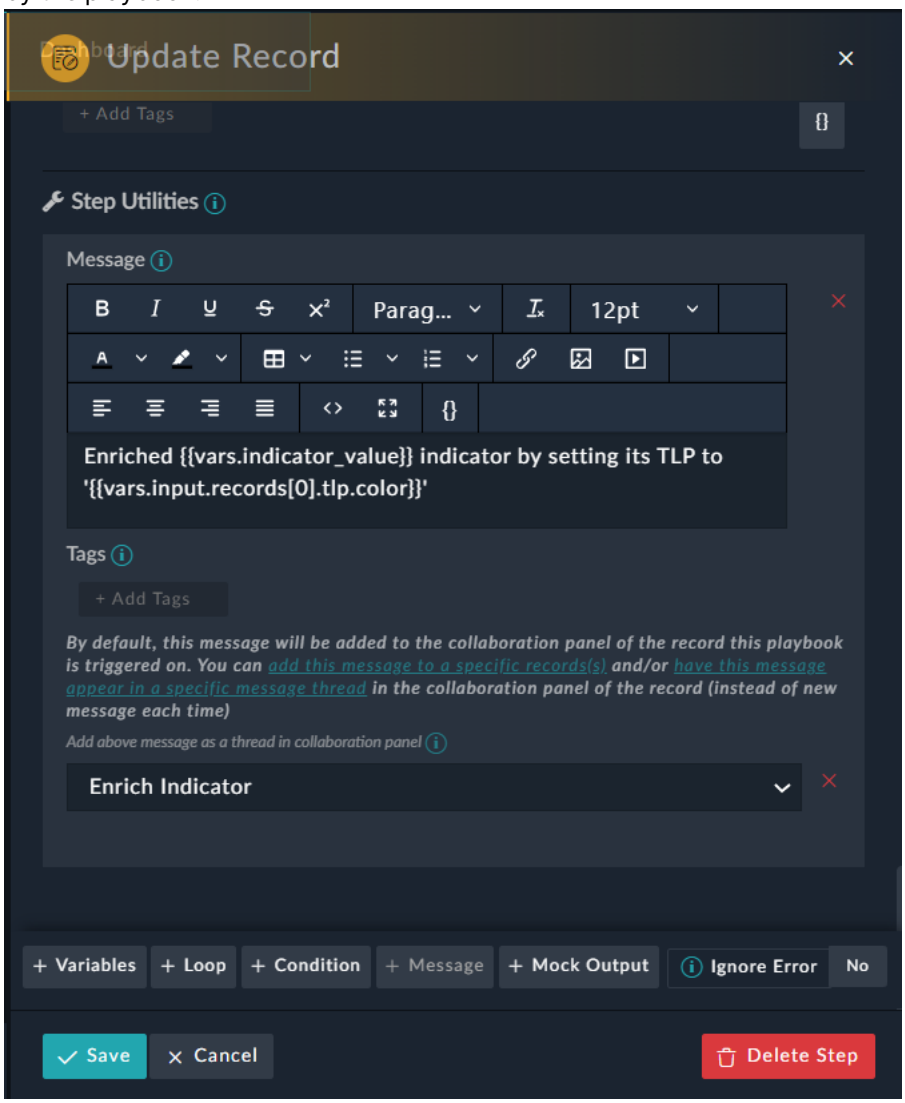
When running a step in a loop, you might need to access the results of the previous item iteration in subsequent iterations. For example, when using LLMs (Large Language Models) to generate steps, you may want to use the results of one iteration as input for the next. To enable this, add `save_iteration_result` as an environment variable and set its value to '`true`'. You can access the results of the previous iteration using `iteration_result_<step_name_with_underscore>`. This works for both `for each` (Sequential mode) and `do until` loops.

Message

You can add a custom message for each playbook step to describe its behavior. You can also use Dynamic Values to add jinja values to the messages. Dynamic Values also displays the output of the current step in the **Message** step. For more information on Dynamic Values, see the Dynamic Values chapter.

These messages appear in playbook logs and are also displayed as part of the collaboration panel of the record. The Message content can be rendered in HTML or Markdown, depending on the whether you have set the **Contents** field in the "Comments" module to Rich Text (Markdown), which is the default or as Rich Text (HTML). By default, the message is added to the collaboration panel of the record that triggered the playbook. You can also choose to add the message to other records, by clicking the **add this message to a specific record(s)** link, and then in **Specify the record ID(s) of the record to** add the message on field, enter an array or comma-separated list of IRIs of the record(s) to whose collaboration panel the message requires to be added.
You can also choose to display the message in a specific message thread in the collaboration panel of the record, instead of displaying a new message each time. To include messages in a thread, select the **have this message appear in a specific message thread** link, and then in the **Add above message as a thread in collaboration panel** field, select the specific step (which adds the message) to associate with the thread. If you do not select any specific step, then by default, the message is added as a thread to the first message posted

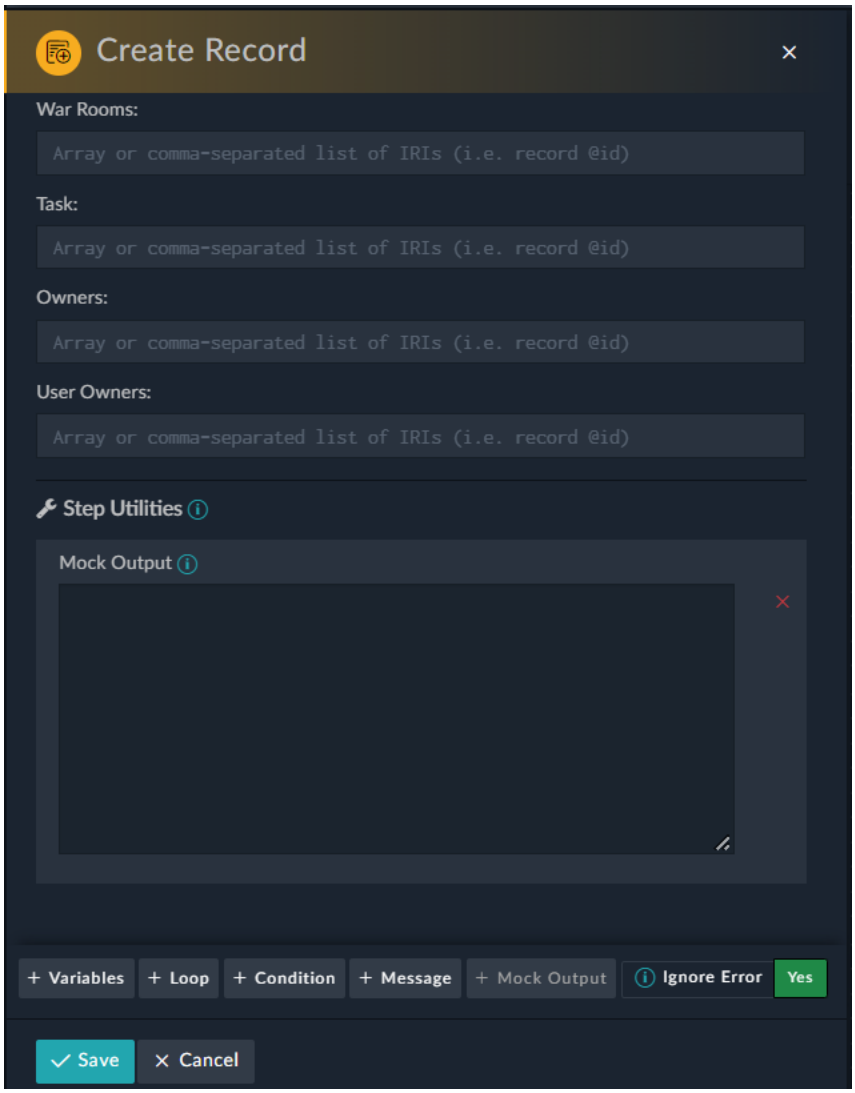
by the playbook:



In case of multi-tenanted configurations, if a playbook that contains steps with "Messages" is added to the record that triggers the playbook on the master node, you can choose to replicate the comments that are linked to the record on the tenant node, so that a user on the tenant node can follow the investigation that is being conducted on the record. To replicate comments on the tenant node, click the **Also send this message to specified tenant** checkbox, and from the **Select Tenant** drop-down list, select the tenant node on which you want to replicate the comments or click **{}** to specify tenant IRIs in this field.

Mock Output

You can use mock output (in JSON format) for a step to simulate real outputs for debugging purposes. Mock output will override the actual step output when the playbook is run.



To enable mock output, either trigger the playbook with the **Use Step Mock Output** option or add a variable named `useMockOutput` and set its value to '*true*' in the trigger step. If this variable is set to false or not declared, the playbook will use actual step outputs. Also, ensure that you write `useMockOutput` as is since this variable name is case-sensitive. For more information on variable, see Variables.

Ignore Error

You can configure a step to continue executing even if it fails by clicking the **Yes**/**No** button next to **Ignore Error**. If this option is enabled, the status of this step will be `Finished with Error` in the playbook log. Open the playbook log by clicking the **Executed Playbook Logs** icon ( ) that appears on the top-right corner of the FortiSOAR screen. Click the step whose log you want to view, and in the `Step View` section, the status of the playbook is displayed in the `status` item, and the error is described in the `result` item.

Tips for Working with Playbook Steps

- **Write Clear Step Descriptions**: Each step should include a brief description that explains its purpose, logic, or context—especially for steps involving conditions, loops, or expressions. This helps users understand the rationale behind the step without opening its full configuration.
- **Use Appropriate Step Types**: Select the step type that best suits the task.
  Example:
  Use Manual Input steps when human intervention is required such as, requesting approval before blocking a port or assigning alerts during triage.
- **Add and Manage Steps**: Drag and drop steps into the playbook designer to add them. To edit or delete steps, double-click on the step.
- **Centralize Variable Declarations**: Add a "Configurations" step at the beginning of the playbook to declare all variables. This improves readability and simplifies maintenance.
- **Avoid Redundant Loops**: Do not loop within the same playbook unless necessary. Consider breaking the logic into child playbooks to improve manageability.
- **Map Decision and Input Steps Carefully**: Ensure that decision and user input steps are properly connected to the correct paths to prevent logical errors.
- **Optimize with Step Utilities**: Optimize playbooks using step utilities such as the 'Variable' option within a playbook step, instead of adding a separate 'Set Variable' step. For details, see the Extending Playbook Steps with Actions topic.
- **Avoid Reserved Keywords as Variable Names**: Do not use Reserved Keywords for variable names. These are reserved for internal use and may cause conflicts or unexpected behavior.

Deprecated Playbook Steps

If you are using a deprecated step or trigger in a playbook, in cases where you have upgraded from an older version of FortiSOAR, then that playbook will continue to work, and you can edit the deprecated step. In case of deprecated steps, FortiSOAR displays a message such as "`This step is deprecated.....`"

If you are using deprecated steps or triggers in your playbook, it is highly recommended that you replace those steps and triggers because over time these steps and triggers will become obsolete, and FortiSOAR will not be able to support or respond to them. You can replace the deprecated steps with the **Utilities** step or by using the FortiSOAR built-in connectors, such as SMTP, Code Snippet, Database, etc. For more information, see the FortiSOAR Connectors page and the Utilities Connector document.

Deprecated Playbook Triggers

Pre-Data Operation Triggers

Pre-data operations have been deprecated and they are intended for synchronous operations, where the data operations might potentially block or affect the final data updates to the database. Therefore, pre-data operation triggers perform some action before the data operation is completed in the database.

Example of a pre-data operation trigger: Suppose your organization has an allowlist database and you want to ensure that before an alert is created its IP address is checked against the database. If the IP address is part of the allowlist database, you do not want an alert to be created.

The following table lists the types of Pre-Data Operations triggers that have been deprecated:

Deprecated Playbook Trigger Name	Brief description of the trigger
Pre-Create	This trigger starts the execution of a playbook immediately before inserting the selected model type to the database. You can create a Pre-Create trigger on almost all models, which includes Modules.
Pre-Update	This trigger starts the execution of a playbook immediately before updating the selected model type to the database. You can create a Pre-Update trigger on almost all models, which includes Modules. You add a Pre-Update trigger in the same way you added a Pre-Create trigger.
Pre-Delete	This trigger starts the execution of a playbook immediately before deleting the selected model type to the database. You can create a Pre-Delete trigger on almost all models, which includes Modules. You add a Pre-Delete trigger in the same way you added a Pre-Create trigger.

Creating and Designing Playbooks

Playbook Designer

The Playbook Designer in FortiSOAR provides a visual interface where users can add steps and define playbook logic. Using drag-and-drop, you can add, edit, arrange, and connect elements to create and manage workflows graphically.

Within the playbook designer, you can:

- **Set appropriate logging levels**:
  - **INFO** (Default):
    - Captures essential execution details, recommended for production environments. This setting optimizes performance, conserves storage, and minimizes log clutter.
  - **DEBUG**
    - Captures detailed logs, including variable values, step outputs, and failure traces. Use DEBUG for designing playbooks or troubleshooting errors. By default, failed playbooks are automatically logged in the DEBUG mode to facilitate issue diagnosis without rerunning the playbook.

      DEBUG logging generates extensive logs that may quickly consume disk space. Regularly review and clear old logs or adjust log rotation settings, if available.

      For information on 'Executed Playbook Logs', which are used to view the logs and results of your executed playbooks and assist in debugging, see the Playbook Execution and Debugging chapter. For information on how to set log levels, see the Setting the logging levels for playbooks topic. Additionally, administrators can configure default global logging levels for all playbooks, which individual playbooks can override. For details see Configuring the logging level for Playbook Execution Logs topic in the *Application Configuration and Customization* chapter of the "Administration Guide."
- **Assign Playbook Ownership appropriately**: Manage permissions and ownership to control access to playbooks. For details, see the Ownership, Roles and Permissions topic.
- **Manage Parameters and Global Variables**: Parameters are variables local to a playbook; Global Variables are accessible across multiple playbooks. Add these via **Tools** > **Parameters** or **Tools** > **Global Variables**. For details, see Editing Playbooks and Global Variables topics.
- **Create Reference Blocks**: Use Reference Blocks (**Tools** > **Reference Blocks**) to access reusable logic snippets and contextual help. For details, see the Reference Blocks topic.
- **Save Playbook Versions**: Save versions of your playbook to track changes over time, roll back to previous working states, collaborate without losing prior logic, experiment safely without fear of losing working logic, and maintain a history of significant updates with context.
  Notes:
  - Manually save a named version before making major changes.
  - Always include versions when exporting playbooks.
  - Share exported playbooks with versions to provide full context for collaboration.

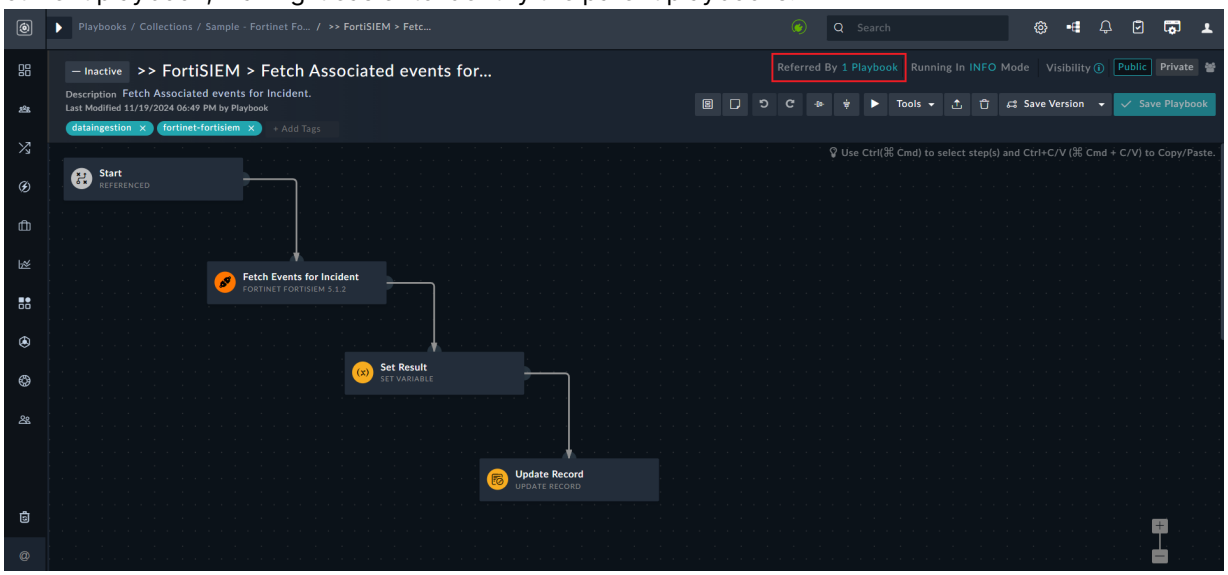
    - Add clear, concise notes (who, what, why) describing the changes (e.g., "Added phishing URL lookup" or "Reworked remediation path") for better traceability.
    - Regularly clean up outdated versions, especially when collaborating with a team.

  You can also export playbooks with versions to back up work, share complete histories with peers or across environments, and restore earlier snapshots after import/export.
  For details, see the Saving Versions and Exporting Versions topics.
- **Playbook Recovery**: FortiSOAR autosaves playbooks every 15 seconds after a change (default). Autosaves are user- and browser-instance-specific, stored locally in the browser.. This settings can be changed by an administrator. For details, see the Playbooks Recovery topic.
  Notes:
    - Avoid using Incognito mode, as autosaves are not retained and are lost if browser data is cleared.
    - Administrators can adjust autosave intervals to balance safety and performance
- **Trigger and Test Playbooks directly from the Playbook Designer**: Run and debug playbooks directly within the designer, simplifying development and troubleshooting. For details, see the Playbook Execution and Debugging chapter.
- **Change Playbook Execution Priority**: **Use Tools** > **Execution History** to prioritize playbook execution based on importance. This ensures higher-priority playbooks run first, even if normal-priority playbooks are already queued. For details, see the Changing Prioritization of Playbook Execution topic.

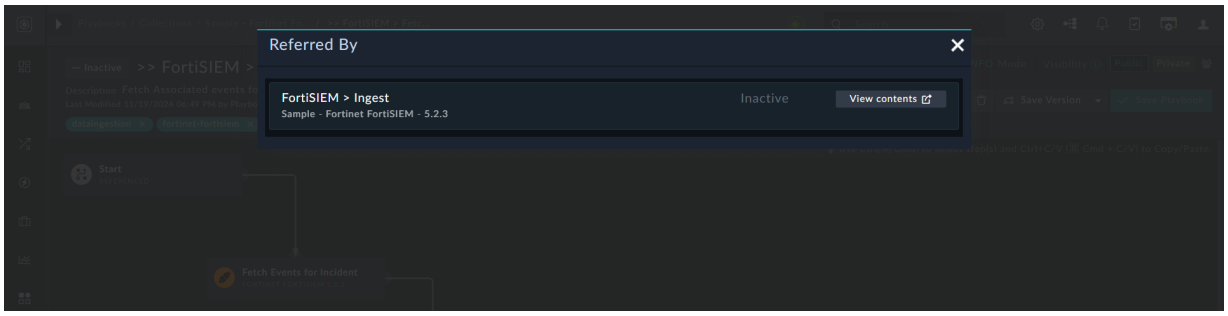
Tips on Designing a Playbook

- **Start with a Trigger step**: Every playbook begins with a trigger that initiates the workflow.
- **Use a Decision Step immediately after the trigger**: Define specific conditions that must be met for the playbook to continue. While the playbook always initiates, the decision step determines whether it proceeds or exits.
- **Add Integrations**: Drag and drop connector steps into the Playbook Designer to integrate external systems.
- **Use Blocks to group related steps**: Logically group multiple playbook steps that share a common purpose, such as *Configure*, *Investigate*, or *Remediate*. For details, see the Adding Blocks and Notes topic.
  Notes:
    - Blocks cannot be nested.
    - Notes cannot be placed inside blocks.
    - A step can belong to only one block at a time.
    - Blocks can be saved as 'Reference Blocks' for reuse in other playbooks. Provide a unique name (3–140 characters) and optional tags. Saved blocks can be accessed via **Tools** > Reference Blocks.
    - In Executed Playbook Logs, hover over the block info icon to see descriptions, total execution time for the block, and step-level timings (e.g., "*Find Indicators" took 137ms*").
- **Add Notes in the Playbook Designer**: Use notes to explain logic, add debug tips, or leave instructions for collaborators.
  Notes:
    - Keep notes concise and meaningful.
    - Use the **Hide Note in Logs** option for internal notes that should not appear in execution logs.
- **Use Jinja2 filters**: Apply Jinja2 filters to manipulate and format playbook step outputs. For details, see the Jinja Filters, Functions, Expressions, and Extensions chapter.

- Use the following tips to work more efficiently with playbooks and playbook steps in the Playbook Designer:
  - You can select a step by the `CTRL+Mouse click` operation. To select all the steps, press `CTRL+A`.
  - You can drag and drop multiple selected steps.
  - You can copy multiple selected steps by pressing `CTRL+C` or copy all the steps by pressing `CTRL+A` and then pressing `CTRL+C`. Ensure that you have clicked on your playbook canvas to bring it into focus before you copy the step(s).
    NOTE: When pasting playbooks with the trigger step that are copied from external sources or widgets such as playbooks created using 'FortiAI', the trigger step will be removed and only the remaining steps will be pasted in the playbook designer for playbooks that have an existing trigger step. However, if you are pasting the playbook into a blank playbook designer (i.e., without a trigger step) then the trigger step along with all the other steps will be pasted.
  - You can paste the copied step(s) into a different playbook by using `CTRL+V`. Ensure that you have clicked on your playbook canvas to bring it in focus before you paste the step(s).
    Note: You can also select **Paste** from the **Edit** menu in your browser to paste the copied steps.
  - You can delete a step or multiple steps by selecting steps and pressing the `backspace` or the `delete` button.
  - You can use the **Auto-Align - Vertical** and **Auto-Align - Horizontal** buttons to align the playbook vertically or horizontally. You can use these buttons to make your playbook look neat and organized, which is especially useful for very large playbooks where playbook readability might be an issue.
  - You can use the **Pan** and **Zoom** tools in case of large playbooks. Use the Pan tool to scroll through your playbook and the Zoom tool to view the details of the playbook.
  - If other playbooks reference the current playbook, you will see '**Referred By: <number> Playbook**' at the top of the playbook designer canvas. This indicates how many playbooks are referencing the current playbook, making it easier to identify the parent playbooks:

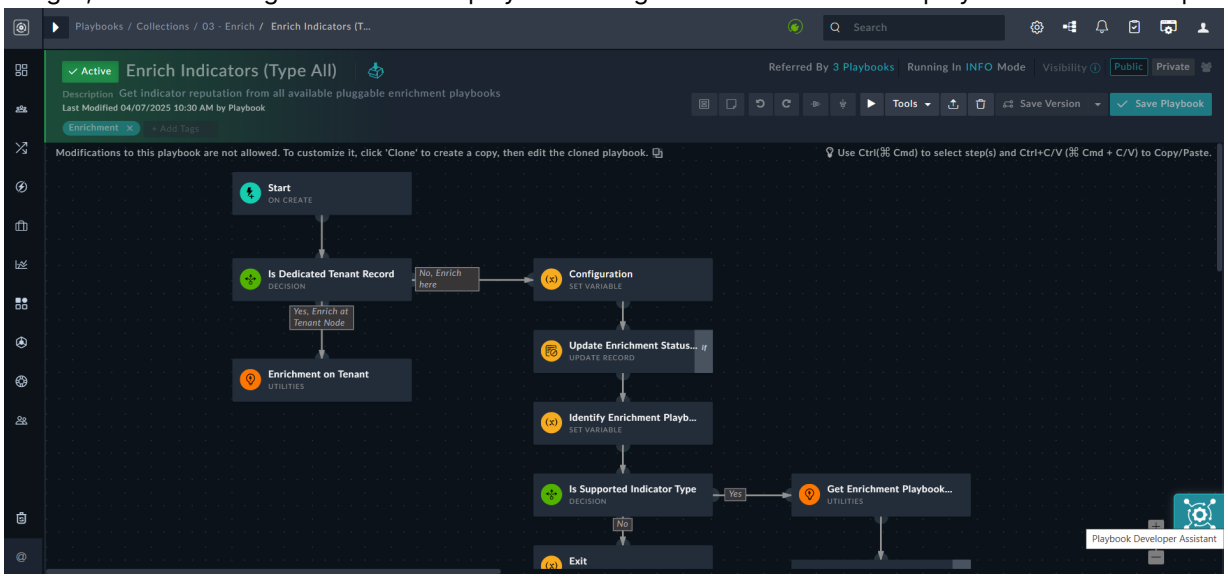


Clicking the **<number> Playbook** link opens the `Referred By` dialog, which lists all playbooks that referencing the current one, along with their playbook collection and status:

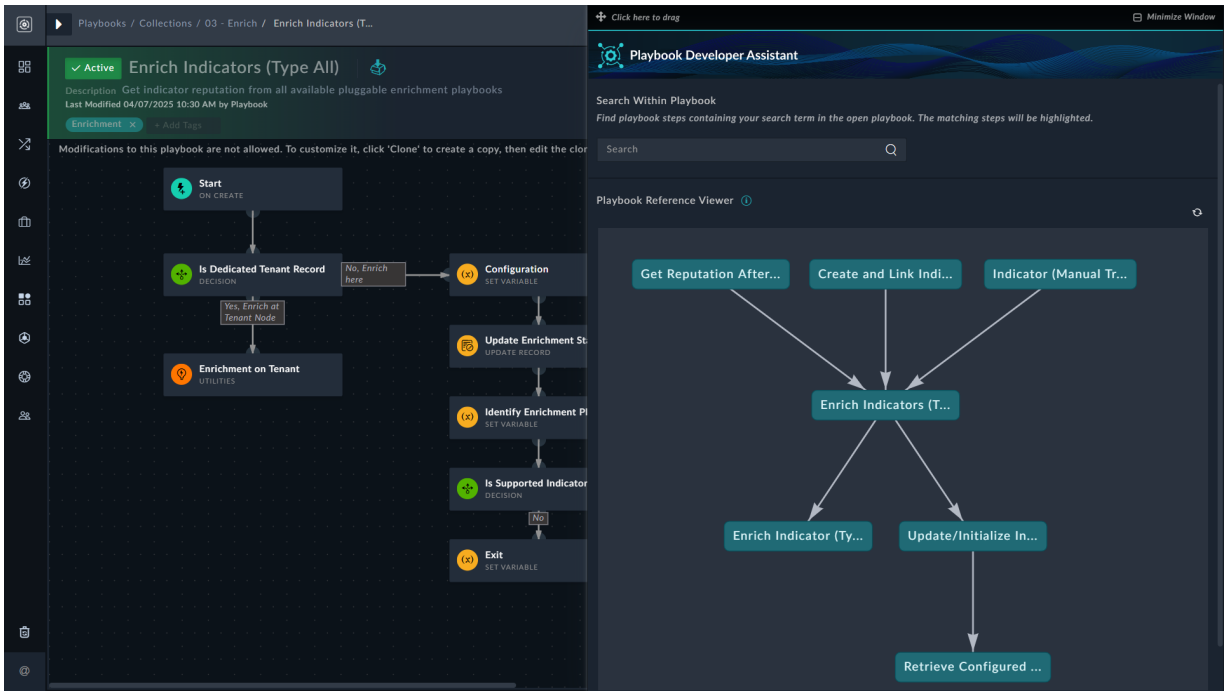


To view the details of a specific playbook, click **View Contents**. This will open the parent playbook in a new window.

- Use the **Playbook Developer Assistant** icon, made available from the 'Playbook Developer Assistant' widget, in the lower-right corner of the playbook designer to view the current playbook's relationships:



Clicking the **Playbook Developer Assistant** icon displays a visual tree structure showing both referencing and referenced playbooks, helping users navigate interconnected and nested playbooks.



For details, see the Playbook Developer Assistant widget documentation.

Adding Blocks and Notes

Adding Blocks

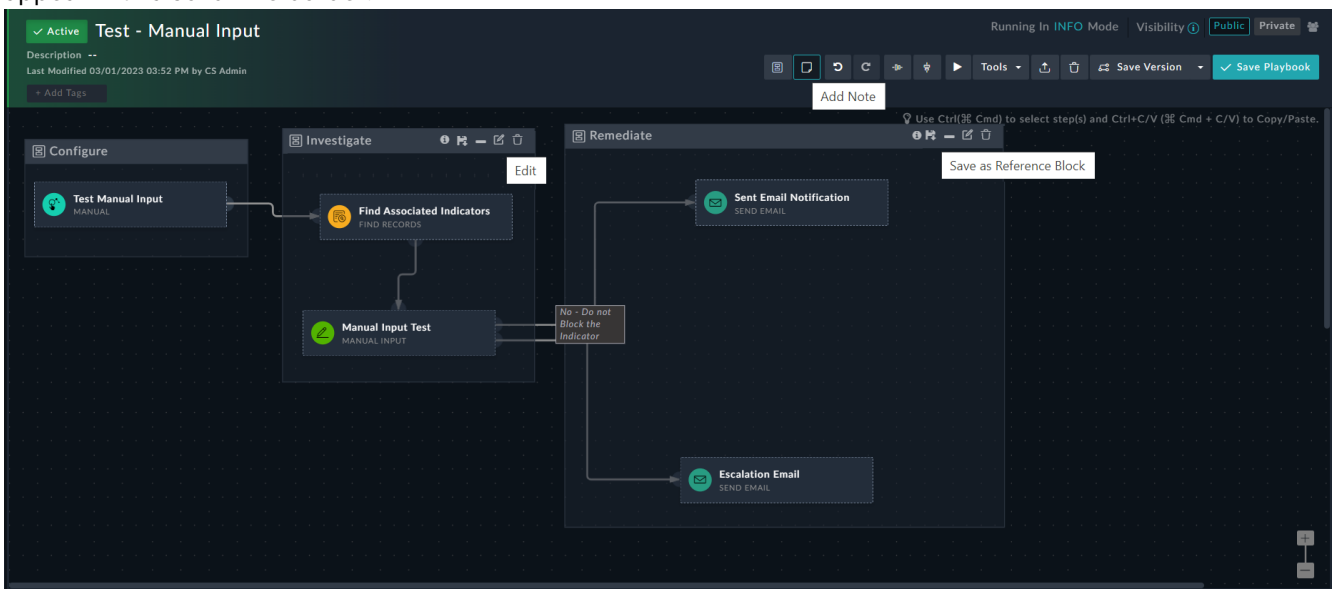
You can add blocks containing multiple playbook steps that achieve a logical group context in a logical workflow diagram. For example, the start or trigger step can form a trigger group, i.e., the 'Configure' group of the playbook; similarly, finding indicators associated with the record, getting the reputation of those indicators using a threat intelligence tool, and then presenting that reputation to users for a decision can form the 'Investigate' group of the playbook, and then performing appropriate actions based on the user's decision can form the 'Remediate' group of the playbook.

To add a block, open a playbook in the playbook designer, click the **Create Block** button, and then draw the box of the preferred size in the designer; that will create the block of that size and release the selection. Playbook steps placed by dragging and dropping the steps within the drawn box are added, by default, to the block. You can also drag and drop the block over the playbook steps that should be part of the block and then release the selection. When you release the selection, the `Add Block` dialog is displayed. In the `Add Block` dialog, enter the title of the block and, optionally, a description of the block, and click **Add**.

To associate a playbook step with a block, drag and drop the playbook step completely into the block. To remove a playbook step from a block, drag and drop the playbook step completely out of the block.

A block can be resized as per your requirement; however, when you are resizing the block, it is possible that you might add additional playbook steps to the block or remove steps from the block. Therefore, you must appropriately resize the block to ensure that playbook steps (appear with dotted lines) that are part of the block are completely within the block.

Steps that are part of a block appear with a dotted-line border, whereas steps that are not part of any block appear with a solid-line border:

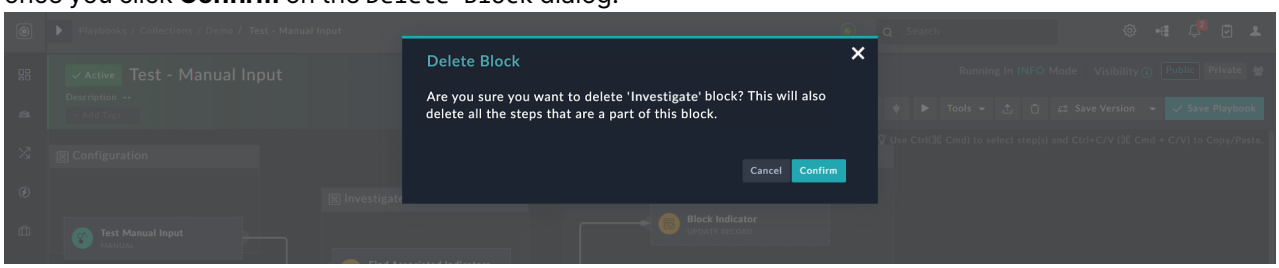


You cannot have nested blocks, i.e., you cannot place one block within another block, and also a note cannot be part of a block. Playbook steps can be part of only a single block and not multiple blocks.

You can select the complete block and copy-paste the complete block, or you can select individual steps within the block and copy-paste them to your required destination playbook.

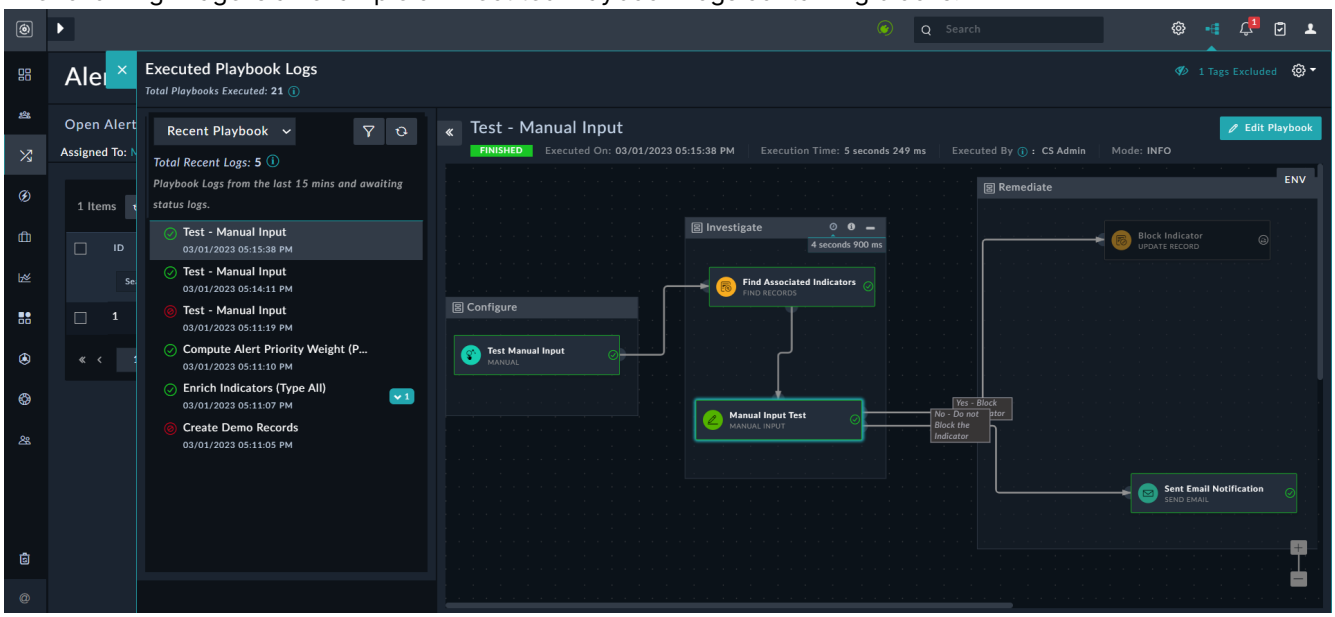
A block contains the following options:

- **Info**: Displays additional information about the block, i.e., the description that you have added for the block.
- **Save as Reference Block**: Saves the block as a 'Reference Block'. For more information, see the Saving a Block as a Reference Block topic.
- **Minimize**/**Maximize**: Reduces or increases the size of the block.
- **Edit**: Opens the `Edit Block` dialog, in which you can edit the name and/or description of the block.
- **Delete**: Opens the `Delete Block` dialog using which you can delete the reference block. To delete a reference block (or global variable) you must be assigned `Read`, `Update`, and `Delete` permissions on the `Playbooks` module. Users with only the `Delete` permission are unable to delete reference blocks (or global variable) since they cannot view the reference blocks or global variables.
  Note that the 'Delete' operation deletes **both** the block and the playbook steps that are part of that block, once you click **Confirm** on the `Delete Block` dialog:

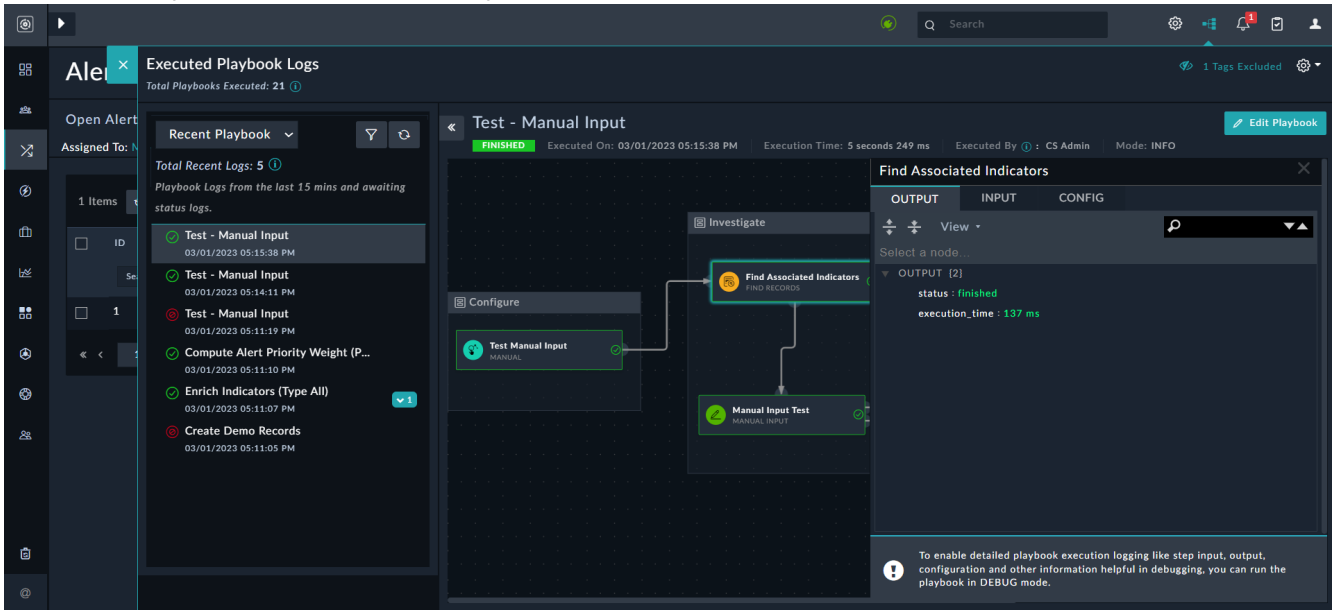
  

  However, if you want to only delete the block and not the steps within the block, you must move the steps out of the block, save the playbook, and then delete the block.

The following image is an example of Executed Playbook Logs containing blocks:



In the Executed Playbook Logs, you can minimize/maximize the blocks and also hover on the information icon to get information about the block. If the description is added in the `Add Block` dialog, then that description is visible when you hover over the information icon. Each block also displays metrics such as the total execution time for each block. For example, the Investigate block in the above image displays 4 seconds 900 ms. The total execution time displayed is the aggregation of the time taken by each step within the block to complete its execution. The time taken for each step, whose status is 'finished', within the block is also displayed in the `execution_time` field of the respective step output. For example, in the following image, the 'Find Associated Indicators' step has taken 137 ms to complete:



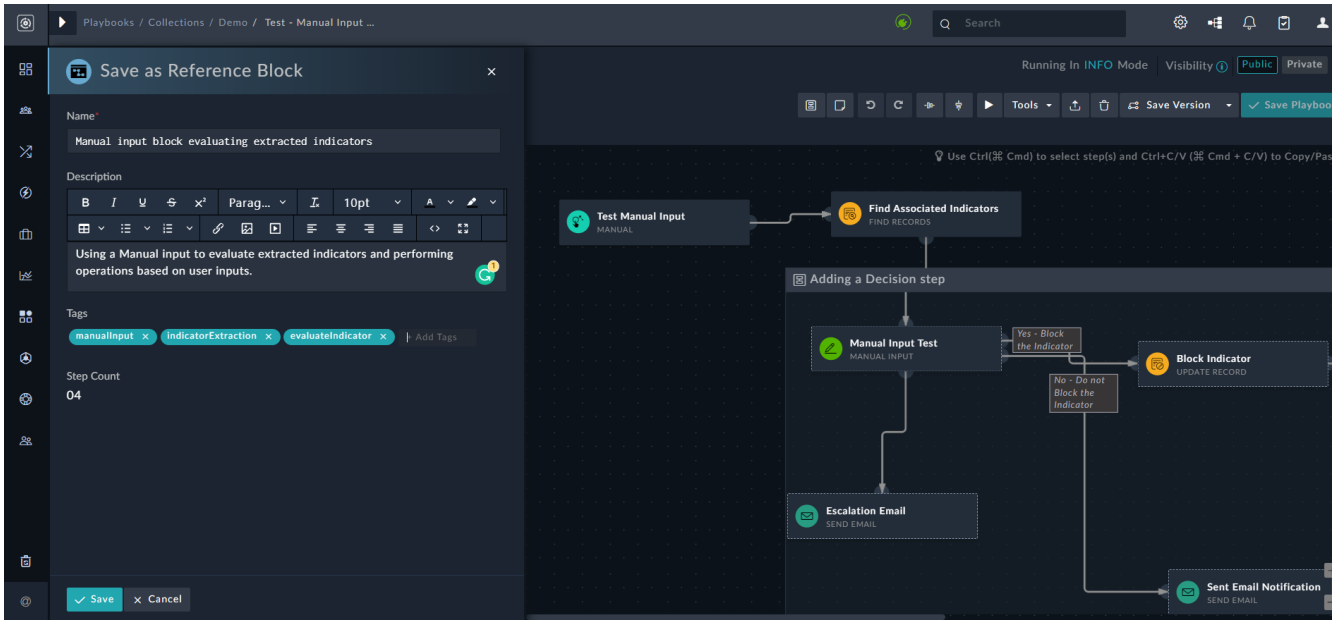
Saving a Block as a Reference Block

To enable the usage of blocks in other playbooks, you can choose to save a block as a 'Reference Block'. From the Block options, click the **Save as Reference Block** icon to display the `Save as Reference Blocks` dialog.

The `Save as Reference Blocks` dialog displays the name and description that you specified while adding the block in the **Name** and **Description** fields, respectively. The name and description of the Reference Block can be distinct from the block's name and description, and therefore, you can choose to update both the name and description of the block in the `Save as Reference Blocks` dialog.

The name specified for Reference Blocks must be unique, with a minimum of 3 characters and a maximum of 140 characters.

Additionally, you can add tags associated with the reference block in the **Tags** field. Click **Save** to save the block as a Reference Block and add the same to the Reference Blocks list (**Tools** > **Reference Blocks**):

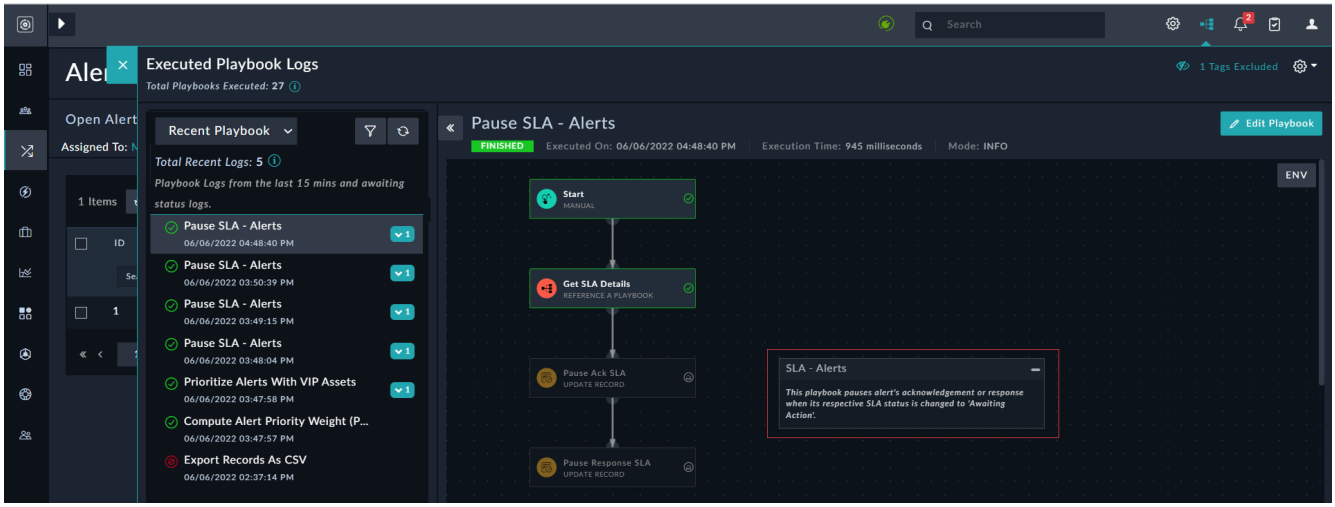


If, in the future, there is a need to edit a Reference Block, it must be edited using **Tools** > **Reference Blocks**.

Adding Notes

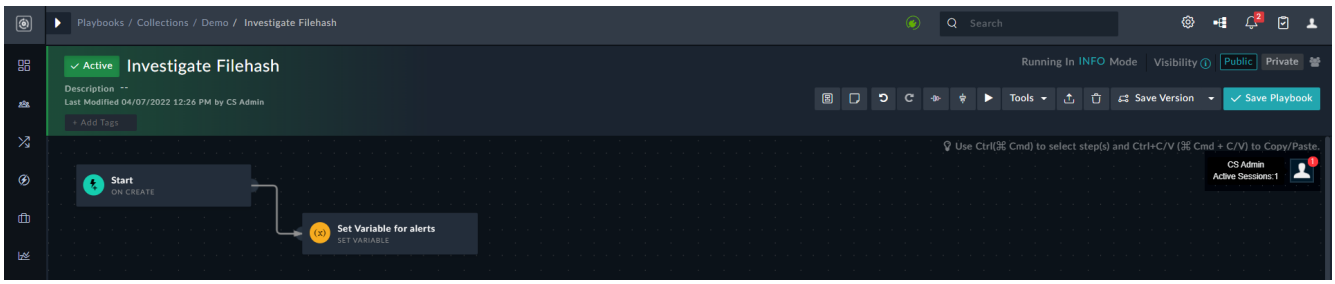
You can also add a note in the playbook designer to provide more information about the playbook or to explain the playbook steps in greater detail. To add a note, click the **Add Note** button. In the Add Note dialog, enter the title for the note in the **Name** field and add the note information in the **Description** field. If you do not want to display the notes that are added to the playbooks in the 'Executed Playbooks Log', select the **Hide Note in Executed Playbook Log** option and click **Add**. If you want to display the notes that are added to the playbooks in the 'Executed Playbook Logs', then clear the **Hide Note in Executed Playbook Log** option. The following

image is an example of an 'Executed Playbook Logs' containing a note:



Live User Implementation in Designer

The playbook designer implements Live Users, which means that the playbook designer displays users who are also currently working on the same playbook. Therefore, when you open a playbook and if there are other users who are working on the same playbook apart from you, then the playbook designer will display the users working on the playbook, as well as the number of sessions that are active for each user. Live Users also notifies users that are working on the same playbook, if any other user or session has saved modifications to the playbook, so that the user can refresh the playbook before working on the same, thereby ensuring that users work on the latest version of the playbook. Users can also save versions of their current modified state of the playbook, thereby providing users with the ability to merge their changes.



Live Users has the following benefits:

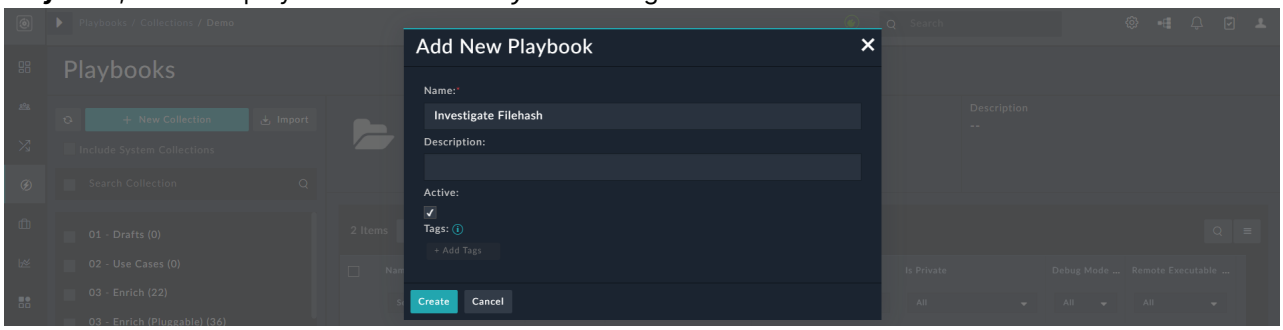
- Users are notified of other users or sessions that are active on the same playbook.
- Users work on the latest version of the playbook, and they do not lose their updates made to the playbook.

Creating New Playbooks

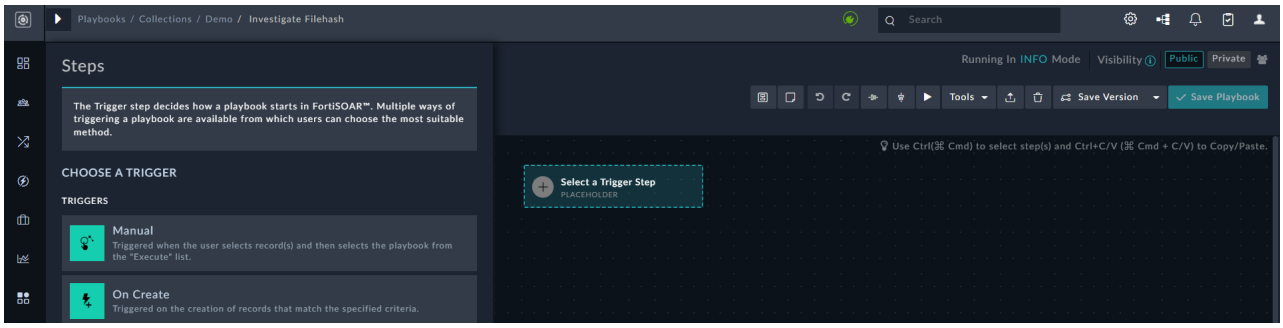
You can use FortiAI to generate customized playbooks or playbook steps based on your specific scenarios, helping you automate and optimize your use cases. For detailed instructions on creating playbooks, see the FortiAI document.

Steps for manually creating playbooks are as follows:

1. Click **Automation** > **Playbooks** in the left navigation bar.
2. On the `Playbook Collections` page, click **New Collection** to define a new playbook collection in which to save the playbook you want to create, or, click an existing playbook collection and add the new playbook in that collection.
   Note: You cannot add a playbook directly on the `Playbook Collections` page, you require to add playbooks to a playbook collection.
3. In the `Add New Playbook Collection` dialog, add the name of the collection in the **Name** field and optionally in the **Description** field, add the description for the playbook collection.
   You can optionally change the icon that represents the playbook collection, by clicking **Change Image** and dragging and dropping your icon to the Upload an Image dialog, or browsing to the icon on your system, selecting the icon and then clicking **Save Image**.
   You can optionally also add keywords in the **Tags** field that you can use to reference the playbook collection and making it easier to search and filter playbook collections and playbooks. You can add special characters and spaces in tags; however, the following special characters are not supported in tags: `'`, `,`, `"`, `#`, `?`, and `/`.
   Click **Create** to create the new playbook collection.
4. To add a playbook, click the collection in which you want to create the new playbook, and then click **Add Playbook**, which displays the `Add New Playbook` dialog:

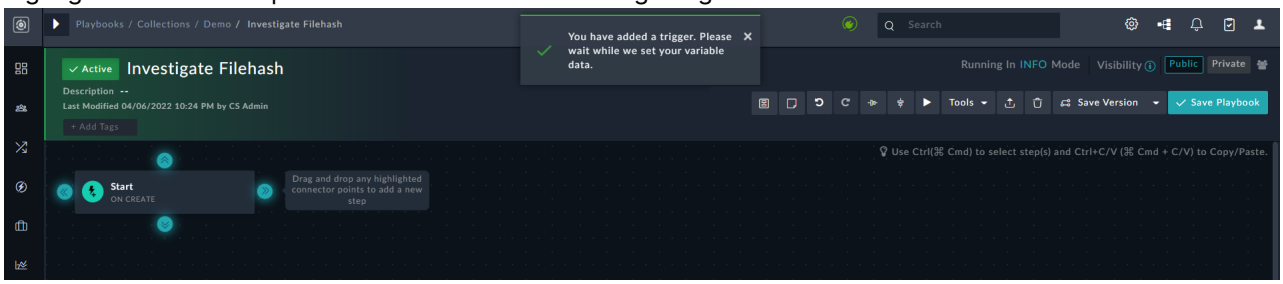


5. In the `Add New Playbook` dialog, add the name of the playbook in the **Name** field and optionally in the **Tags** field, add keywords that you can use to reference the playbook, making it easier to search and filter playbooks. You can optionally in the **Description** field, add the description for the playbook.
   Important: Playbook names must be unique within a collection.
   The **Active** checkbox sets the state of the playbook as `Active`, or `Inactive`. By default, the **Active** checkbox is selected, i.e., new playbooks are created in the `Active` state.
   Click **Create** to add the new playbook.
   Note: The logging level of new playbooks are set as `INFO`, if you want detailed logging for the playbook, then you can open the playbook and set its mode to `DEBUG`.
6. FortiSOAR displays the `Playbook Designer` for the newly added Playbook, with a placeholder trigger step and the name you have specified being displayed in the **Name** field at the top of the Designer. Now, you must select a playbook trigger from the `Triggers` section and enter the necessary variables for the selected trigger, and then click **Save**.



For information on the various triggers, see the Playbook Triggers chapter.
Note: Specific conditions that the playbook should meet before continuing can be called out by creating a `Decision Step` immediately after the trigger. While the playbook will still execute, the decision step (s) determines if the playbook continues through the following steps or is considered finished.

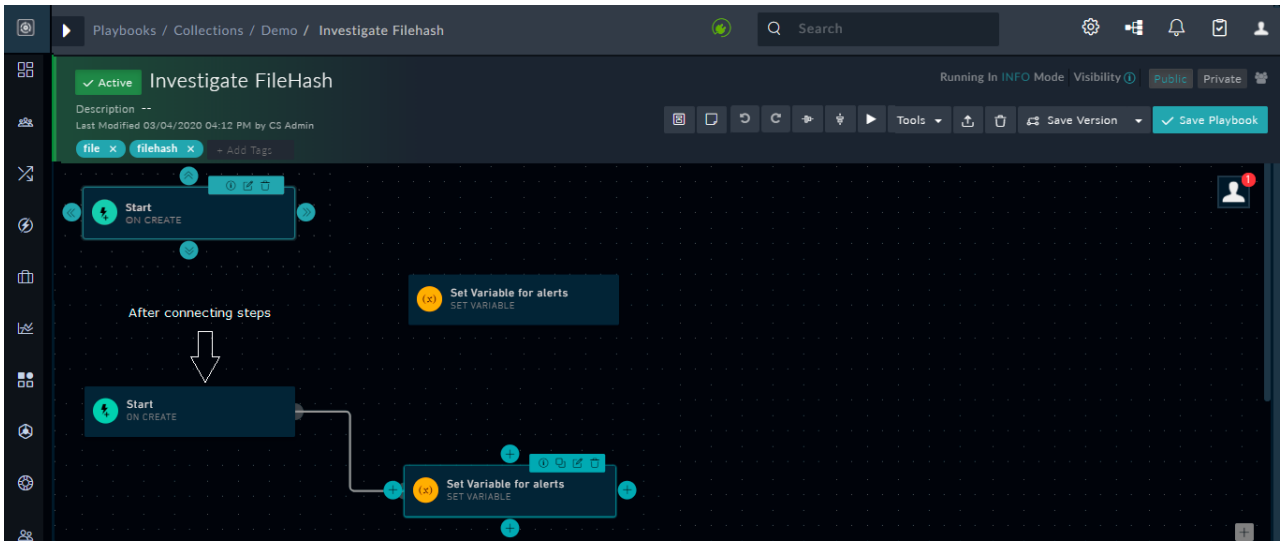
7. Add playbook steps.
Once you've selected a trigger, FortiSOAR displays the trigger step in the `Playbook Designer` with highlighted connector points as shown in the following image:



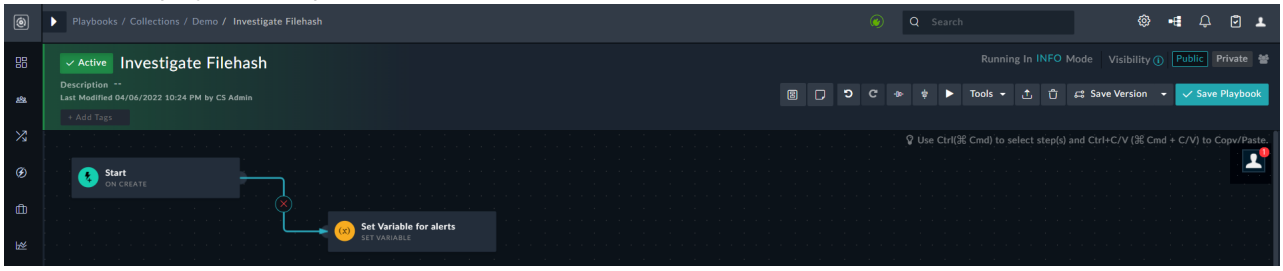
Drag-and-drop a connector point to connect to another playbook step. FortiSOAR adds a placeholder step on the playbook designer page and opens the `Steps` tab which displays all the available playbook steps, select the playbook step that you need next, add the **Step Name** and the required variables and click **Save**.

Similarly, you can add further steps and create the desired flow for the playbook. A playbook ends when there are no additional steps to run. For more information on steps, see the the Playbook Triggers chapter.

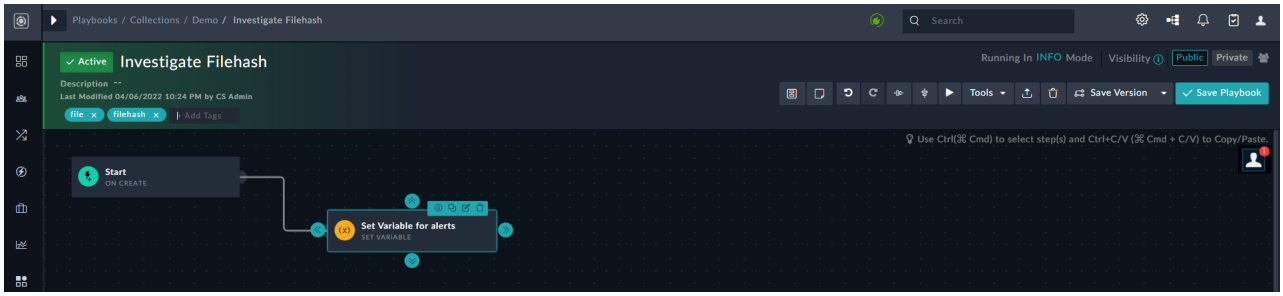
8. Connect playbook steps or remove a connection between steps.
It is straightforward to connect playbook steps as well as to remove the connection between playbook steps.
To connect a playbook step, use the connection points that appear when you hover on a Playbook step. Select a connection point and drag and drop the arrow connector on the step you want to connect.



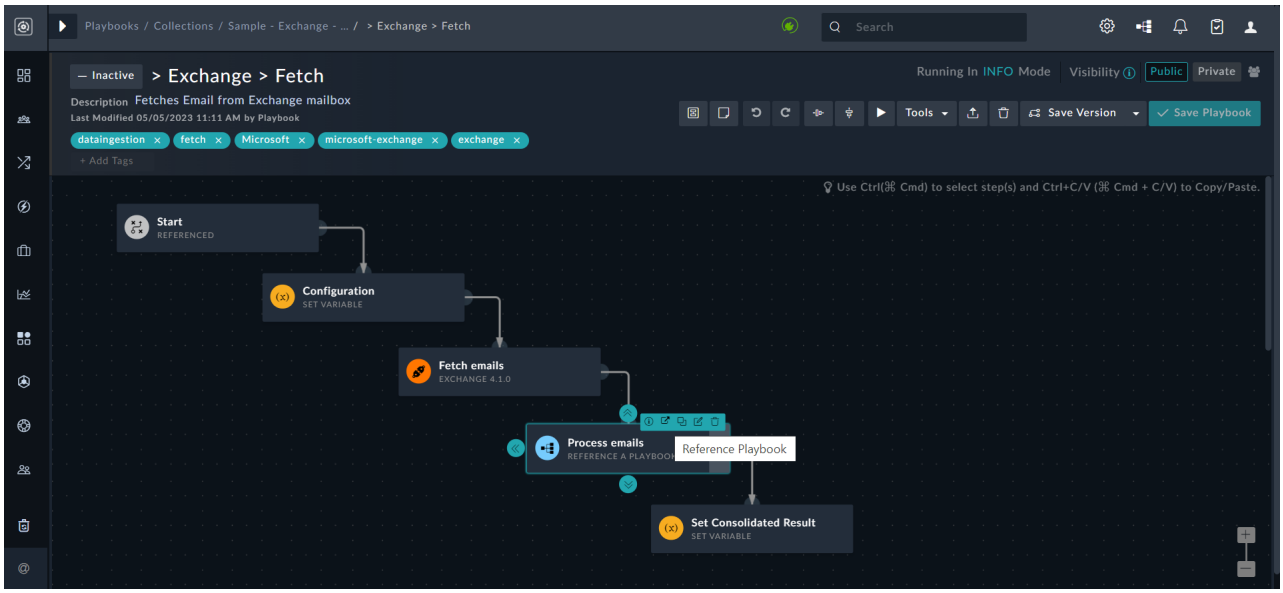
To remove a connection between playbook steps, hover on the arrow connector between the steps, which then displays a cross (**X**) red color. Clicking **X** displays a `Confirm` dialog, click **OK** to remove the link between the playbook steps.



9. (Optional) To edit or remove an existing playbook step double-click on the step to reopen it and then you can edit the step or delete the step entirely by clicking **Delete Step**.
Playbook steps include icons for the **Info**, **Reference Playbook** (only for the Reference A Playbook step), **Edit**, **Clone**, and **Delete** actions enabling you to perform these actions in the step itself using the respective icons.



Clicking the **Info** icon displays additional information, if available, about the step.
Clicking the **Reference Playbook** icon (applicable only to the Reference A Playbook step) opens a referenced playbook in a new window. By allowing users the option to view the contents of the referenced playbook in a new window while viewing the referencing playbook's whole flow on the same canvas in the playbook designer, improves the user experience when creating playbooks. However, note that the **Reference Playbook** will open in a new window only when users select the reference playbook from the **Playbook Reference** drop-down list in the 'Reference A Playbook' step and not when user refer to a playbook using Dynamic Values (jinja).

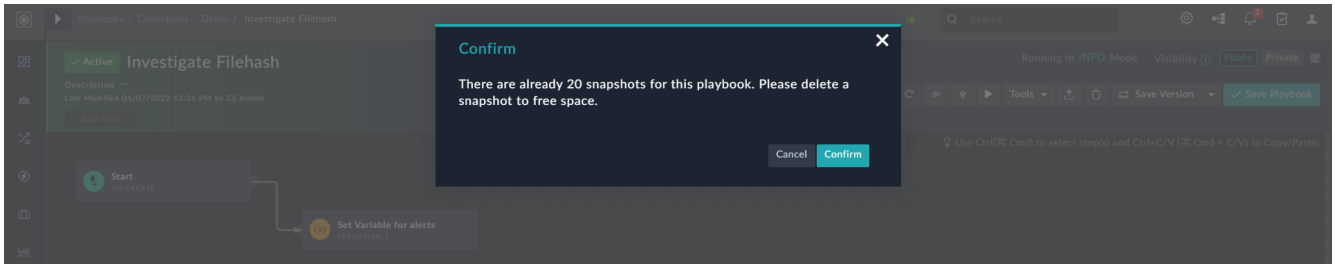


Clicking the **Clone** icon creates a copy of the current step and opens the step with the name as `Copy of %Step Name%`. All the properties of the current step are copied to the cloned step. You can edit the properties of the cloned step as required and then save the step.
Clicking the **Edit** icon reopens the step, and you can edit the properties of the step and then save the step.
Clicking the **Delete** icon deletes the step entirely.

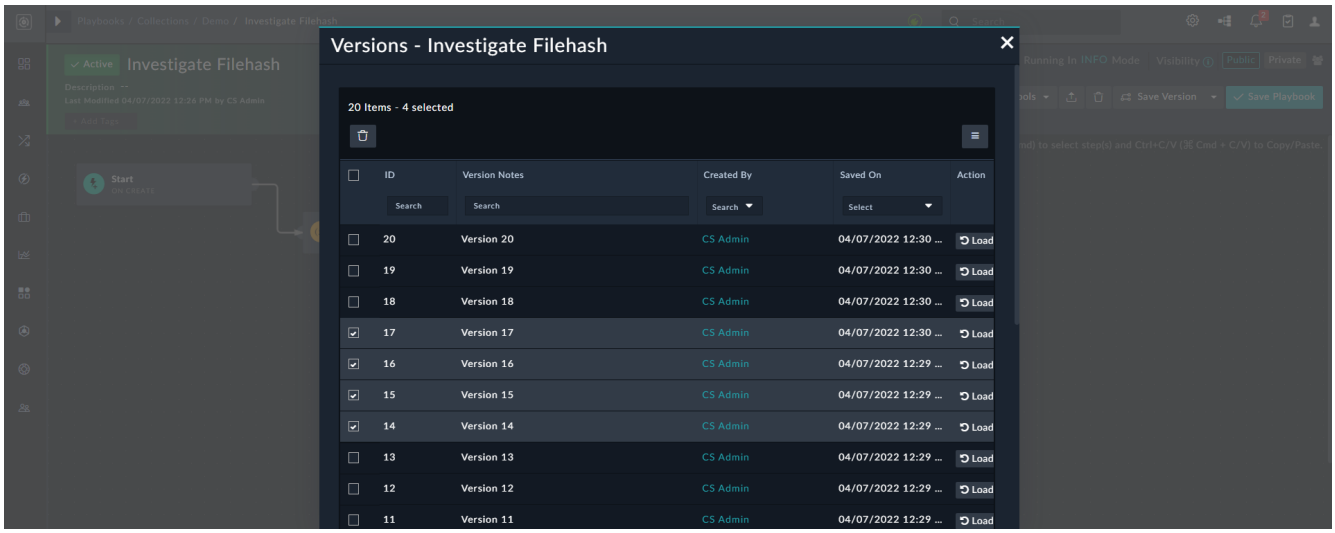
Saving Versions of Playbooks

You can save versions of a playbook that you are creating or updating. Using versioning, you can save multiple versions of the same playbook. You can also revert your current playbook to a particular version, making working in playbooks more effective.

The maximum number of versions that can be taken, across all users working on a playbook is 20. If you or other users try to take more than 20 snapshots, a confirm dialog is displayed that prompts you to delete a version so that you can free space and save a new version, as shown in the following image:



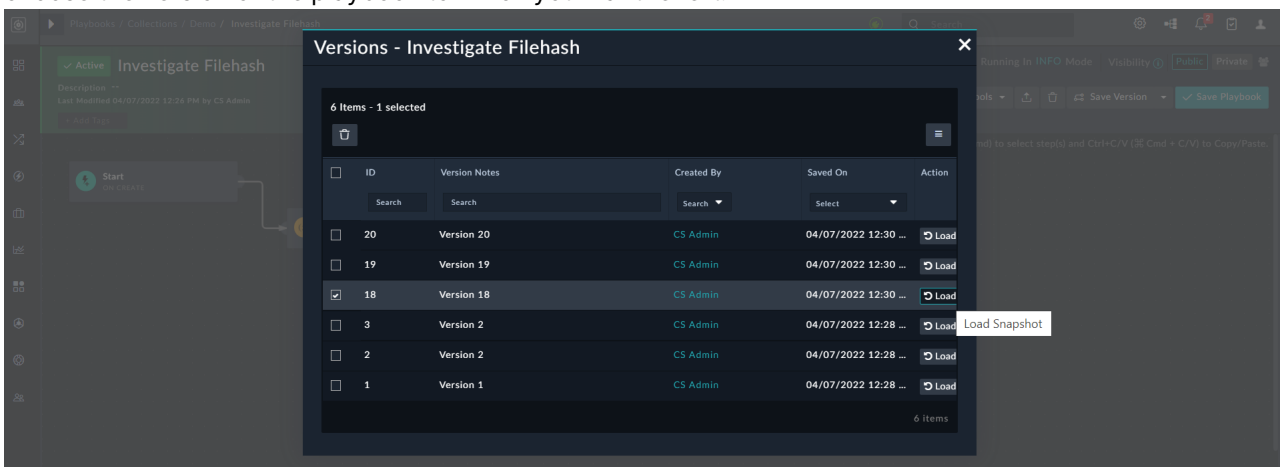
When you click **Confirm**, the `Versions - <Name of Playbook>` dialog is displayed. You can now choose the version(s) that you want to delete, click the **Delete** icon, and then click **Confirm** on the confirmation dialog and close the `Versions - <Name of Playbook>` dialog. This frees up space and you can now save a new version.



Using the `Versions - <Name of Playbook>` dialog, you can search for versions based on the notes you have added, and also filter versions by the Created By (user who has created the version), Saved On (time the version was saved) and action performed.

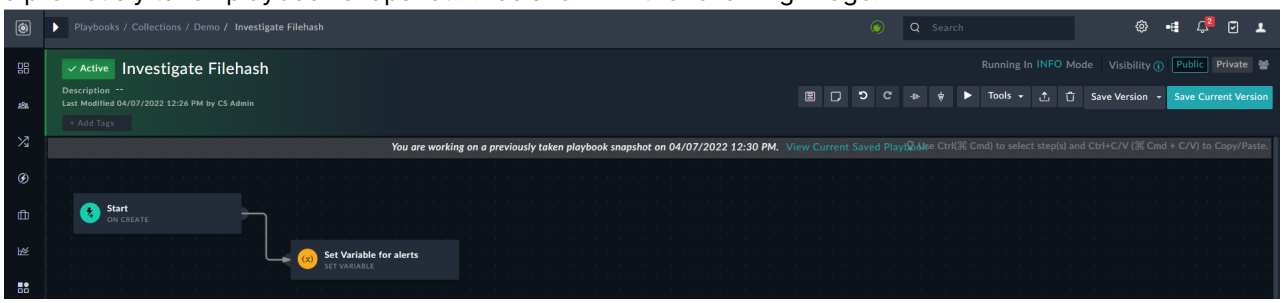
To take a snapshot and revert the playbook to a particular snapshot do the following:

1. In the playbook in which you are working in the playbook designer, click **Save Version**.
2. In the `Save Version` dialog, add a note that you want to associate with the version and click **Save Version**. It is recommended that you add meaningful notes for versions as these names will help you in identifying the snapshots when you want to revert to a particular version.
3. To revert a version, click **Save Version** and then either click **Revert to Last Saved** or click **View Saved Versions**. Clicking **Revert to Last Saved** reverts the playbook to the last saved version of the playbook Clicking **View Saved Versions** displays the `Versions - <Name of Playbook>` dialog that allows you to choose the version of the playbook to which you want revert:



In the `Versions - <Name of Playbook>` dialog, in the version row to which you want to revert, click **Load**. Once you click **Load**, that snapshot is loaded in the playbook designer, with a message: "You are working on

a previously taken playbook snapshot...." as shown in the following image:

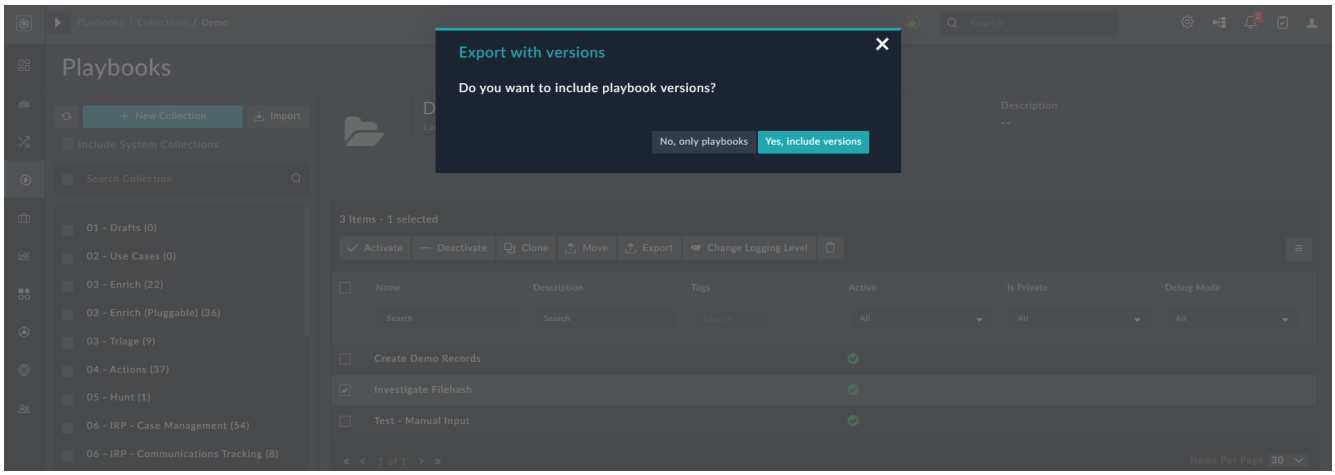


4. You can choose to view the playbook that is currently saved, by clicking the **View Current Saved Playbook** link, or you can click **Save Current Version** to make this version the current saved version of the playbook and continue to work on the playbook.

Exporting Versions of Playbooks

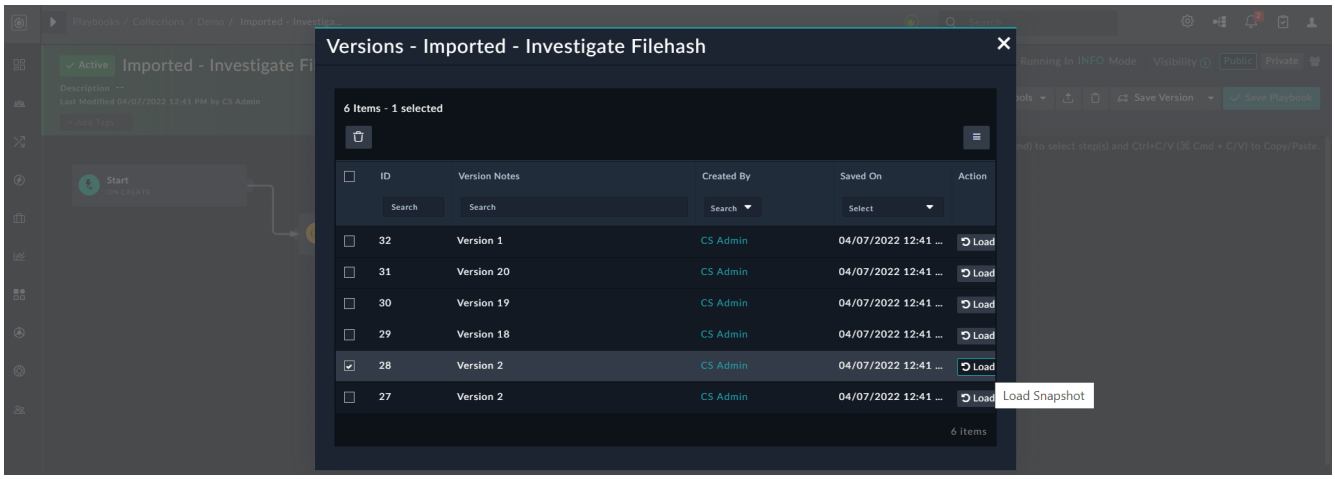
You can choose to export playbook collections or playbooks with saved versions of the playbooks. This is extremely useful while developing playbooks, especially if you erroneously delete a step in the playbook or you want to go back to the previous state of the playbook. Retaining the versions of playbooks while exporting playbooks enables you to load a snapshot of a previously saved version of the playbook into an imported playbook

You can choose to export playbook collections or playbooks with saved versions of the playbooks as shown in the following image:

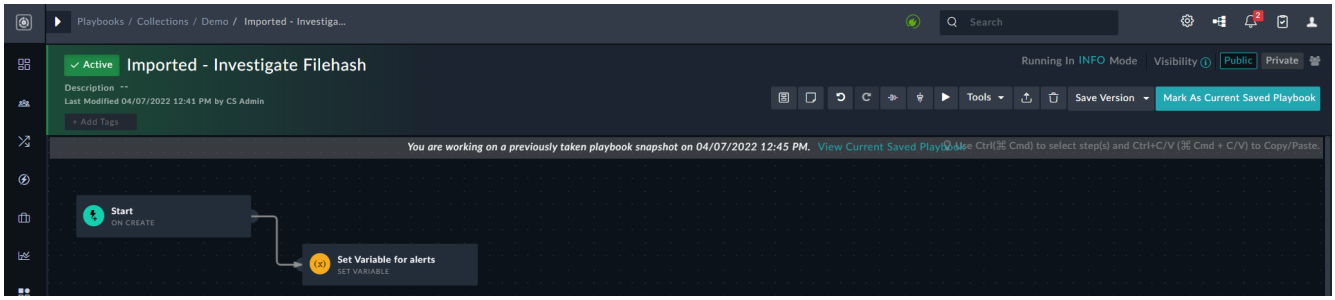


Clicking **Yes, include versions** on the above dialog will export playbooks or playbook collections with the saved versions of the playbook.

You can then import the playbook and then open that playbook in the playbook designer, you can see the previously saved versions of the playbook by clicking **Save Version** > **View Saved Versions**. This opens the `Versions` dialog as shown in the following image:

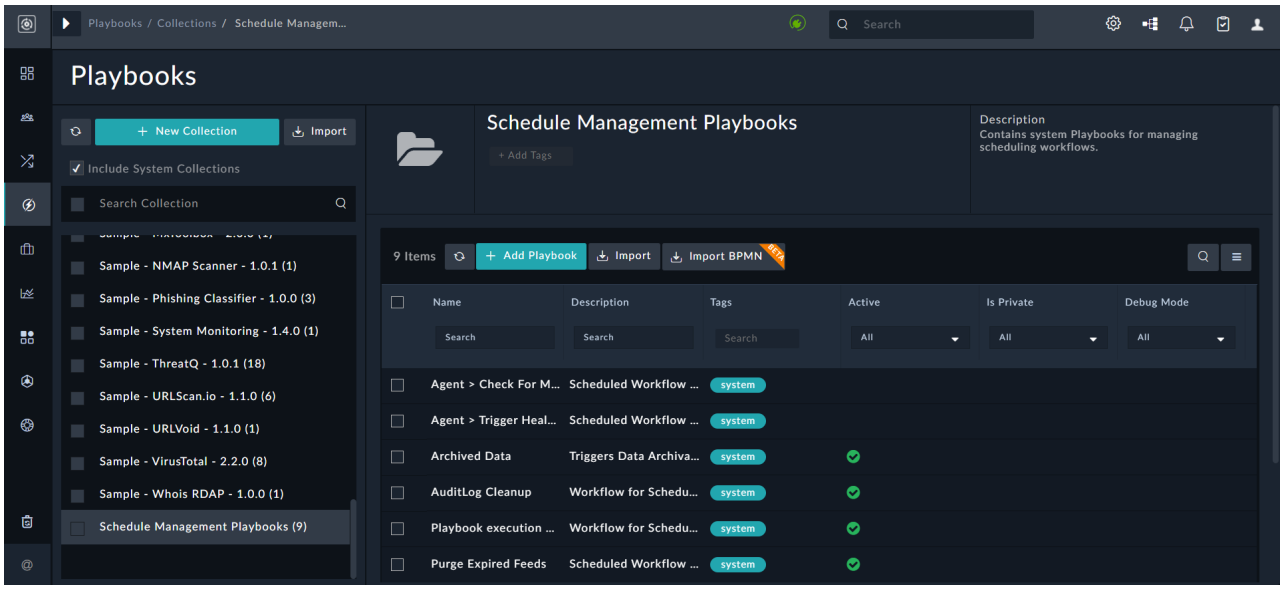


You can load a snapshot of a previously saved version of playbook in the `Versions` dialog by selecting the snapshot that you want to load in the playbook designer and clicking **Load**. This will display a message such as "You are working.....playbook snapshot...." as shown in the following image:

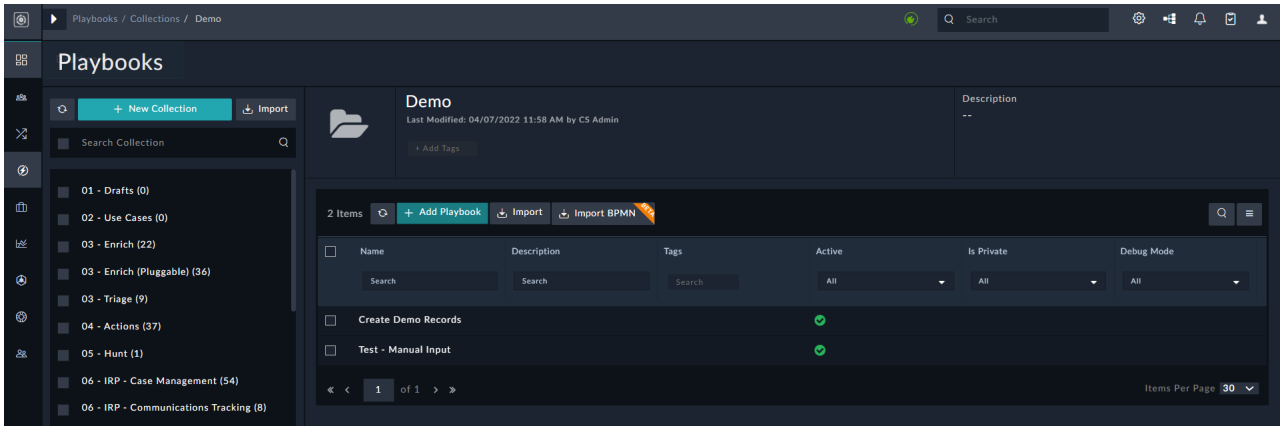


Editing Playbooks (Solution Packs, System Playbooks)

1. Click **Automation** > **Playbooks** in the left navigation bar.
2. On the `Playbooks` page, you can view playbook collections and their associated playbooks.
   Users with a minimum of `Update` permissions on the `Security` module can view all system playbook collections by clicking the **Include System Collections** check box. Clicking the **Include System Collections** checkbox displays the hidden playbook collections, both system fixtures as well as collections that contain data ingestion playbooks created by the data ingestion wizard, allowing you to view all the hidden playbook collections at once. For example, the Schedule Management playbook collection in the following image is a system playbook collection:



If you are a user without `Security Update` permissions, the **Include System Collections** checkbox will not be visible:



3. For all playbook collections, you can perform the following actions without having to select any playbook collection:
   - **Search**: Use the **Search Collection** field to search for playbook collections.
   - **Import**: You can import a playbook collection into FortiSOAR if it is in the appropriate JSON format. To import a playbook collection into FortiSOAR, on the `Playbook Collections` page, click **Import**. On the `Import Collections` dialog, drag and drop the JSON file, OR click the **Import** icon and browse to the JSON file to import the playbook collection into FortiSOAR, and then click **Import**.
     Note: The name of the playbook collection being imported must be unique; otherwise you will get a conflict while importing the playbook collection. However, if you want to replace an existing playbook collection, then click the **Replace existing playbook collection** checkbox.
     FortiSOAR also displays the list of global variables that would be imported along with the playbook collections or playbooks on the `Import Collections` dialog. These are the global variables that were part of the playbook that you exported. You can review the imported global variables, and choose to modify them as per your requirements.
     If the JSON format is incorrect, FortiSOAR displays an error message and does not import the file.
     If the JSON format is correct, FortiSOAR imports the playbook collection and displays a success message.

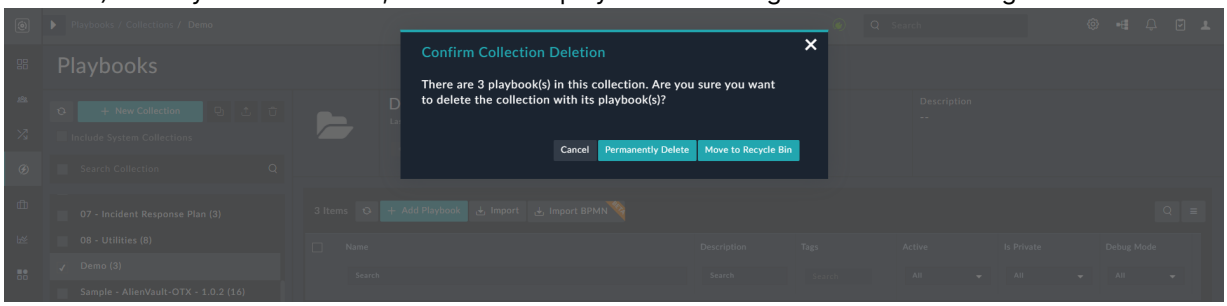
Note: Any tags associated with the playbook collection are upserted into the system when you import a playbook collection.

4. For playbook collections, you can perform the following operations on selected playbook collections:
   - **Clone**: To clone playbook collections, select the desired collections and click **Clone**.
     Starting with release 7.6.1, playbooks included in solution packs will no longer be overwritten during upgrades. This ensures that any custom changes made to your playbooks are retained, streamlining the upgrade process.
     NOTE: The behavior of playbooks that were part of connectors has not changed. It is still recommended to clone playbooks before editing them, as sample playbooks from connectors will be overwritten during upgrades.
     IMPORTANT: In systems upgraded to release 7.6.1, all imported playbooks from solution packs will be editable by default. To preserve any custom changes made to playbooks, clone the playbooks before upgrading the solution packs. In contrast, in new installations of release 7.6.1, most playbooks will be non-editable, except for those that require user configuration to work, for example, playbooks that block IP addresses are editable.
     Additionally note the *behavior of playbooks during solution pack upgrades*:
     - **Non-Editable Playbooks**: Playbooks that are part of a solution pack and are non-editable will be overwritten during upgrades. An example of a non-editable playbook is the '*Reset Enrichment Global Variables (03 - Enrich)*' playbook from the SOAR Framework solution pack.
       NOTE: Post- FortiSOAR release 7.6.1, clone any playbooks with custom changes that are part of the solution pack, before upgrading the solution pack to ensure the modifications are preserved.
     - **Editable Playbooks**: Playbooks that are part of a solution pack and are editable will have new changes added to a 'Base' version during upgrades. For example, the '*Incident - Get Running Process (04 - Actions)*' from the SOAR Framework solution pack is editable. For more information, see the Editing playbooks that are part of Solution Packs topic.

     Prior to release 7.6.1, it was recommended to clone playbooks before editing them when using sample playbooks from connectors and solution packs to prevent losing customizations during upgrades.
     Cloning the playbook collection clones all the playbooks within that collection. You can select more than one playbook collection to clone at a time. clones the playbook collection and saves it with the name `%Playbook Collection% copy-1(No of playbooks in collection)`. Note the following for playbook references within playbook collections:
     - If you have cloned a playbook collection that contains playbooks that reference where either one references the other or they reference each other, then the references get automatically updated. For example, consider a playbook collection named 'Demo (5)' containing playbooks named 'Enrich Indicators' and 'Update Alert Severity' where the 'Update Alert Severity' playbook references the 'Enrich Indicators' playbook. Now, if you clone the 'Demo (5)' playbook collection to create the 'Demo copy-1 (5)' collection, any references to the 'Enrich Indicators' playbook in the 'Update Alert Severity' playbook are updated to reference the 'Enrich Indicators' playbook in the 'Demo copy-1 (5)' collection.
     - If you have cloned a playbook collection that contains playbooks that reference playbooks in a different collection, then the references do not get automatically updated.
       However, if you **simultaneously** clone playbook collections containing playbooks where either one references the other or they reference each other, then the references get automatically updated. For example, consider a playbook collection named 'Demo (5)' containing the 'Escalate Alert and Create Incident' playbook that references the 'Update Alert Severity' playbook in the 'Incident Management (3)' playbook collection. Now, if you simultaneously clone both the 'Demo (5)' and 'Incident Management (3)' playbook collections to create the 'Demo copy-1 (5)' and 'Incident Management copy-1 (3)' collections, any references to the 'Update Alert Severity' playbook in the

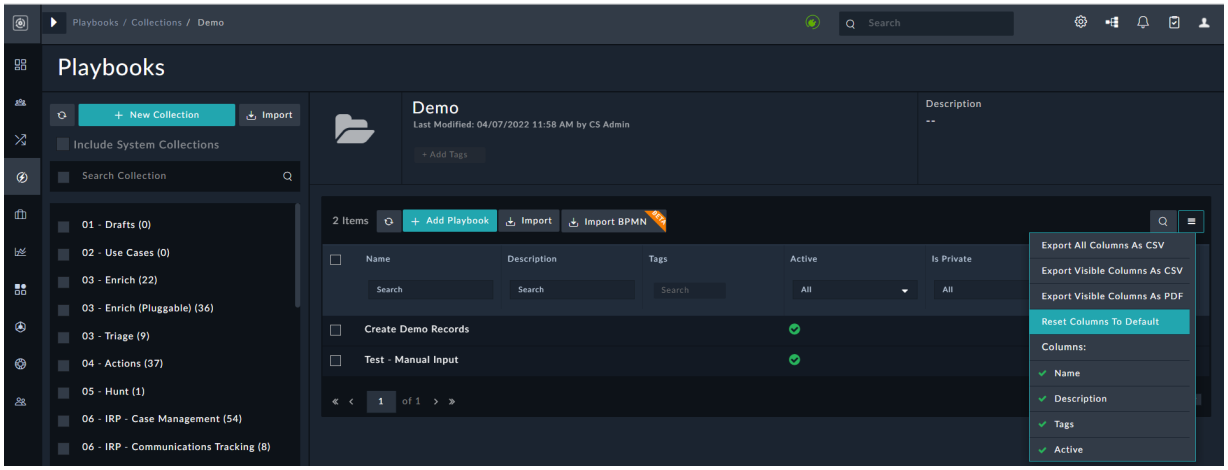
'Escalate Alert and Create Incident' playbook are updated to reference the 'Update Alert Severity' playbook in the 'Incident Management copy-1 (3)' collection.

- **Export**: To export a particular playbook collection, select the playbook collection and click **Export**. Any tags associated with a playbook collection are exported when you export a playbook collection. FortiSOAR exports the playbook collection in JSON format.
- **Delete**: To delete a playbook collection, select the playbook collection and click **Delete**. Users with `Delete` permissions on the `Playbooks` module can delete playbook collections. You can choose to permanently delete the playbook collection or move the playbook collection to the Recycle Bin (soft deletion). Once you click **Delete**, FortiSOAR displays the following confirmation dialog:



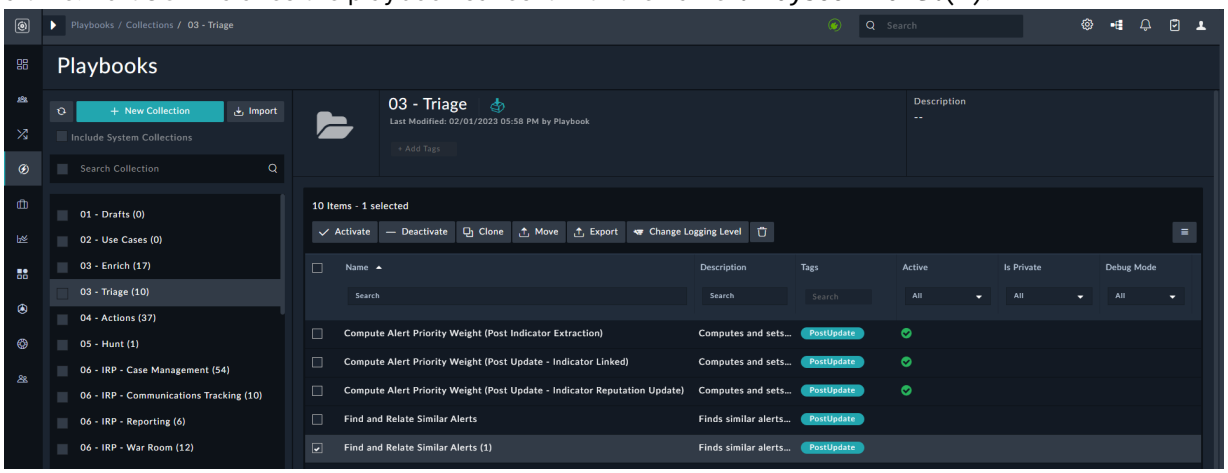
On the confirmation dialog, select your deletion preference based on which the playbook collection is either permanently deleted or moved to the recycle bin. Clicking **Move to Recycle Bin** moves all the playbooks in that collection to the recycle bin. Similarly, when you restore any playbook from a playbook collection, the collection containing those playbooks is also restored. For example, if you have a 'Demo' collection containing 3 playbooks, A, B, and C that you move to the recycle bin, and then restore B from the recycle bin, the Demo collection containing the B playbook gets restored. For more information on the 'Recycle Bin', see the Recovering Deleted Module Records and Workflows topic in the "Best Practices Guide."

5. On the `<Playbooks Listing>` page, you can perform the following operations without selecting a particular playbook:
   - **Add Playbook**: To add a new playbook to the collection, click **Add Playbook**, which opens the `Add New Playbook` dialog. In the `Add New Playbook` dialog, enter the required details that are mentioned in the Creating New Playbooks topic.
   - **Import** - To import a playbook, click **Import Playbook**. The playbook must be in an appropriate JSON format. Any tags associated with the playbook are upserted into the system when you import a playbook.
   - **Import BPMN** - To import BPMN Shareable Workflows and convert them to FortiSOAR playbooks, click **Import BPMN**. For details, see the Importing the BPMN Shareable Workflows as FortiSOAR Playbooks topic.
   - **Search**: Click the Search icon to search for playbooks in the collection.
   - **Other Actions**: Click the **More Options** icon ( ) to export records from the playbooks listing view in the csv or pdf format.



You can also reset the playbook record fields to the default fields specified for the playbook module, click the **Reset Columns To Default** option. You can include the **Created By**, **Created On**, **Modified On**, and **Modified By** fields in a playbook record for tracking purposes.

6. On the `<Playbooks Listing>` page, you might see a message such as "`The count that you see on the playbook collection and the playbooks that you see.....`". This message is shown since RBAC is enforced on playbooks, and this means that you can only see a listing of those playbooks for which you (your team) are the owner, i.e., you cannot view ' Private' playbooks owned by teams to which you are not assigned.
   On the `<Playbooks Listing>` page, you can perform the following operations on selected playbooks:
   - **Search**: To search for a playbook you can type keywords in the **Search** textbox.
   - **Activate**: To mark playbooks as 'Active', select the playbooks on the `<Playbooks Listing>` page and click **Activate**.
   - **Deactivate**: To mark playbooks as 'Inactive', select the playbooks on the `<Playbooks Listing>` page and click **Deactivate**.
   - **Clone**: To clone playbooks, select playbooks on the `<Playbooks Listing>` page and click **Clone**. You might clone playbooks if you want to reuse the playbook as a starting point for a new playbook. Cloning the playbook clones every step within the playbook. You can select more than one playbook to clone at a time. FortiSOAR clones the playbook saves it with the name `%Playbook Name%(1)`:



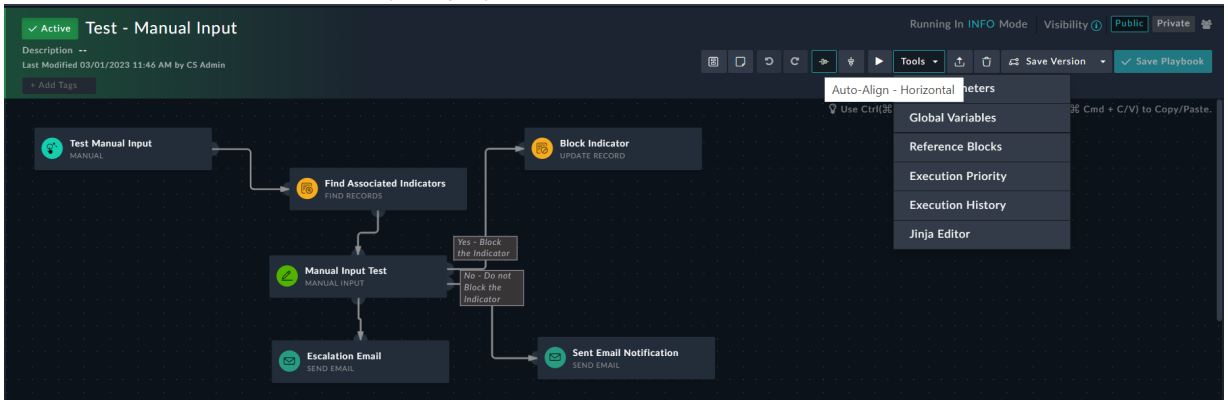
Note the following for playbook references within playbooks:

- If you have cloned a playbook that contains a reference to other playbooks, either within the same collection or in different collections, then the references do not get automatically updated. However, if you have **simultaneously clone playbooks in the same collection** where either one

references the other or they both reference each other, then the references get automatically updated. For example, consider a playbook named 'Escalate Alert and Create Incident' playbook that references the 'Update Alert Severity' playbook. Now, if you simultaneously clone both the 'Escalate Alert and Create Incident' and 'Update Alert Severity' playbooks to create the 'Escalate Alert and Create Incident (1)' and 'Update Alert Severity (1)' playbooks, all references in the 'Escalate Alert and Create Incident (1)' playbook are updated to 'Update Alert Severity (1)' playbook.

- **Move**: To move playbooks to another existing collection, select playbooks on the `<Playbooks Listing>` page and click **Move**. FortiSOAR displays the `Move Playbook` dialog that contains the `Move to collection` section. Clicking **Select** in the `Move to collection` section displays the `Collection` dialog. From the `Collection` dialog, select the collection to which you want to move the playbooks and click **Submit**.
- **Export**: To export playbooks, select playbooks on the `<Playbooks Listing>` page and click **Export**. FortiSOAR exports playbooks in JSON format. Any tags associated with playbooks are exported when you export a playbook.
- **Change Logging Levels**: To change the logging level for playbooks, select playbooks on the `<Playbooks Listing>` page and click **Change Logging Levels**, which displays the `Playbook Execution Log Level` dialog. From the **Select Execution Log Level** field, select `DEBUG` or `INFO` as the logging levels for the playbooks, and click **Apply**. For more information on playbook logging levels, see the Setting Logging Levels topic.
- **Delete**: To delete playbooks, select playbooks and click **Delete**. Users with `Delete` permissions on the `Playbooks` module can delete playbooks. You can choose to permanently delete the playbook or move the playbook to the Recycle Bin (soft deletion). Once you click **Delete**, FortiSOAR displays a confirmation dialog, on which you can choose from the following options: **Permanently Delete** or **Move to Recycle Bin**.

7. To edit a playbook, on the `<Playbooks Listing>` page, click the playbook that you want to edit.
   In the `Playbook Designer`, you can configure the following for the playbook:
   - Change the **State** of the playbook by clicking the **Active** box, for example, change the state of the playbook from `Active` to `Inactive`.
   - Change the **Name** of the playbook, by clicking the name box and updating the name.
   - Add or update the **Description** of the playbook, by clicking the description box and updating the same.
   - Add **Tags** to the playbook using the **Add Tag** box, or remove tags from the playbook.
   - Modify the trigger for the playbook, change or add steps or actions to the playbook. For information on the triggers and steps of a playbook, see the Playbook Triggers chapter.
   - Use the **Tools** menu to enhance your playbook:



- To add parameters, use the **Parameters** option.
- To add or delete global variables, use the **Global Variables** option.
  Note: To delete a global variable (or reference block) you must be assigned `Read`, `Update`, and `Delete` permissions on the `Playbooks` module. Users with only the `Delete` permission are unable to delete global variables (or reference blocks) since they cannot view the reference blocks or global variables.
- To view or edit existing reference blocks, use the **Reference Blocks** options. For more information, see the Reference Blocks topic.
- To view the execution history of the playbook, use the **Execution History** option. For more information, see Playbook Execution and Debugging chapter.
- To change the execution priority for a playbook, use the **Execution Priority** option. For more details, see the Changing Prioritization of Playbook Execution topic.
- To apply a Jinja template to a JSON input and then render the output, use the **Jinja Editor** option. You can thereby check the validity of the jinja and the output before you add the jinja to the playbook. For more information, see the Dynamic Values chapter.

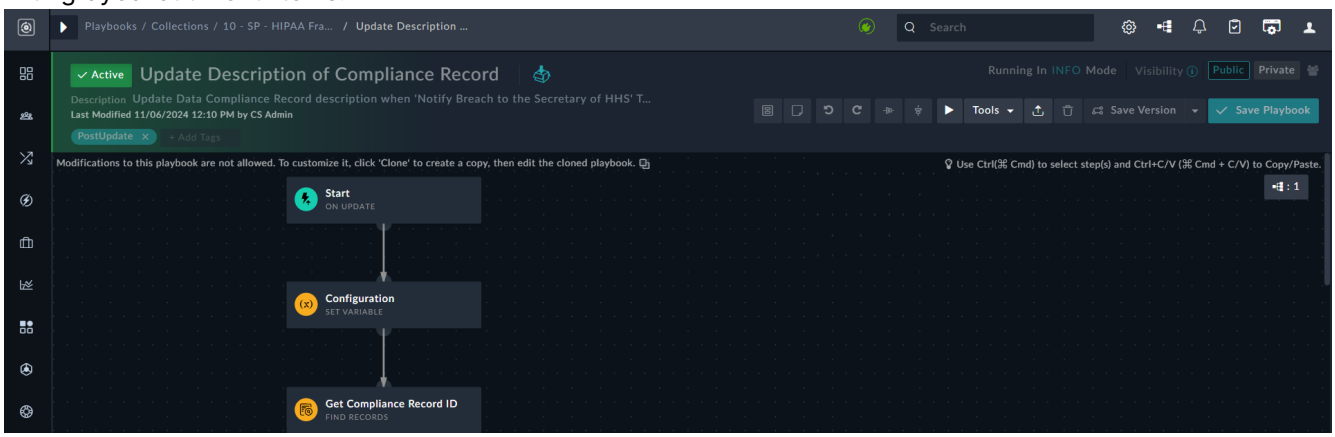
8. (Optional) Other actions that you can perform in the playbook designer are:
   - Use the **Export** button to export the playbook in JSON format.
   - Use the **Delete** button to delete the playbook.
   - Use the **Trigger Playbook With Sample Data** button to run the playbook directly from the playbook designer. This simplifies testing and debugging during development.. For more details, see the Playbook Execution topic.
   - Use the **Auto-Align - Vertical** and **Auto-Align - Horizontal** buttons to align the playbook vertically or horizontally.
   - The Undo and Redo buttons are very useful while building a playbook when there is a lot of trial and back and forth to be done. Use the **Undo** button or use `Ctrl+z`(Windows)/`Cmd+z` (Mac) to reverse changes made in a playbook, and use the **Redo** button or use `Ctrl+y`(Windows)/`Cmd+shift+z`(Mac) to reverse the steps that you have undone; therefore, you can use the **Redo** operation only after you have performed the **Undo** operation in a playbook. The playbook designer displays messages about the effect of the Undo/Redo operation in the bottom-right corner. When you perform bulk operations such as moving, cloning, or deleting a number of steps in one go, clicking **Undo** reverts the step modification. Similarly, if you have modified a step and saved it, clicking **Undo**, reverts the step modifications. Note that when editing inputs in the step argument form, the browser's default change tracking is in effect; therefore, the Undo/Redo operations are applicable only after you save the step. Also, note that if you have made multiple changes in a small time period (around a second), then all these small changes are considered a single operation.
   - To add a block to a playbook, click the **Create Block** button, and to add a note for the playbook, click the **Add Note** button. For more information, see the Adding Blocks and Notes topic.
9. Once you have completed updating the playbook, click **Save Playbook**.

Editing playbooks that are part of Solution Packs

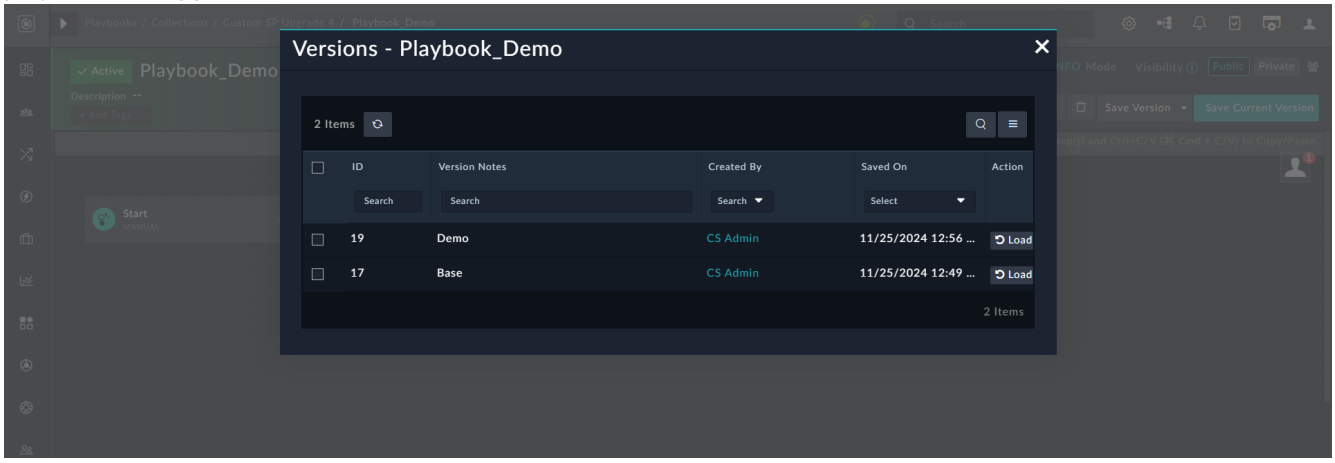
Prior to release 7.6.1, it was recommended to clone playbooks from solution packs or connectors before editing them. This was necessary because sample playbooks were overwritten during upgrades.

Starting with release 7.6.1, customized playbooks that are part of a solution pack will no longer be overwritten during solution pack upgrades. This change ensures that custom modifications to playbooks are retained. However, playbooks that are part of a connector will still be overwritten when the connector is upgraded.

When solution packs are installed or upgraded, playbooks imported with the solution pack may be non-editable, with options and menu items either hidden or grayed out. The following image shows a non-editable playbook with grayed-out menu items:



To modify a non-editable playbook, you must first clone the playbook and then make your changes. Some playbooks, such as those that need user configuration to work, for example, playbooks that block IP addresses, can still be edited directly, even if they are part of a solution pack. If you have previously edited a playbook (without cloning it) that was part of a solution pack, and the solution pack is later updated, the updated playbook will appear with a '**Base**' version label:



You can compare the 'Base' version with your edited version by using the **Load** button to view both the versions. To incorporate changes from the updated version, you can either:

- Copy the updated steps or blocks from the 'Base' version to your edited playbook and save that playbook. For information on how to add blocks to playbooks, see the Adding Blocks and Notes topic.
- Load the 'Base' version as your main playbook, save it, and then continue to update the playbook.

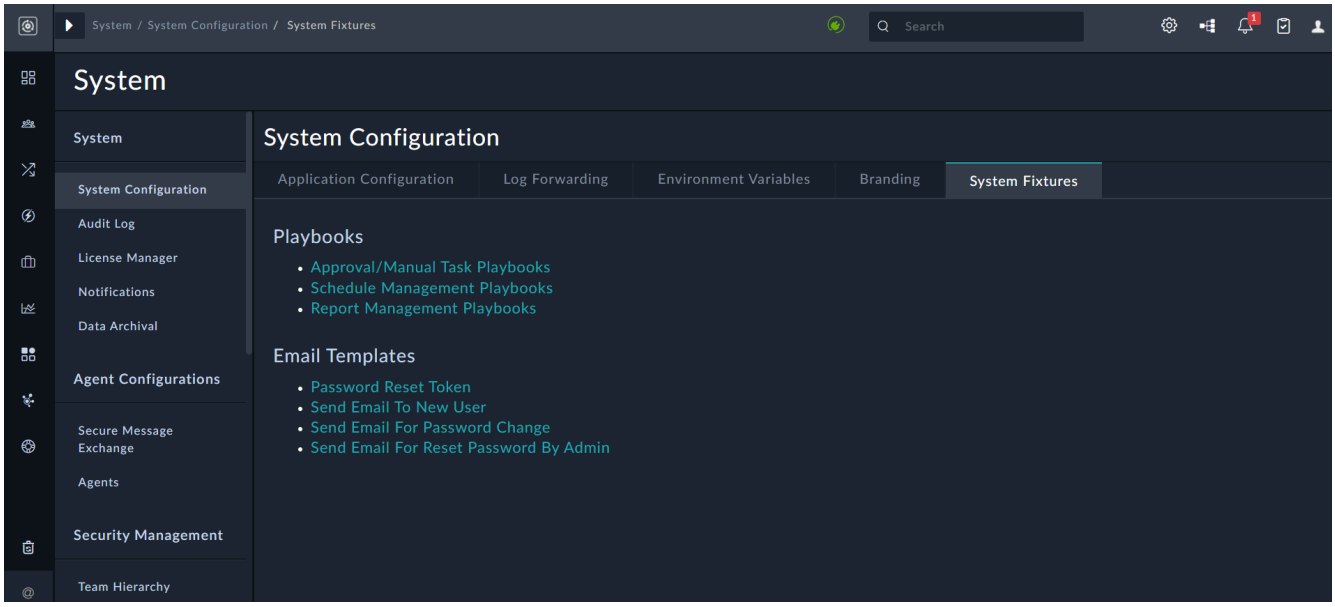
You can save this version of the playbook and continue to work on it or you can click **View Current Saved Playbook** to revert back to the state of the playbook when it was last saved.

System Playbooks

FortiSOAR includes some system playbook collections that are used to automate tasks, such as the `Schedule Management Playbooks` collection can be used to schedule various tasks such as cleaning up playbook execution history, purging integration logs, etc. Or, you can use the `Report Management Playbooks` collection to

manage generation of reports. For example, the `Generate Report By Scheduler` playbook generates reports based on schedules that you have specified. You can also reference system playbook from other playbooks.

The FortiSOAR UI includes links on the `System Configuration` page to the various playbook collections and templates, which are included by default when you install your FortiSOAR instance. Administrators can click the **Settings** ( ) icon to open the `System Configuration` page and click the **System Fixtures** tab to access the system playbooks or fixtures. The `System Fixtures` page contains links to the system playbook collections and templates. Administrators can click these links to easily access all the system fixtures to understand their workings and make changes in them if required. For example, to access Schedule Management Playbooks, click the **Schedule Management Playbooks** link.



You can modify system playbooks as per your requirements. However, incorrectly modifying any system playbook can affect FortiSOAR functionality.

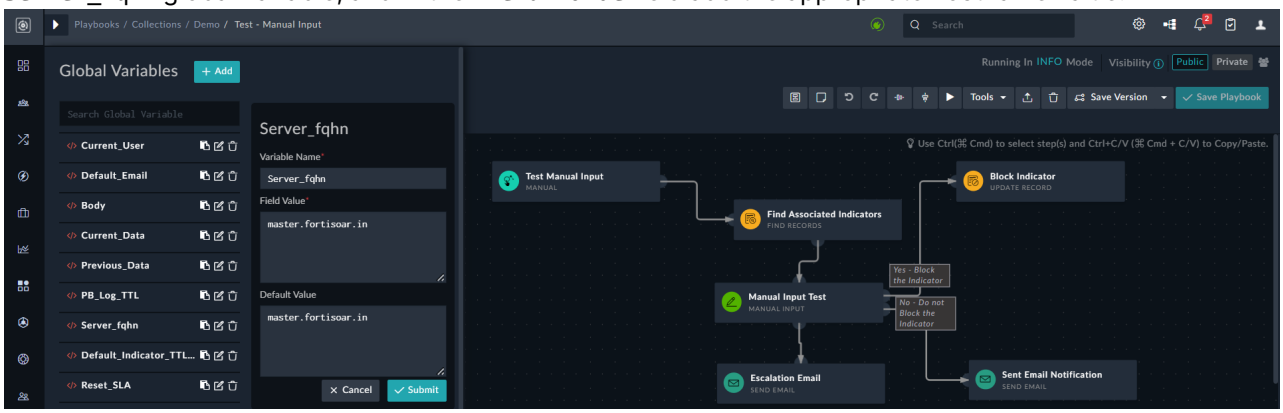
For example, if you want to modify the default email signature, which is currently `Regards, FortiSOAR Admin`, for a system playbook, open the playbook and double-click on its `Send Email` step. In the `Send Email` step, in the **Content** field, modify the signature as per your requirements and click **Save**.

In the system playbook (or any playbook) that is sending an email, ensure that you have used the `Server_fqhn` global variable in the `Send Email` step.

When you are using a system playbook that sends an email, for example, when an alert is escalated to an incident, and an Incident Lead is assigned, then the system playbook sends an email to the Incident Lead specified. The email that is sent to the Incident Lead contains the link to the incident using the default hostname, which is the hostname that you had specified or that was present when you installed FortiSOAR. To ensure that the correct hostname is displayed in the email, you must update the appropriate hostname as per your FortiSOAR instance, in the playbook, using the Playbook Designer as follows:

1. Open the Playbook Designer.
2. Click **Tools** > **Global Variables** to display a list of global variables.

3. In the **Global Variables** pane, search for the `Server_fqhn` global variable, then click the `Edit` icon in the `Server_fqhn` global variable, and in the `Field Value` field add the appropriate hostname value.



   You can optionally specify a default hostname value in the `Default Value` field.

4. Click **Submit**.
   This adds the updated hostname for your incident and then when a system playbook sends an email the link contains the correct hostname.

Playbooks that contain a reference to the `approvalHost` global variable fail with the '`approvalHost variable undefined`' error, since the `approvalHost` global variable is removed from release 7.2.0 onwards. To resolve this error, replace the `approvalHost` global variable in the playbook with the `Server_fqhn` global variable.

For information about all the system playbook collections and templates, which are included by default when you install your FortiSOAR instance, see the System Fixtures topic in the "Administration Guide."

Importing Playbooks

You can import playbook collections into your FortiSOAR instance using **Automation** > **Playbooks** . To import a collection click **Import** in the Collections (left) pane. To import individual playbooks, click **Import** in the Playbooks (right) pane. Playbooks and collections must be in JSON format to be imported successfully.

You can also use the Export and Import Wizards to manage playbooks and playbook collections.. For more information, see the Export and Import Wizards topic in the *Application Configuration and Customization* chapter of the "Administration Guide".

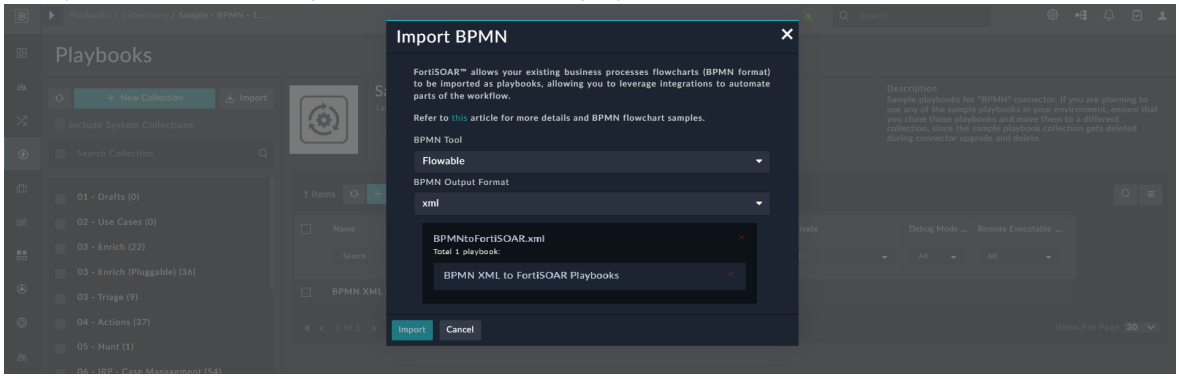
Importing BPMN Shareable Workflows

FortiSOAR provides you with the ability to convert a BPMN Shareable Workflows to FortiSOAR playbooks. Business Process Model and Notation (BPMN) is a tool using which you can create flowcharts, and these flowcharts tend to be specific towards cybersecurity workflows. Therefore, this feature provides you with the advantage of importing your BPMN workflows and directly converting them into FortiSOAR playbooks, without the need to again create the same workflow in FortiSOAR.

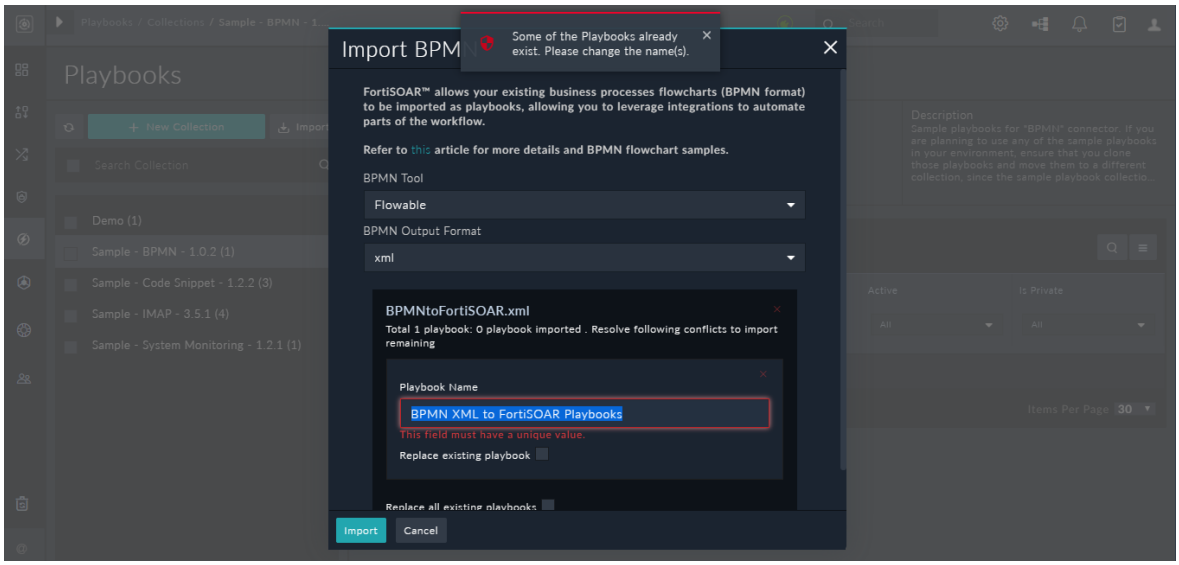
The feature is introduced as a "BETA" feature with more enhancements being planned to be added in the subsequent releases to make the BPMN import more robust.

Import the BPMN Shareable Workflows into FortiSOAR as follows:

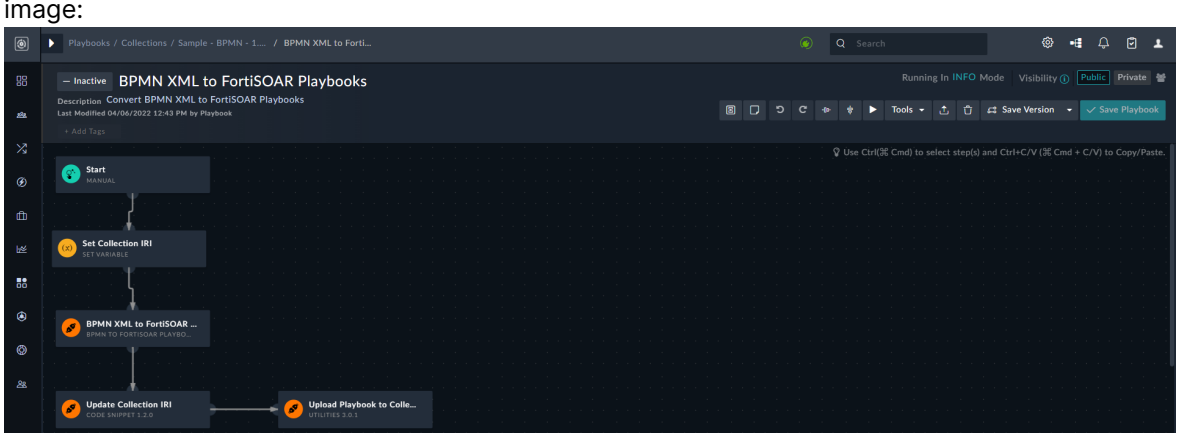
1. Export your BPMN Shareable Workflows from your tool, such as Flowable, Camunda, or Signavio. BPMN workflows are exported in the XML format.
2. To import the BPMN workflows into FortiSOAR:
   Note: FortiSOAR supports importing only a single BPMN workflow, i.e., you cannot import a collection of BPMN workflows.
   a. Log into FortiSOAR and click **Automation** > **Playbooks** in the left navigation bar.
   b. Click **Import BPMN**, which opens the `Import BPMN` dialog.
      Note: We are providing a "BETA" Version of this feature so that users can get a preview of this feature.
   c. In the **Import BPMN** dialog, do the following:
      i. From the **BPMN Tool** drop-down list, select the tool in which you have created your BPMN workflows.
         Note: FortiSOAR supports Flowable, Camunda, or Signavio.
      ii. From the **BPMN Output Format** drop-down list, select the output format in which you want to convert your BPMN workflow.
         Note: FortiSOAR supports only XML as an output format.
      iii. Drag and drop the BPMN XML file, or click the **Import** icon and browse to the XML file to import the BPMN XML file into FortiSOAR.
         If the XML of the BPMN workflow does contain errors, then a warning will be displayed in the `Import BPMN` dialog, which will contain the reason why the XML cannot be imported into FortiSOAR.
         If the XML of the BPMN workflow does not contain any mismatched elements or any other errors, then you will be able to import the workflow as a playbook in FortiSOAR.



      iv. To import the BPMN workflow file, click **Import**.
         This imports the workflow as a playbook in FortiSOAR with the same name as the workflow.
         Note: The name of the playbook must be unique, i.e., if you have two workflows with the same name that you want to import, you must either change the name of the playbook or click the **Replace existing playbook** checkbox to replace the existing playbook.



FortiSOAR displays the imported workflow in the Playbook Designer as shown in the following image:



Now you can edit the playbook as required in the playbook in FortiSOAR and easily create the automated workflow.

Translation of BPMN Workflow Steps

The following table specifies which BPMN (*Flowable* in this case) workflow steps maps to which of the FortiSOAR steps in the playbooks:

Flowable (BPMN) step	FortiSOAR steps	Notes
SequenceFlows	Routes	Any `SequenceFlows` defined in your BPMN workflow get converted to a `Decision` step in FortiSOAR playbooks.
StartEvents	Trigger steps	Your BPMN workflow must mandatory have a “Start” event which is the starting point of the BPMN workflow. The Start event in the BPMN workflow get converted to a `Manual Trigger` in FortiSOAR playbooks.
Gateways	Decision Step	Your BPMN workflow must mandatorily have a “Flow Condition” input

		which must be referenced to the Gateway ID.
UserTasks	Manual Tasks step	Note: If the `<userTask>` is not created according to FortiSOAR Manual Task step requirements, then a generic manual task step is created in the FortiSOAR playbook instead of failing the playbook. After you import the workflow you can update the manual task step.
ServiceTasks	Create Record step Or Update Record step	A `<serviceTask>` in your BPMN workflow must have the following: - A "Class" attribute to validate the model. - The "Class" attribute must be specified as a `module` - Addition of a "Class field" which contains either Create or Update.
ScriptTasks	Connector step or as a Code Snippet step	A `<scriptTask>` in your BPMN workflow must have the following: - `Name` = `{{ConnectorName}}` - `scriptFormat` = `{{FortiSOAR Connector Action}}` - `<script>` => `CDATA[ {{property mapping}} ]` **Note**: If the connector that you have defined in the `<scriptTask>` step is not installed in your FortiSOAR instance, then a generic connector step is created in the FortiSOAR playbook instead of failing the playbook. After you import the workflow you can update the connector step.

MailTasks	SMTP step	The `mailTask` is type of a `<serviceTask>` and it must be defined in your BPMN workflow as following:
		`<serviceTask>`
		`Flowable:type` = `mail`

HttpTasks	FortiSOAR Utility Step (REST API call)	The `httpTask` is type of a `<serviceTask>` and it must be defined in your BPMN workflow as following:
		`<serviceTask>`
		`Flowable:type` = `http`

Playbook Execution and Debugging

Playbook Execution

FortiSOAR allows developers to run and debug playbooks directly from the Playbook Designer, streamlining the development workflow. Playbooks can be edited and triggered in place, eliminating the need to navigate to modules, select records, initiate playbook execution, and return to the designer for modifications.

Triggering a playbook from the designer starts the execution of the playbook, which can cause changes to your data, leading to unintentional changes or loss of data. Therefore, it is important to review the playbook before it is triggered.

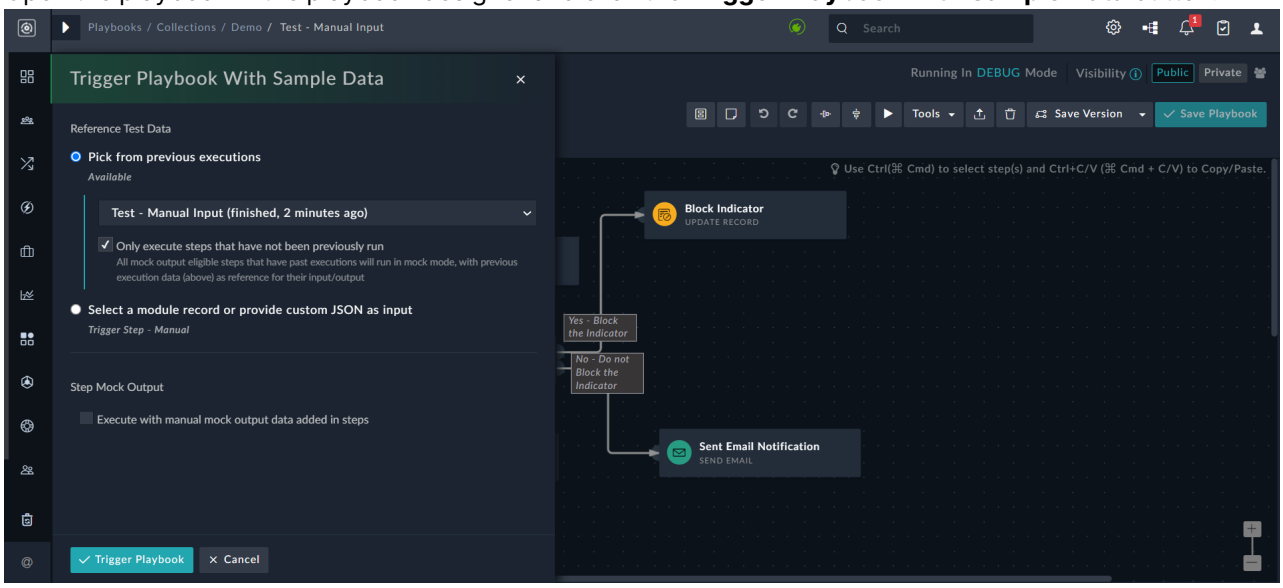
In release 7.4.2, the user experience of using mock output is streamlined, where the mock outputs can optionally be ingested from the most recent successful playbook execution, saving users the time of having to manually provide mock outputs on a step-by-step basis. This ensures users can test playbooks in development environments without needing to re-execute previously successful steps to obtain live results.

If you want subsequent playbooks to be run in the mock mode using the data from the previous execution, the playbook must be executed in the 'DEBUG' mode. To know more about playbook modes, see the Setting the logging levels for playbooks topic.

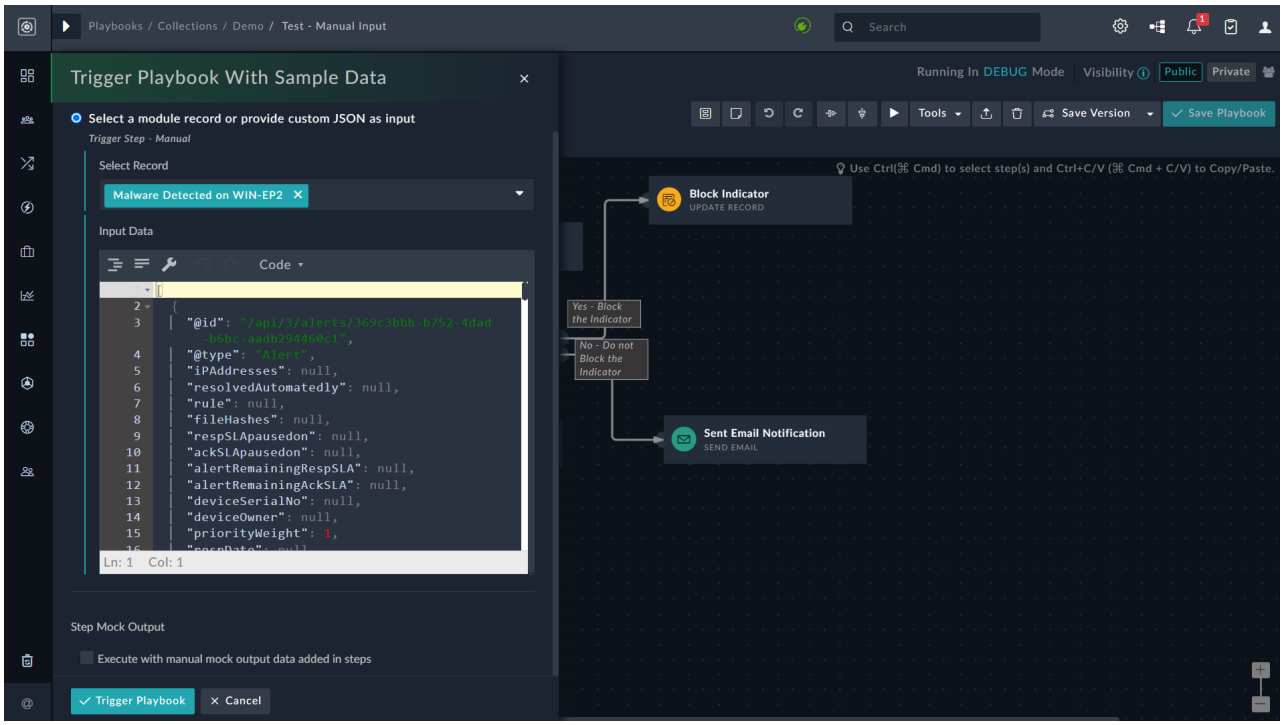
Only steps where 'Mock Output' can be enabled at the step level (see the Mock Output topic in the Playbook Triggers chapter) are supported for enabling mock output at the playbook level. Examples of steps that support mock output include create record, ingest bulk feed, reference playbook, and so on. Steps like manual task, wait, approval, and so forth do not support mock output.

To trigger a playbook from the playbook designer with mock output, do the following:

1. Open the playbook in the playbook designer and click the **Trigger Playbook with Sample Data** button.

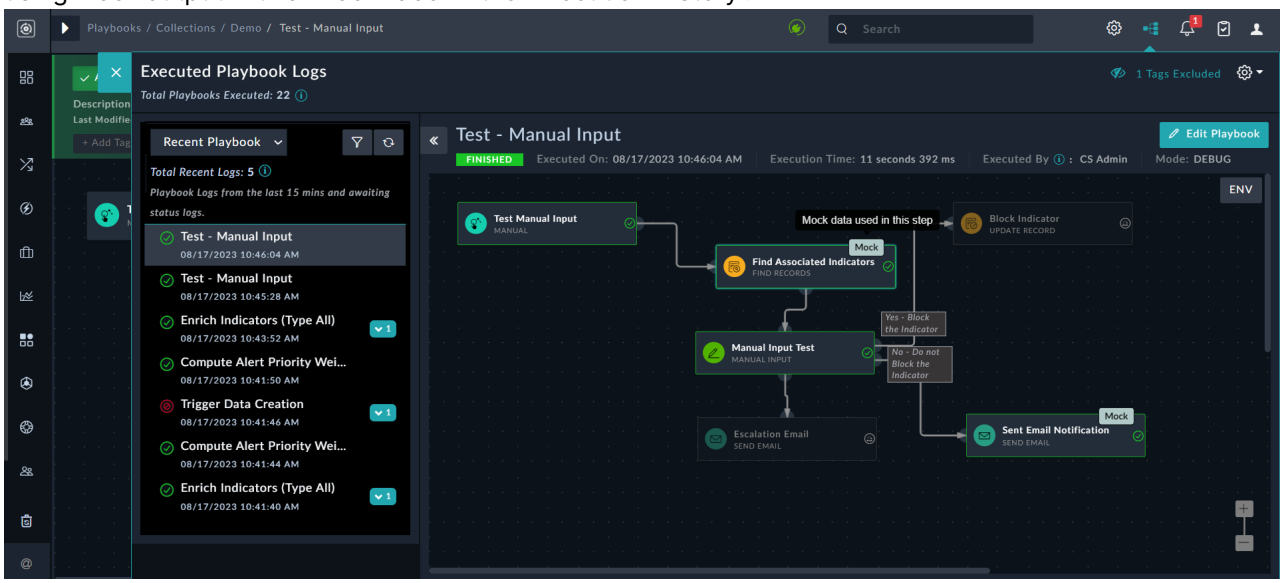


2. If you have run the playbook earlier in the 'Debug' mode, you can choose the **Pick from previous executions** option. Then, from the drop-down list, select the playbook execution with whose environment you want to trigger the playbook.
   When running steps that support "Mock Output" and have already been run, select the **Only Execute Steps that have been previously run** option to run the eligible steps in the 'Mock' mode to use the data from the prior executions as a reference for their input/output. Selecting this option displays steps that are run using mock data with a 'Mock' label in the 'Execution History'.
   OR
   You can choose the **Select a module record or provide custom JSON as input** option, where you can either select a module record or provide custom JSON input to trigger the playbook.
   If you have a playbook with a 'Manual' trigger step, then from the **Select Record** drop-down list, choose the record(s) using whose data, i.e., fields and values, you want to use to trigger the playbook.
   NOTE: Only the 30 most recent records are shown in the **Select Record** drop-down list.



ELSE,
Enter custom JSON in the **Input Data** field to trigger the playbook using custom JSON.

3. Select the **Execute with manual mock output data added in steps** option to execute the playbook with the mock output data that has been added in the steps. Selecting this option displays steps that are run using mock output with a 'Mock' label in the 'Execution History':



4. Click **Trigger Playbook** to trigger the playbook using the selected options.

The playbook can also use the "Mock Output" defined in the steps while running the playbook if you choose to **Enable** mock output.

Triggering Playbooks – Notes & Recommendations:

- **Use Mock Output for Debugging**: When triggering a playbook, mock output can be enabled for supported steps such as *Create Record*, *Ingest Bulk Feed*, and *Reference Playbook*. Mock output, provided in JSON format, simulates real outputs to assist in debugging. For details, see the Mock Output topic.
  NOTE: Steps requiring human input—such as *Manual Input*, *Wait*, and *Approval*—do not support mock output. These steps must be validated during actual playbook execution.
- **Triggering with Sample Data Options**: The **Trigger Playbook with Sample Data** option in the Playbook Designer offers several input simulation options:
  - **Pick from Previous Executions**: Use data from past successful executions (in JSON format) to simulate input without re-running actions. This feature is particularly useful in DEBUG mode to test playbooks without affecting real data.
  - **Choose a data source**: Select from the latest 30 records in a module or input custom JSON to provide playbook input.
  - **Execute with Manual Mock Output**: Mock data can be manually added in supported steps to simulate outputs from previous steps.

Data Integrity Warning: Triggering a playbook may modify data. Always review playbook logic carefully before execution, especially when mock output is used.

Changing Prioritization of Playbook Execution

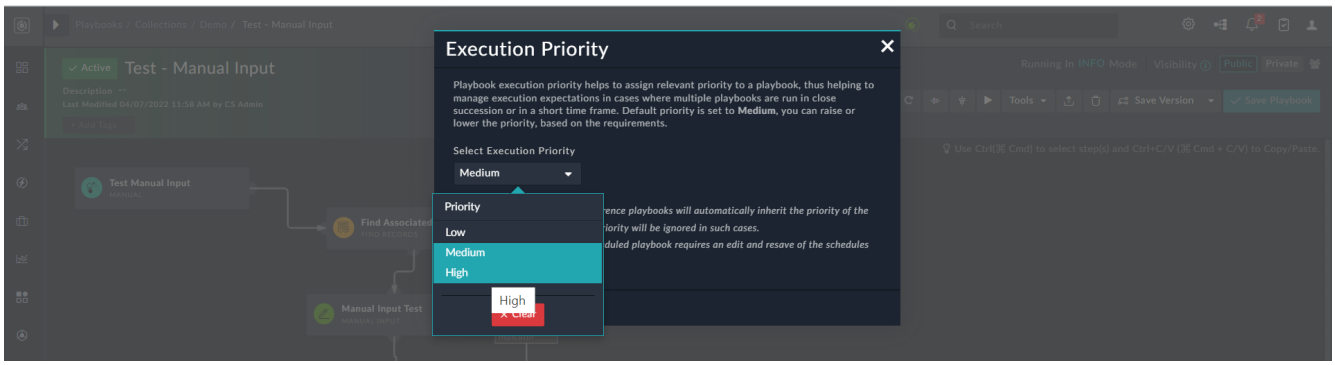
You can change the prioritization of playbook execution based on the importance of that playbook, thereby enabling the higher priority playbooks to be executed first even if there are some normal priority playbooks already queued for execution. Earlier, the playbook execution queue was based on first in first out method, with round robin assignment of workers (processes), which meant that important playbooks might get queued after lower-priority playbooks.

For example, if you have set up data ingestion to run every minute, then possibly you would have many data ingestion playbooks queued up, and then if you also require to run an important playbook with a manual action, it would earlier be run only once the data ingestion task that was scheduled before it was completed. Now, you can change the prioritization of the manual input playbook to "High" enabling it to get executed on priority.

You can set the priority for playbook execution as High, Medium, or Low. The default priority is set as "Medium". Playbook execution prioritization works as follows:

- If any worker is available for the task execution, it gets assigned a task from the "High" queue first and so on.
- If all workers are occupied with lower priority tasks and any higher priority task comes up, the high priority task gets executed only when any worker is again available.
- Low priority tasks do not get executed if there are high priority tasks.

To set a priority for playbook prioritization, open that playbook in the playbook designer. Click **Tools** > **Execution Priority**. In the `Execution Priority` dialog, you can set the playbook execution prioritization to **High**, **Medium**, or **Low**:



To list the number of pending tasks (workflow count) in the playbook ('celery') queue, use the following command:
`sudo rabbitmqctl list_queues -p fsr-cluster --no-table-headers --silent | grep -E "^\s*celery\s+" | awk '{print $2}'`
When there is no queue, it will display 0 (default), and when the queue builds up, it will display the queue count number (number of pending tasks) such as 10, 25, etc.

Notes:

- All 'sync' reference playbooks automatically inherit the priority of their parent playbooks, thereby ignoring any preset priority.
- If you update the execution priority of a scheduled playbook, then you require to edit and resave the schedules associated with that playbook.
- If you want to schedule a data ingestion playbook, then you must set the priority of the data ingestion playbook before scheduling the same.

FortiSOAR also integrates with a GUI-based celery monitoring tool called **Flower**, using which you can monitor and administer celery cluster and playbook execution queues. You can start a Flower web server using the following process:
`#cd /opt/cyops-workflow/sealab`
`#../.env/bin/celery -A sealab flower --port=5555`

Note: Ensure that port that you are specifying in the URL, `5555` in the above sample, is opened in your firewall and can be accessed.

Disabling Prioritization of Playbook Execution

You can disable playbook priority, in case playbook priority queuing is hampering your system's performance. FortiSOAR uses rabbitmq as the message broker, and because of this priorities of an already declared queue in rabbitmq cannot be changed dynamically; for more information, see https://www.rabbitmq.com/priority.html#using-policies.

Use the `ENABLE_PRIORITY` setting in the workflow engine to enable/disable playbook priority. However, before changing the `ENABLE_PRIORITY` setting, first delete the `celery` queue, after you have ensured that no data is present in the `celery` queue.

To enable/disable the `ENABLE_PRIORITY` setting in `/opt/cyops-workflow/sealab/sealab/config.ini`, do the following:

1. List queues and check the `celery` queue and its count of messages, use the following command:
   `sudo rabbitmqctl list_queues -p intra-cyops`

If the `celery` queue has zero messages in it, then the output would be:
`celery 0`
This ensures that no data is present in the `celery` queue.

2. Delete the `celery` queue , using the following command:
`sudo rabbitmqctl delete_queue celery -p intra-cyops`
3. Change the `ENABLE_PRIORITY` flag to `false` to disable playbook priority, or `true` to enable playbook priority.
4. Restart `celeryd` using the `# sudo systemctl restart celeryd` command.

Playbook Debugging

You can easily debug playbooks in FortiSOAR using execution history and executed playbooks logs.

The Integrations API call supports only `POST` calls. Therefore, if you have any existing playbooks that use the `GET` calls, then those playbooks will fail. To resolve this issue, you have to manually change the method from `GET` to `POST` in your playbooks.

You can define the logging levels (`INFO` or `DEBUG`) for your playbook execution logs, both globally as well as at the individual playbook level. For more information on playbook logging levels, and how to set those levels on individual playbooks, see the Setting the logging levels for playbooks topic. Note that the 'Debugging Playbooks' content assumes that you are running the playbooks in the `DEBUG` mode.

As you develop more sophisticated Playbooks, the ability to easily debug playbooks becomes exceeding important. FortiSOAR has designed the Execution History to make it easier for you to see the results of your executed playbooks and for you to debug playbooks.

Use the **Executed Playbook Logs** icon ( ) that appears on the top-right corner of the FortiSOAR screen to view the logs and results of your executed playbooks as soon as you log on to FortiSOAR. You can also use the executed playbook logs to debug your playbooks.

FortiSOAR implements Playbook RBAC, which means that you can view logs of only those playbooks of which you (your team) are the owner. For more information, see the Ownership, Roles, and Permissions topic.

The Execution History provides the following details:

- Playbooks have been organized by the parent-child relationship.
- Playbooks have a console using which you can see debug messages with more significant details.
- Playbook designer includes the playbook execution history option.
- Playbooks can be filtered by Playbook Name or Record IRI, user, date range, or status.
- Playbook Execution History Log contains details of the playbook result, including information about the environment and the playbook steps, including which steps were completed, which steps are awaiting some action, which steps were failed, and which steps were skipped.

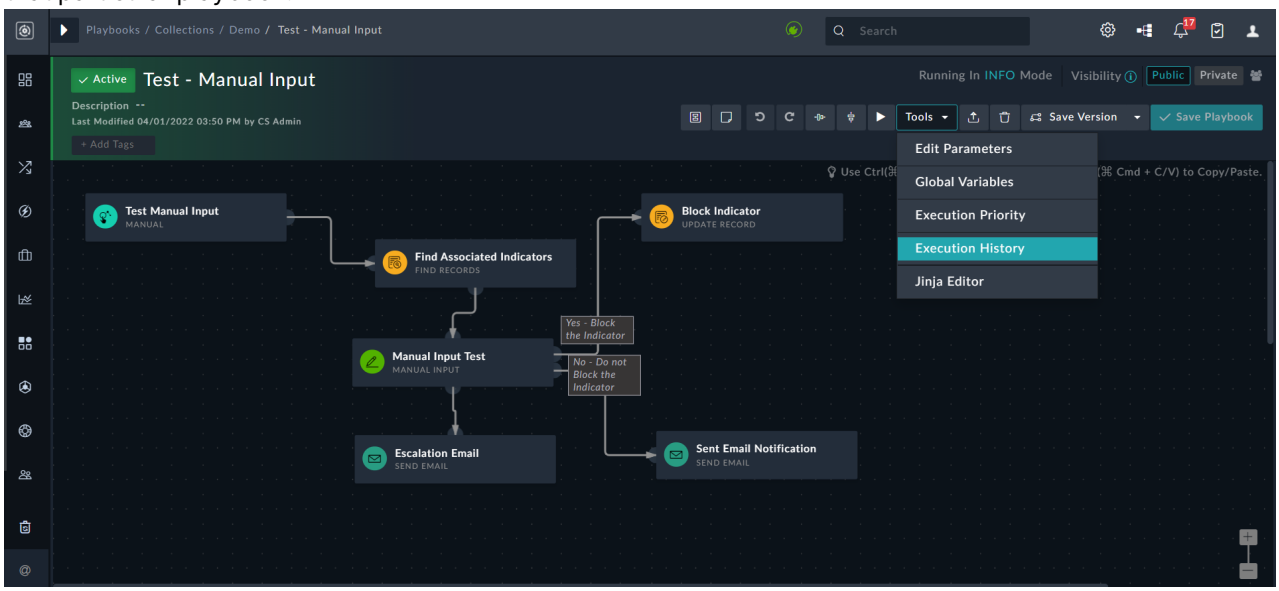
From version 7.0.2 onwards, users will not be able to view the execution history of 'legacy' playbooks, i.e. the execution history of playbooks that were run before release 6.0 will not be visible.

The Executed Playbook Logs do not display the Trace information from the error message so that the readability of the Executed Playbook Logs is enhanced since the clutter in the error details screen is reduced and you can directly view the exact error. The Trace information is yet present in the playbook logs.

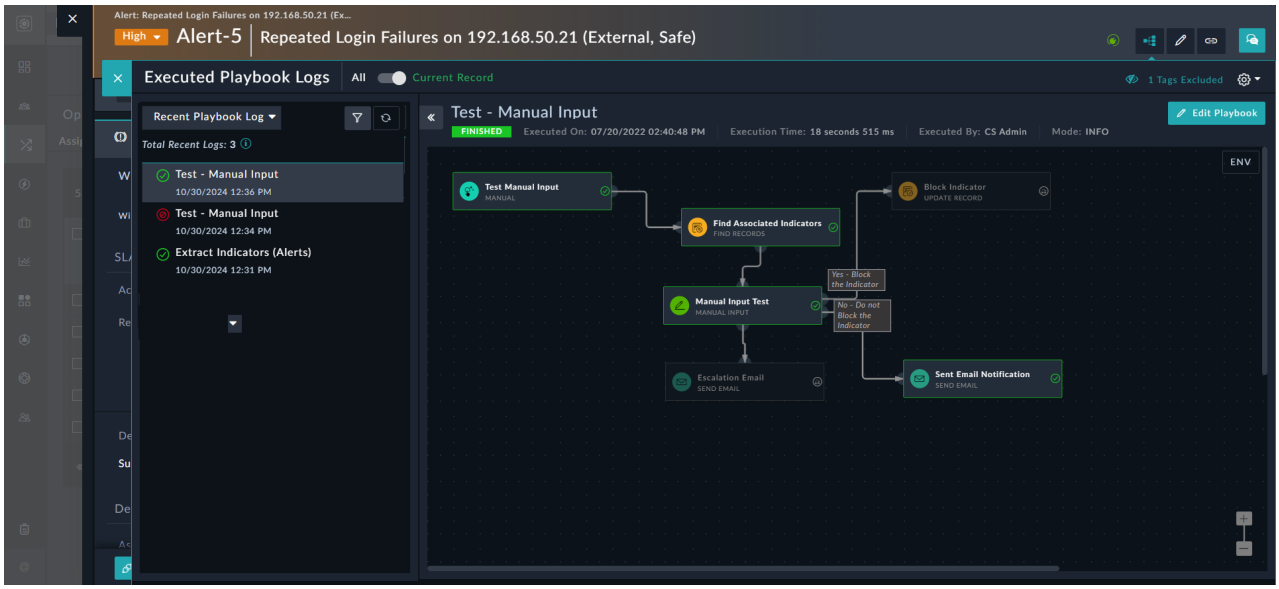
FortiSOAR also contains enhanced the error messages that are precise and detailed making it easier for you to debug playbook issues. For information about the common playbook error messages and how to debug them, see the Debugging common playbook and connector errors article present in the Fortinet Knowledge Base.

You can access the playbook execution history as follows:

- Clicking the **Executed Playbook Logs** icon ( ) in the upper right corner of the FortiSOAR screen. You have an option of purging executed playbook logs from the `Executed Playbooks Log` dialog. For more information see the Purging Executed Playbook Logs topic.
- Clicking **Tools** > **Execution History** in the playbook designer to view the execution history associated with that particular playbook.



- Clicking the **Executed Playbook Logs** icon in the detail view of a record such as an alert record to view the playbooks that have been executed on that particular record in a flowchart format. This makes it easier for users to view the flow of playbooks, especially useful for viewing the parallel execution paths in playbooks.



You can toggle between the **Record** view, which displays only the logs of the playbooks that are executed on that particular record and the **Global** view displays logs for all the playbooks that are executed on the FortiSOAR system. You can also purge executed playbook logs for a particular record by clicking the **Settings** icon on the top-right of the `Executed Playbook Logs` dialog in the 'Detail' view of that record, and then selecting the **Purge Logs** option. For more information, see the Purging Executed Playbook Logs topic.

Playbook Recovery

FortiSOAR autosaves playbooks so that you can recover playbook drafts in cases where you accidentally close your browser or face any issues while working on a playbook. These unsaved (autosaved) drafts do not replace the current saved version of the playbook and only ensure that you do not lose any of your work done in the playbook, by providing you the ability to recover the drafts.

Autosaves in FortiSOAR is user-based, which ensures that users see their own unsaved drafts of the playbook. Since it is browser-based, it comes into effect as long as the same browser instance is used by the user. Also, playbook drafts might not be saved if you are working in the "Incognito" mode.

By default, FortiSOAR autosaves playbook drafts **15** seconds after the last change. However, you can ask your administrator to change this time across all playbooks by modifying the time, in seconds, on the **Application Configuration** tab in the `System Configuration` page. The minimum time that your administrator can set for saving playbook drafts is **5** seconds after the last change. You can also choose to disable (and later enable) playbooks recovery for all playbooks. For more information, see the System Configuration and Management chapter in the "Administration Guide."

If the browser data is cleared, then the autosaved drafts will get deleted.

Viewing and Editing Blocks

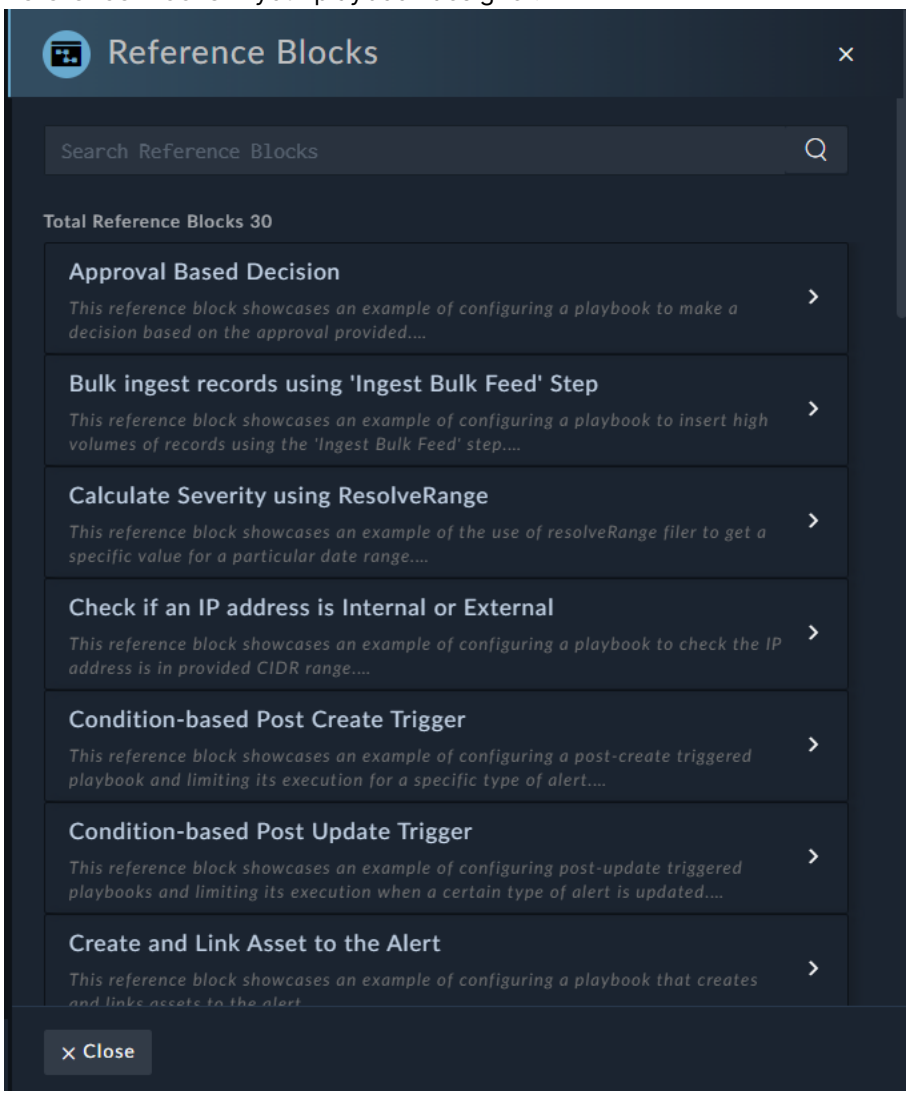
Reference Blocks

Reference Blocks are reusable, modular sets of playbook steps that serve as pre-built logic templates. They help simplify and accelerate playbook design by encapsulating commonly used patterns—such as trigger blocks, enrichment steps, and condition-handling flows. Using reference blocks, users can get contextual aid while building playbooks, including relevant samples and help references.

Reference blocks contain reference or sample steps or steps that can be added to your playbook to help you build your playbook. FortiSOAR provides some ready-made samples for widely used steps, including trigger steps in playbooks.

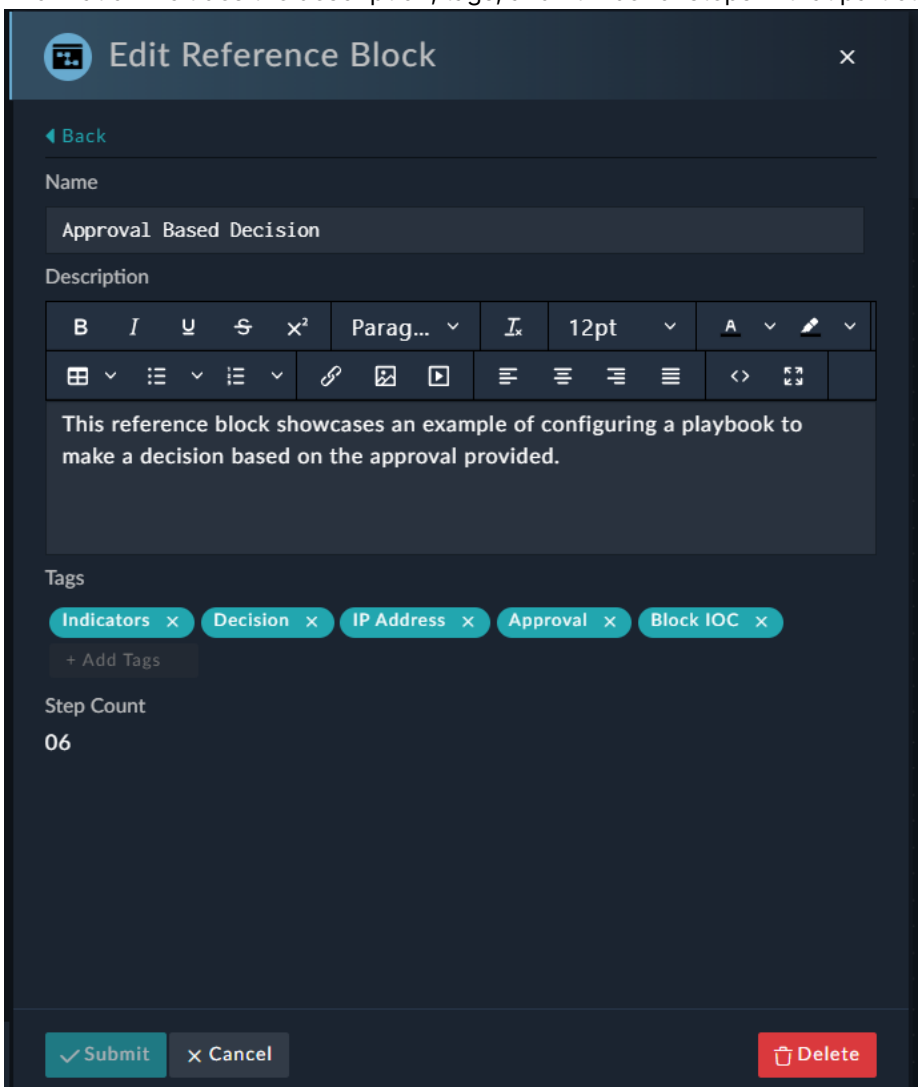
The SOAR Framework solution pack contains several sample reference blocks to provide you with all the help you need while building your playbook. For detailed information about the SOAR Framework Solution Pack (SP), see the SOAR Framework SP documentation in the Content Hub.

To view all the ready-made reference blocks, i.e., both trigger blocks and non-trigger blocks, click **Tools** > **Reference Blocks** in your playbook designer:

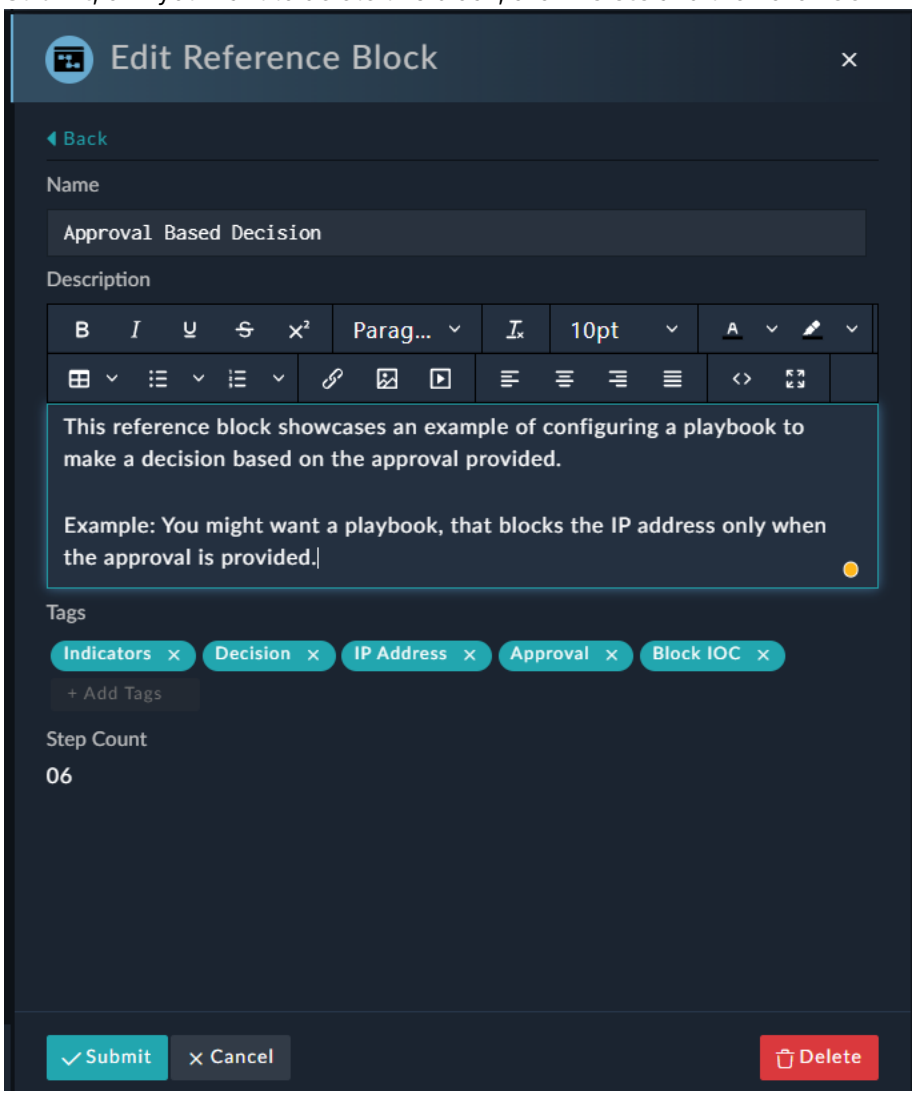


Use the **Search Reference Blocks** field to search for reference blocks using tags (exact match supported), the name of the block, or its description.
Click **>** or click anywhere in the block row to expand a particular block and get information about the block;

information includes the description, tags, and number of steps in that particular block:



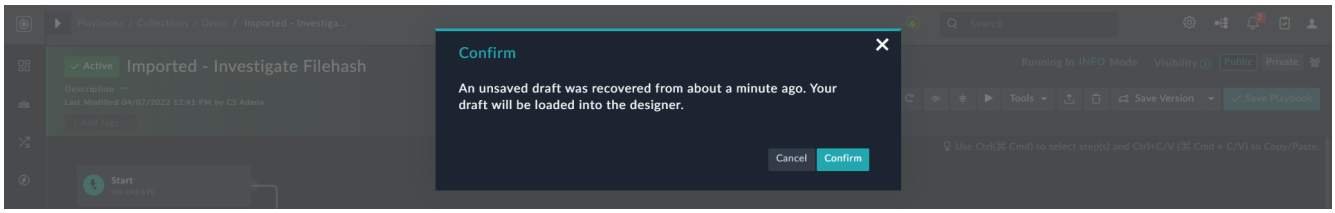
To edit a particular block, expand the block and update the name, description, or tags of that block, then click **Submit**, or if you want to delete this block, click **Delete** and then click **Confirm**:



If you add a reference block to a playbook and then make changes to any step or steps that form a block, then these changes are applicable only to the current playbook, and they are not reflected in the steps that form the reference (original) block.

You can export and import playbook blocks from one FortiSOAR environment to another using the Export Wizard and Import Wizard. For more information, see the Export and Import Wizards topic in the *Application Configuration and Customization* chapter of the "Administration Guide". Reference Blocks can also be included in Solution Packs, see the Install and Manage Solution Packs chapter in the "User Guide" for more information.

To recover an unsaved draft of the playbook, reopen that playbook in the playbook designer you will be prompted to confirm whether you want to recover the draft of the playbook as shown in the following image:



Once you click **Confirm** on the `Confirm` dialog, the autosaved version of the playbook is loaded in the playbook designer, and you can then choose to save this playbook using **Save Playbook** and make it the current working copy.

Key Behaviors and Capabilities for Reference Blocks:

- **Reusability with Isolation**: Adding a reference block to a playbook creates a local copy. Edits made to this instance affect only that playbook; the original reference block remains unchanged. This supports reuse of base logic while allowing customization without impacting shared templates.
- **Metadata Editing**: Reference block metadata—such as name, description, and tags—can be edited independently of the block's logic. These changes apply only to metadata, not the underlying steps.
- **Portability**: Reference blocks can be exported and imported across environments using the built-in Import/Export wizards, enabling portability in multi-environment deployments.
- **Pre-built Samples from SOAR Framework SP**: The SOAR Framework Solution Pack, includes curated reference blocks for common use cases such as, common trigger types, reusable code snippets, API handling and checks, etc. These serve as foundational components for building standardized and maintainable playbooks.
- **Bundling in Solution Packs**: Reference blocks can be included in Solution Packs for distribution and reuse across teams and environments. For packaging details, see the Install and Manage Solution Packs chapter in the "User Guide".

Trigger Blocks

'Trigger Blocks' are a specialized type of Reference Blocks that include a trigger step. They are designed to promote consistency, reduce redundancy, and streamline the design of playbooks with common initiation logic. For details, see the Trigger Block topic.

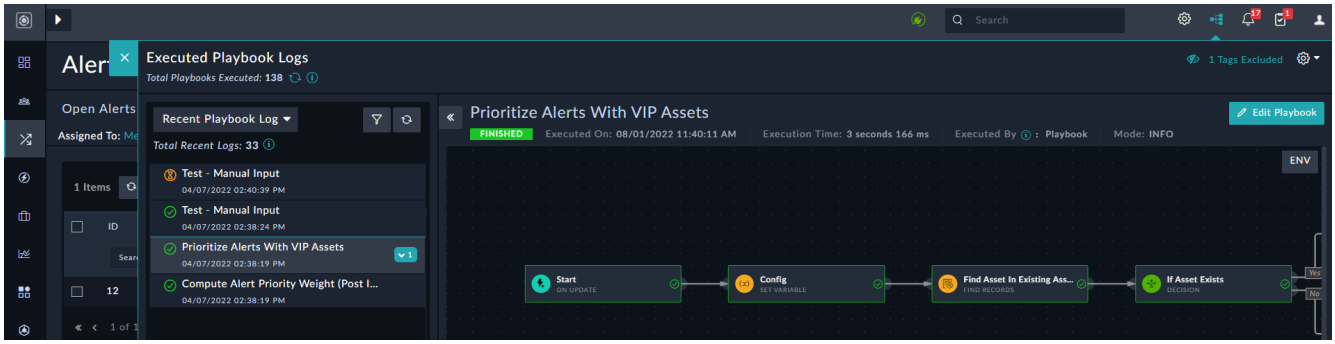
Best Practices for Using Trigger Blocks:

- **Organize and Tag Effectively**: When saving a trigger block, use clear, descriptive names, add meaningful tooltips and detailed descriptions, and apply relevant tags (e.g., `manual`, `api`, `alert`, `incident`, `trigger`). These details improve discoverability via the **Search Reference Blocks** field and provide clarity through the **Info** icon in the Playbook Designer.
- **Use for Complex Trigger Logic**: Trigger blocks are especially useful when handling advanced conditions such as *Visibility conditions*, *API-based triggers*, *Manual triggers*, etc. This simplifies onboarding and provides a consistent reference point for teams developing similar automation workflows.
- **Maintain a Standardized Library**: Build and maintain a centralized repository of validated trigger blocks (e.g., *Investigate Incident*, *Fetch Alert Data*). Sharing these across teams helps ensure consistent design patterns and logic.
- **Safe Customization**: When a trigger block is added to a playbook, any edits made apply only to that instance. This allows customization for specific use cases without modifying the original reference block.

- **Create and Reuse Variants**:If a customized version of a trigger block is needed across multiple playbooks, use **Save as Reference Block** to create a new version. Assign a unique name and relevant tags to support future discovery and reuse.

Viewing Executed Playbook Logs

Click the **Executed Playbook Logs** icon in the upper-right corner of FortiSOAR to view the logs and results of your executed playbook. Clicking the **Executed Playbook Logs** icon displays the Executed Playbook Logs dialog as shown in the following image:



The **Executed Playbook Logs** displays the executed playbooks in the flowchart format, as is displayed in the playbook designer. This makes it easier for users to view the flow of playbooks, especially useful for viewing the parallel execution paths in playbooks.

Playbooks are sorted by chronological datetime, with the playbook that was executed last being displayed first. All playbooks are displayed with **10** playbooks being displayed per page. Click a playbook in the list to display it in the flowchart format and also see the details of the playbook result, the environment and the playbook steps, including which steps are completed, failed, awaiting or skipped.

The Execution Playbook Log dialog also displays a count of the total playbooks executed, the date and time of when the playbook was executed, the time taken for executing the playbook and the mode in which the playbook was run, i.e., INFO (default) or DEBUG. From release 7.4.0 onwards, the default logging level for failed playbooks is set to DEBUG so that users do not need to rerun the playbook to view the exact reason for playbook failures. For more information on the modes of running a playbook and how to change the modes on individual playbooks, see the Setting Logging Levels topic.

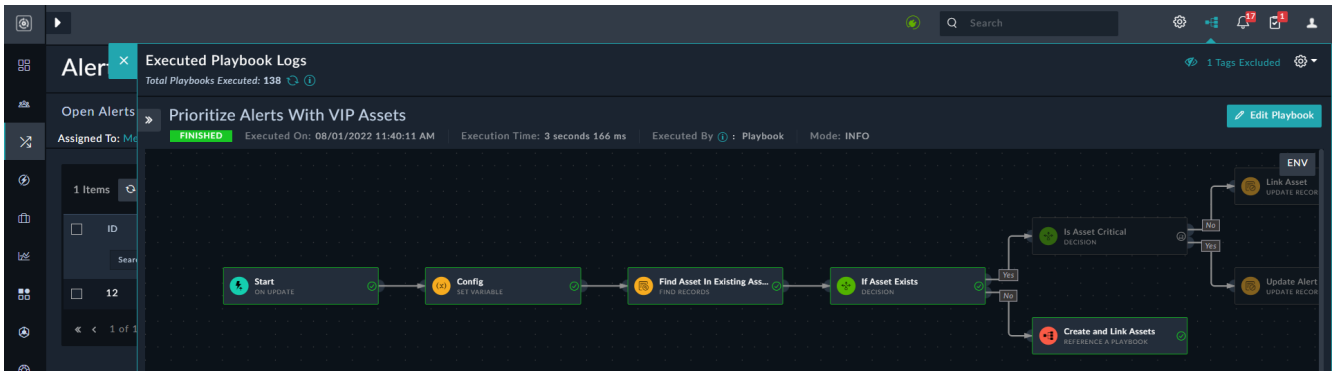
The executed playbook log also displays information about who has triggered or terminated the playbook in the 'Executed by' field as follows:

- Manually-triggered playbooks display the username of the user who has triggered or terminated the playbook in the 'Executed by' field.
- API playbooks that are triggered by a particular user's token display the username of that user in the 'Executed by' field.
- Playbooks that are triggered using the On Create, On Update, or On Delete triggers display 'Playbook' in the 'Executed by' field if the record creation, updation, or deletion is a result of an automated action using playbooks, for example, data ingestion, enriching indicators, etc.
- Playbooks executed by users from the playbook designer, irrespective of their trigger type, display the username of the user who has triggered or terminated the playbook in the 'Executed by' field.

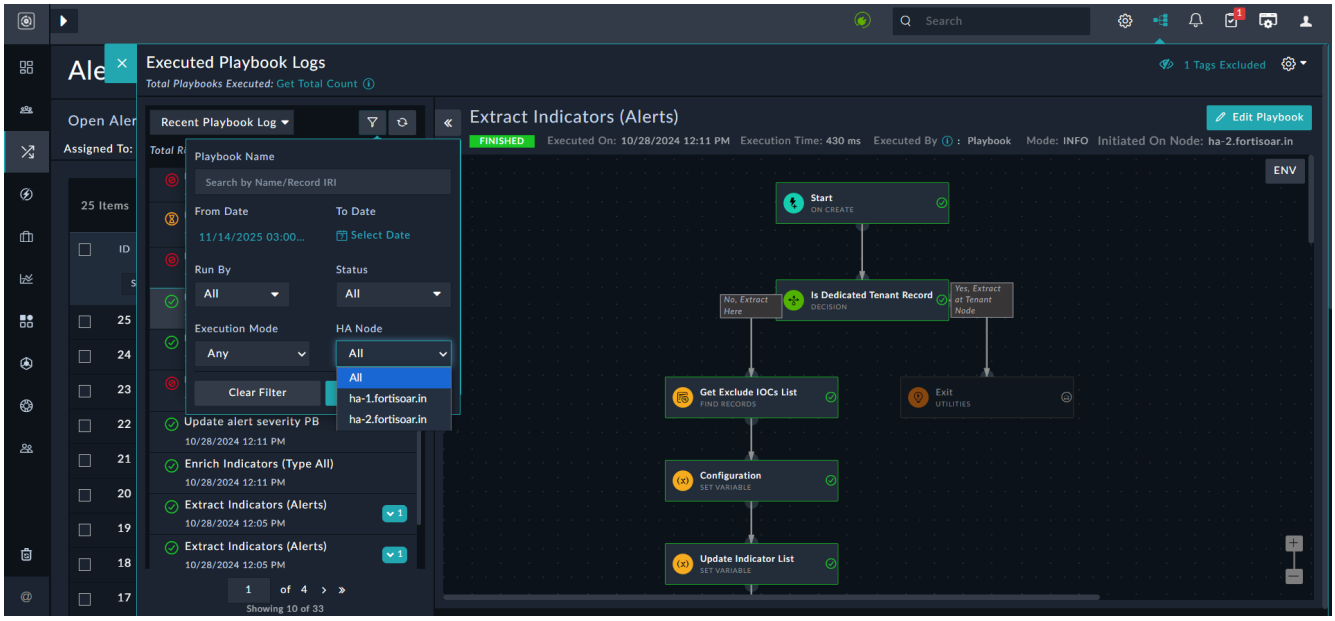
When you click on playbook steps, you can toggle the **ENV** button to toggle between the environment of the step and the output of the step. You can also copy the environment, error, and step details to the clipboard by clicking the **Copy 'Env' to Clipboard** or **Copy 'OUTPUT' to Clipboard** button.

You can also open the playbook directly in the playbook designer from the **Executed Playbook Logs** dialog by clicking the **Edit Playbook** button that appears in the right section of the dialog.

You can collapse and expand the **Executed Playbook Logs** dialog by clicking the **<<** or **>>** arrows as shown in the following image:



You can refresh the playbook logs using the **Refresh** icon, and filter logs associated with playbooks using the **Filter** icon:



Clicking the **Filter** icon allows you to filter playbook logs using the following options:

- **Playbook Name**: In the **Search by Playbook Name or Record IRI** field, filter the log associated with a particular playbook, based on the playbook name or the record IRI associated with the playbook. Example of filtering logs using the Record IRI: `/alerts/bd4bf0a6-b023-4bd7-a182-f6938fa37ada`.
- **From Date**: Use this field to filter logs by the date when playbooks were executed. Beginning with release 7.6.5, the default view shows playbook logs from the past 7 days. You can modify this limit or disable it entirely.Details on changing this setting are provided in the Managing the Day Filter Limit for Global Executed Playbook Logs topic.

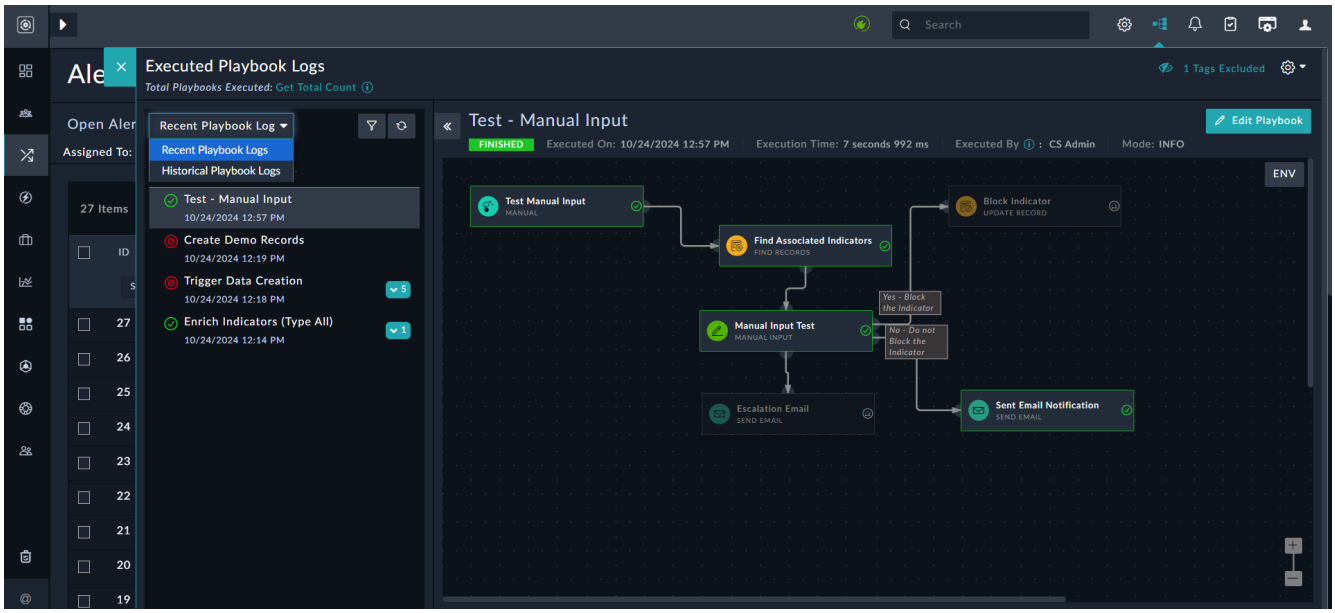
- **To Date**: Use this field to filter logs up to the date when playbooks were executed. Together, the From Date and To Date fields let you define a date range for retrieving playbook logs within that period.
- **Run By**: Use the **Run By** drop-down list to filter logs associated with a particular playbook, based on the user who has run the playbook.
- **Status**: Use the **Status** drop-down list to filter logs associated with a particular playbook, based on the status of the playbook execution.
  In case of the 'Recent Playbook Logs', you can choose from the following options: Incipient, Active, Awaiting, Paused, Failed, Finished, Skipped, Terminated or Finished with error. In case of the 'Historical Playbook Logs', you can choose from the following options: Failed, Finished, or Terminated.

  Playbooks stuck in the 'Active' or 'Incipient' state are terminated automatically when their permissible execution time limit is exceeded. Using the `CELERYD_TASK_SOFT_TIME_LIMIT` and `CELERYD_TASK_TIME_LIMIT` keys in `/opt/cyops-workflow/sealab/sealab/config.ini` you can modify the time limits for playbook execution. For more information, see the Optimizing and Troubleshooting chapter.
  Playbooks stuck in the 'Awaiting' state are automatically terminated after 7 days. To adjust this timeframe, you can modify the `WF_MAX_WAITING_DAYS` parameter in the `/opt/cyops-workflow/sealab/sealab/config.ini` file.
- **Execution Mode**: Use the **Execution Mode** drop-down list to filter playbook logs based on the execution mode of the playbooks. You can choose between INFO, DEBUG, or ANY.
- **HA Node**: Available only in high availability (HA) cluster environments, the **HA Node** drop-down list allows you to filter playbook logs by node name, making it easier to find specific playbook executions on individual nodes.
  NOTE: Passive nodes are not listed in the **HA Node** filter because playbooks are not triggered on them.

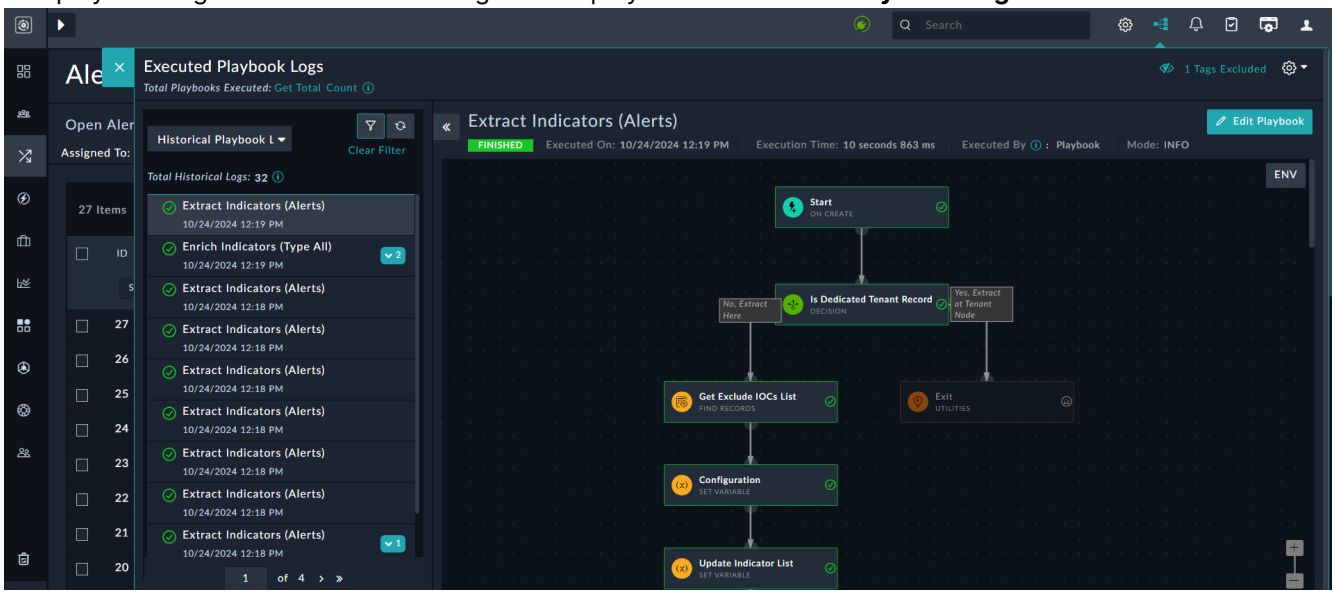
Clicking the **Clear Filter** button removes any applied filters from the playbook execution logs.

You will also see the timestamp when the playbook was executed and the time it took for the playbook to complete its execution. Pagination is supported in the **Executed Playbook Logs** dialog, allowing users to effortlessly navigate through your executed playbook logs and easily find what they need.

To purge Executed Playbook Logs, click the **Settings** icon in the top-right of the `Executed Playbook Logs` dialog and select the **Purge Logs** option. For more information, see Purging Executed Playbook Logs. Large execution histories can increase disk space usage, leading to potential issues. To clean up your workflow execution history use the full vacuum option or advanced PostgreSQL utilities, 'pg_squeeze' and 'pg_repack', for optimal disk space reclamation. For details, see Auto-cleanup of Workflow Execution History.
In release 7.6.1, FortiSOAR introduces 'playbook log movement' to optimize workflow logs storage. This feature automatically moves playbook logs to historical storage after playbook execution is complete. Historical storage supports active storage by retaining only active and awaiting playbooks in the active storage, while completed playbook logs are transferred to historical storage. This process reduces the size of the active storage, enhancing overall performance. Additionally, the reclaim disk space operation runs more efficiently, saving time and system resources due to the reduced table size. Currently, completed playbook logs are moved to historical storage every 15 minutes, while logs for failed or terminated playbooks are transferred every 60 minutes. You can adjust these timings using the options in the **Playbook Log Movement Settings**, as explained in the System Configuration and Management chapter of the "Administration Guide."
In the `Executed Playbook Log` dialog, playbook logs in the active storage are displayed as '**Recent Playbooks Logs**':



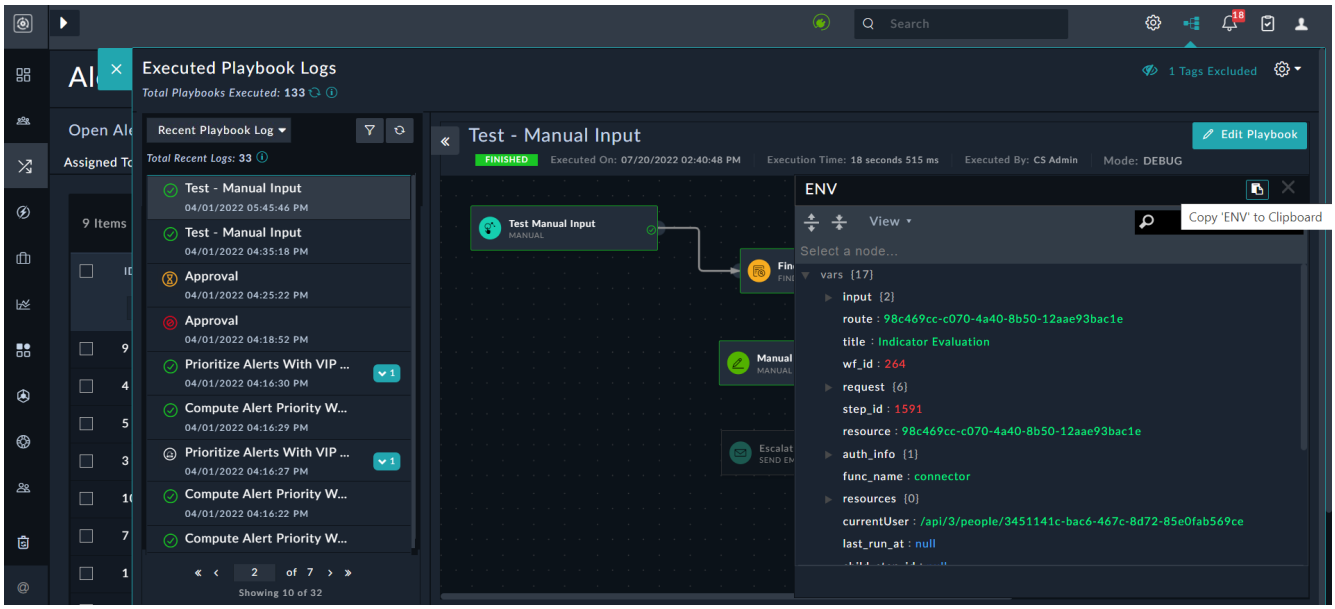
The playbook logs in the historical storage are displayed as '**Historical Playbook Logs**':



To terminate a playbook that are in the **Active**, **Incipient**, or **Awaiting** state, click the **Terminate** button. To terminate all running instances of a particular type, click the **Settings** icon and select the **Terminate Running Instances** option. For more information, see Terminating playbooks.

Environment

Click **Env** to view the complete environmental context in which the playbook was executed, including the input-output and computed variables across all steps in the playbook, if your playbook is executed in the 'Debug' mode. If the playbook is executed in the 'Info' mode it displays the status. In release 7.2.1, the complete JSON tree has a reference of the "vars" root node (earlier `ENV` was written) at the top of JSON), making the writing of jinja in playbooks easier.



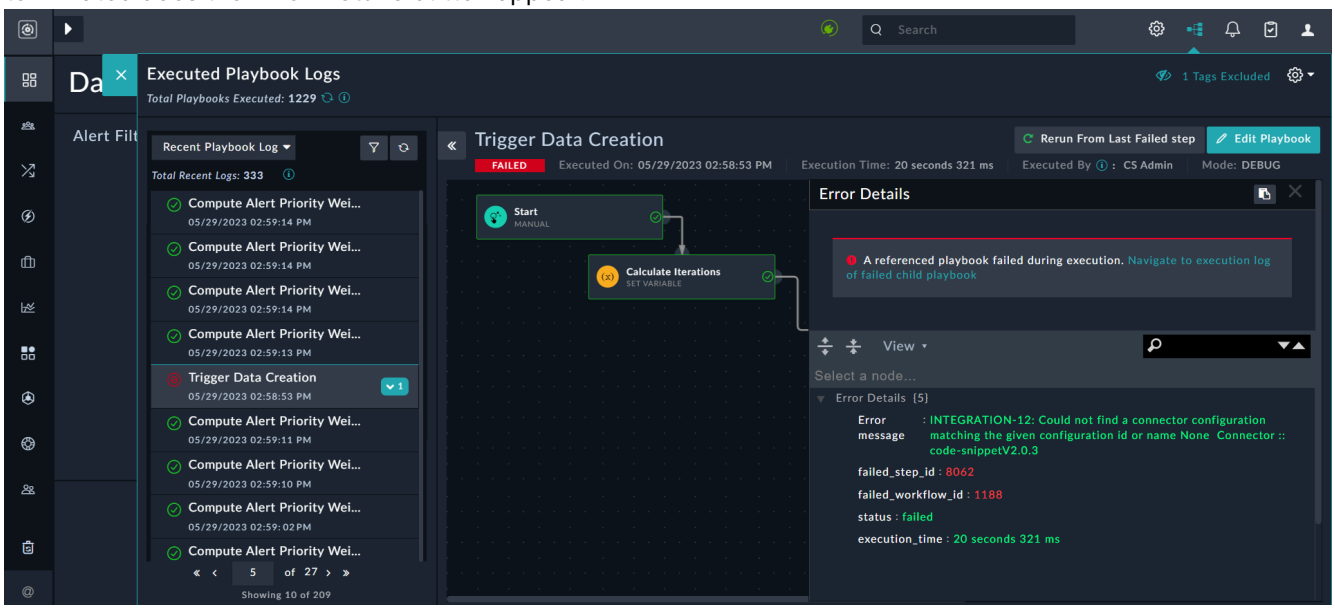
Running huge ingestions and other workflows that load several records into memory can cause the memory usage to be high. In case of connector actions, the 'env' was passed on to each action and it also used to get the 'env' triggered from the workflow (worker 'env'). This unnecessarily increases the memory requirement and limits workflow scaling. Also, connectors only need a minimal information such as `requests headers`, `public-private key`, and `auth info` from the 'env' that can be selectively passed. Therefore, to solve the memory consumption issue, connectors are passed only the required 'env' fields and not the complete environment information. However, if you observe any issues in a connector or you specifically require the complete environment to be passed to the connector, then you need to add the `CONNECTOR_KEEP_COMPLETE_ENV` variable at the end of the `/opt/cyops-workflow/sealab/sealab/settings.py` file, and save the file. Then, you must restart the `celeryd` service using the following command:

```
# sudo systemctl restart celeryd
```

Error Details

Click **Error Details** to view the reason for the failure of a playbook, making it simpler to identify the error's root cause and troubleshoot a playbook failure. Only in the event that a playbook has 'Failed', 'Skipped', or been

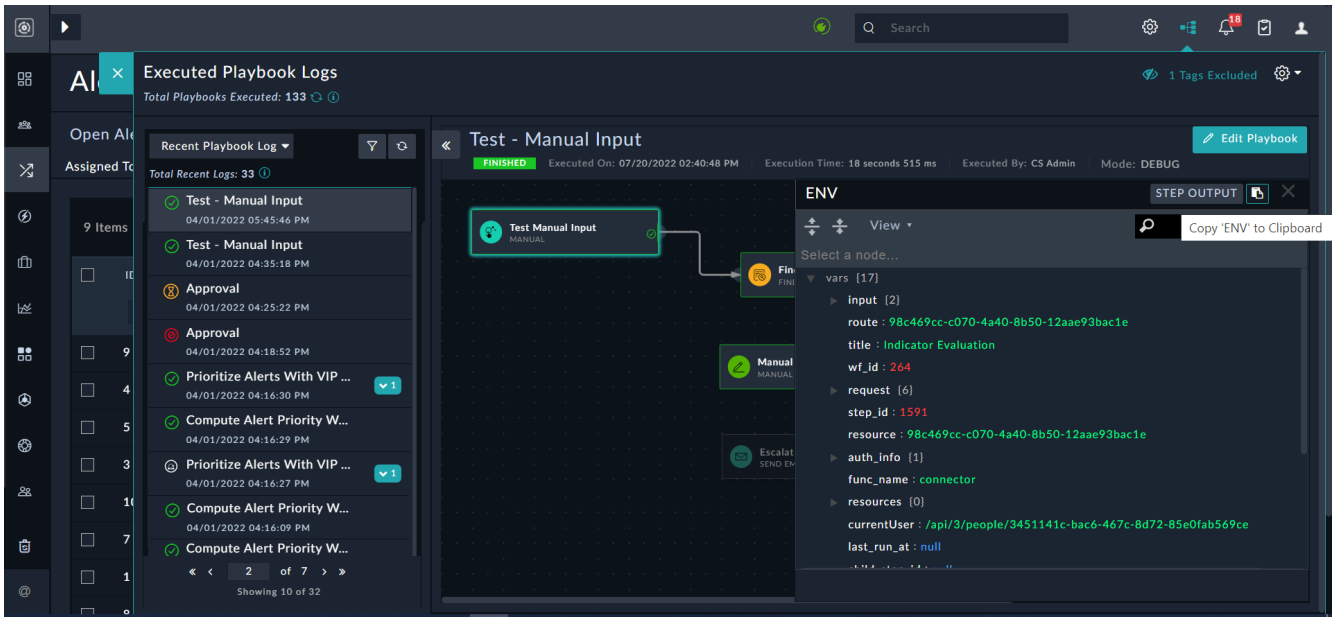
terminated does the **Error Details** button appear:



Error Details in case of 'Terminated' playbooks contain the reason for termination, which is useful for playbooks are terminated for reasons, such as exceeding the permissible time limits set for playbook execution as this helps in the debugging process.

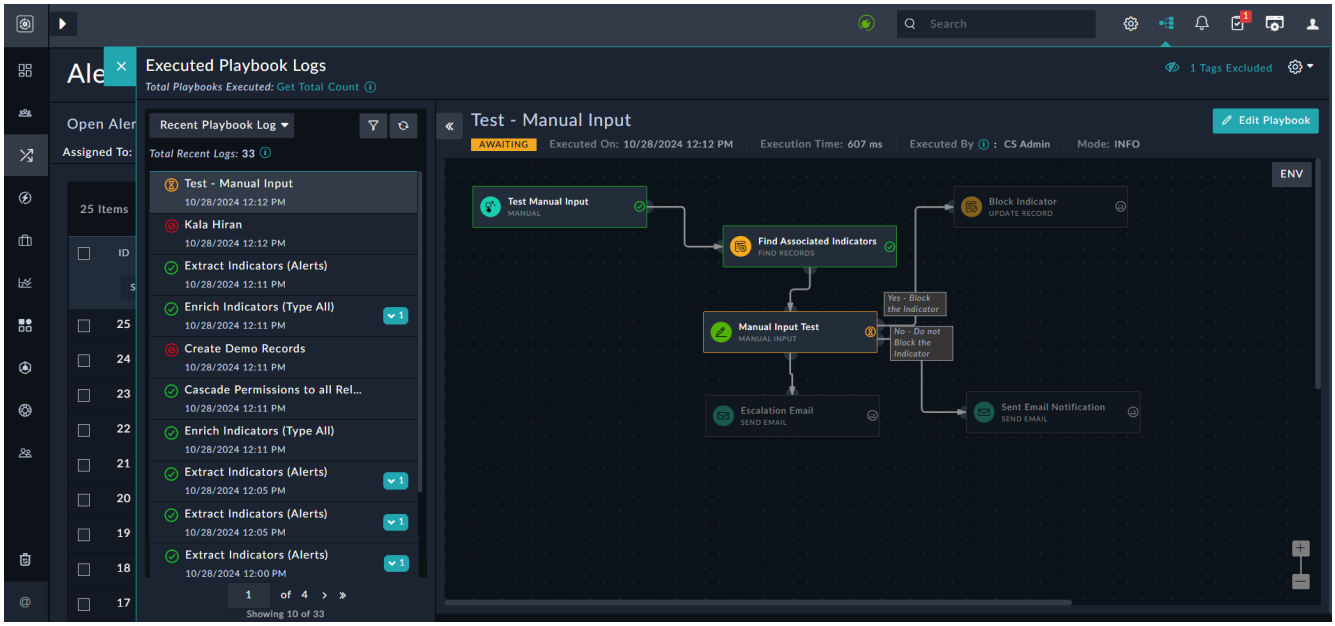
Playbook Steps

Clicking playbook steps display the input, output, and configuration for that step. You can toggle the **ENV** button to toggle between the environment in which the playbook was executed and the steps of the playbook. You can also copy the environment, error, and step details to the clipboard by clicking the **Copy 'ENV' to Clipboard** or **Copy 'OUTPUT' to Clipboard** button.



Clicking playbook Steps section lists all the steps that were part of the playbook and displays the status of each step using icons. The icons indicate whether the step was completed (green tick), skipped (grey skipped symbol), awaiting some action (orange hour glass symbol) or failed (red failed symbol).

For example, if a playbook is awaiting some action, such as waiting for approvals from a person or team who are specified as approvers, then the state of such playbooks is displayed as **Awaiting**.

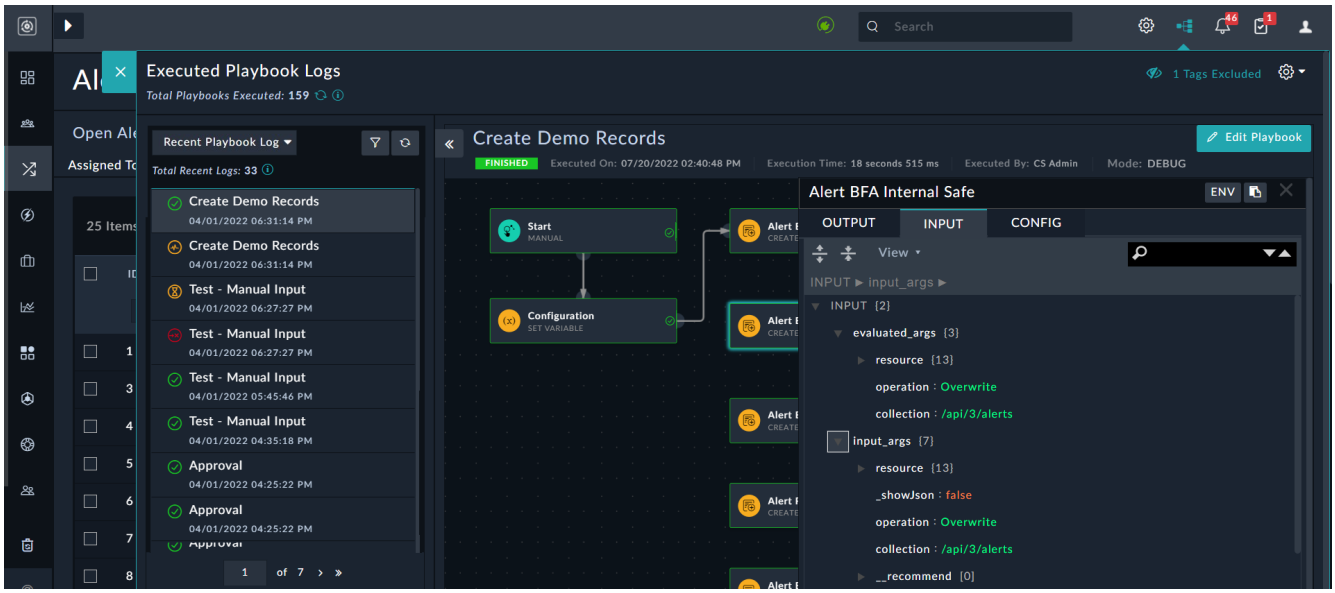


The status of the playbook will display as "Awaiting" till the action for which the playbook execution halted is completed, after which the playbook will move ahead with the workflow as per the specified sequence.

You can click on a playbook step for which you want to view the details, and you will see tabs associated with the playbook step: **Input**, **Pending Inputs** (if the playbook is in the awaiting state), **Output** (if the playbook finishes) or **Error** (if the playbook fails), and **Config**.

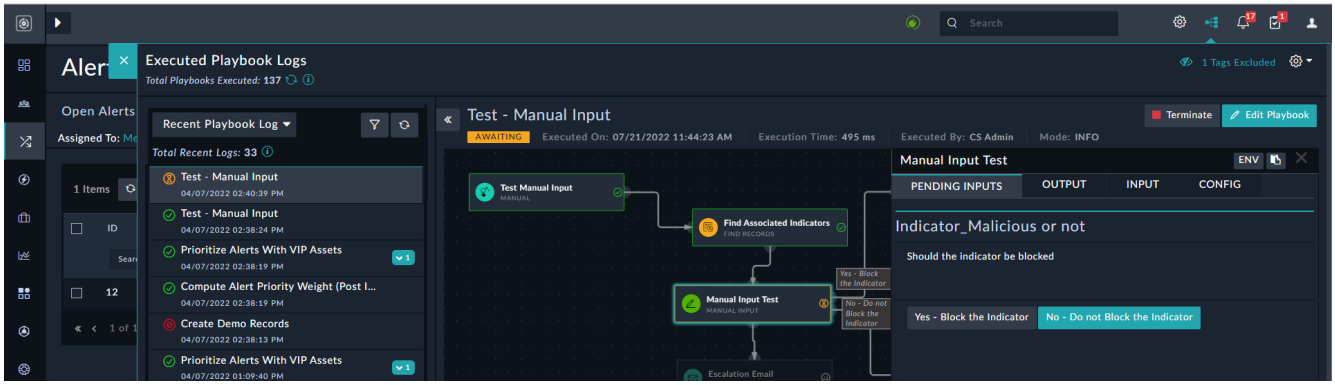
Input Tab

The input tab displays data, in the case of the first step of the playbook such as the `Start` step, input arguments and evaluated arguments. The `data` displays the trigger information for the playbook. The `input_args` displays the input in the jinja format that the user has entered for this step. The `evaluated_args` displays what the user input was evaluated by the playbook once the step gets executed.



Pending Inputs Tab

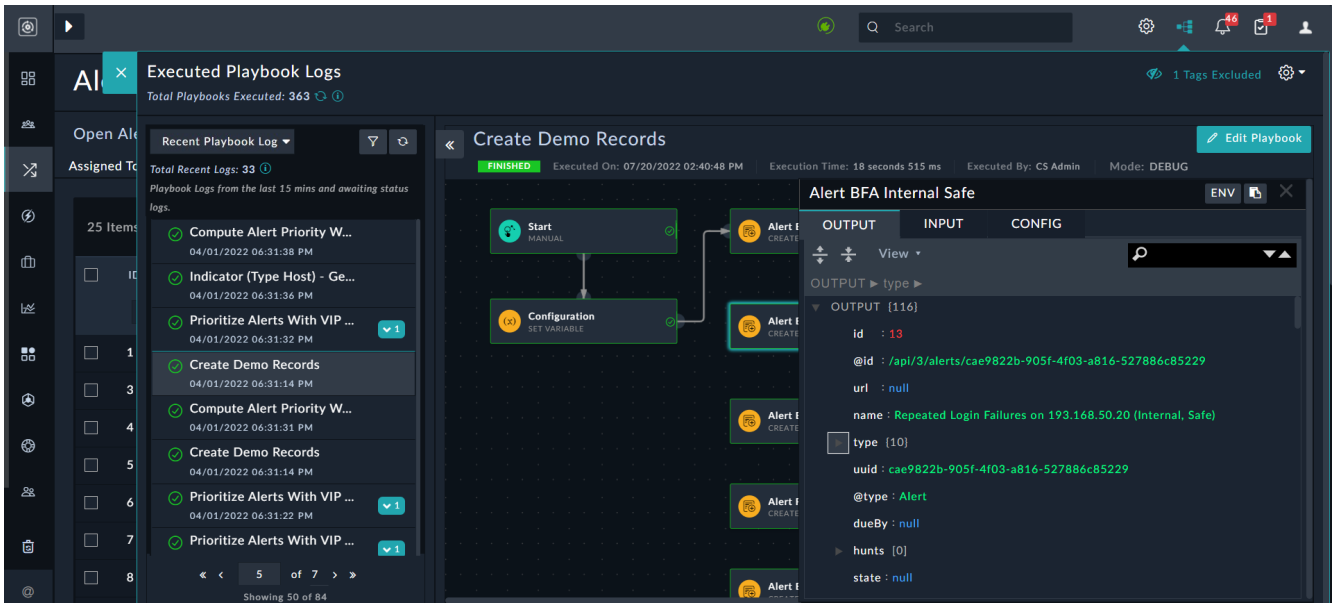
If a playbook is in an "Awaiting" state, i.e., it requires some input or decision from users to continue with its workflow, then the Pending Inputs tab is displayed:



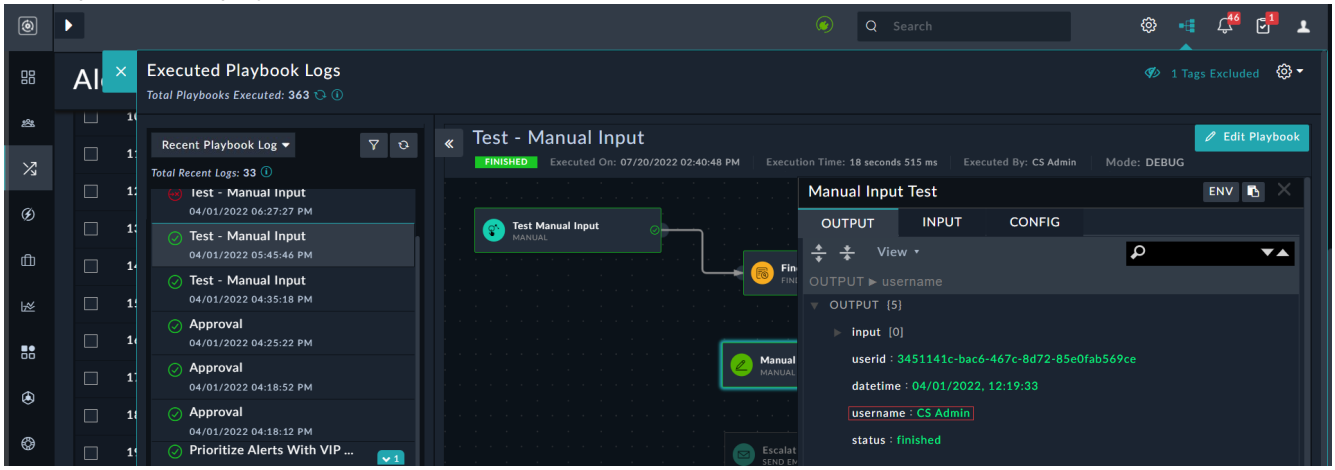
Once the user provides the required inputs and submits their action, the playbook continues its execution as per the defined workflow.

Output or Error Tab

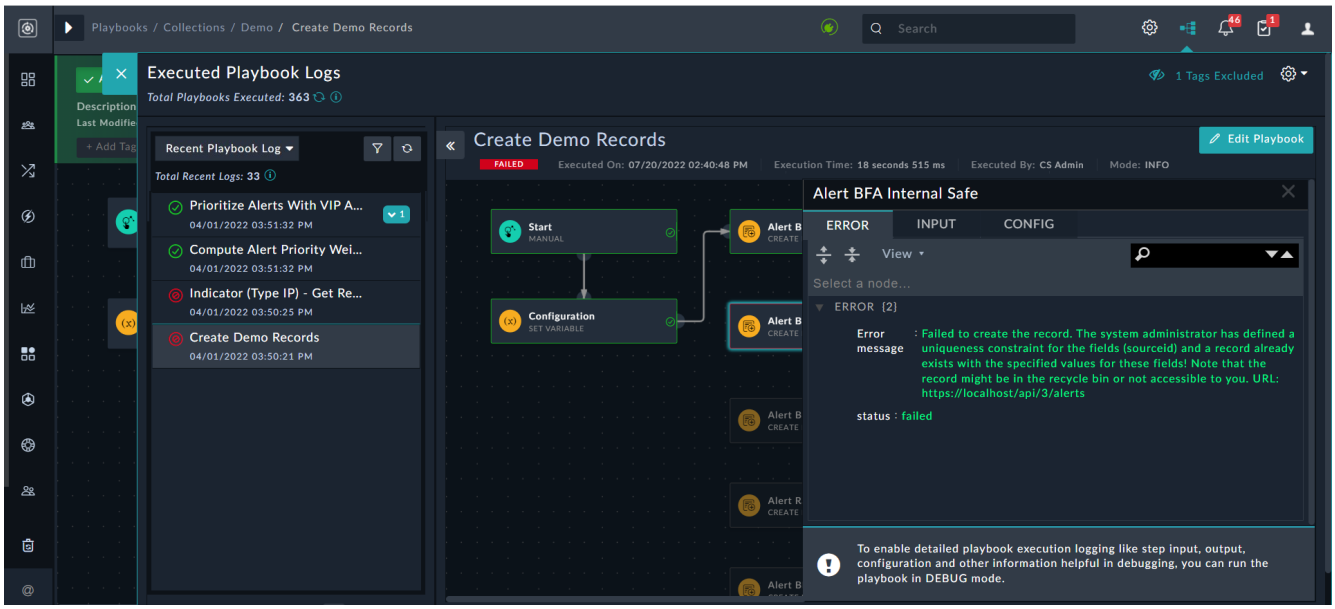
If the playbook step finishes, then the **Output** tab displays the result/output of the playbook step.



From release 7.2.0 onwards, the contents of the output tab have been enhanced in case of manual input playbooks to include the name of the user (`username`) who has taken the decision on manual input leading to the resumption of the playbook:



If the playbook step fails, then the **Error** tab displays the **Error message** for that step. Click the step that has the error (step with a red cross icon) to view the error message, so that it becomes easier for you to know the cause of the error and debug the cause of the playbook failure.



FortiSOAR has enhanced error messages by making them more precise and thereby making it easier for you to debug the issues. Also, the Trace information has been removed from the executed playbook log to reduce the clutter in the error details screen and directly display the exact error. The Trace information will be present in the product logs located at:

- For Playbook runtime issues: `/var/log/cyops/cyops-workflow/celeryd.log`
- For connector issues in cases where playbooks have connectors: `/var/log/cyops/cyops-integrations/connectors.log`

For information about the common playbook error messages and how to debug them, see the Debugging common playbook and connector errors article present in the Fortinet Knowledge Base.

FortiSOAR also provides you with the option to resume the same running instance of a failed playbook from the step at which the playbook step failed, by clicking the **Rerun From Last Failed Step** button. This is useful in cases where the connector is not configured or you have network issues that causes the playbook to fail, since you can resume the same running instance of the playbook once you have configured the connector or resolved the network issues. However, if you change something in the playbook steps, then that would be a rerun of the playbook and not a resume or retry of that playbook.

Users who have `Execute` and `Read` permissions on the `Playbooks` module can rerun playbooks in their own instance. Administrative users who have `Read` permissions on the `Security` module and `Execute` and `Read` permissions on the `Playbooks` module can rerun their own playbooks and also playbooks belonging to users of the same team.

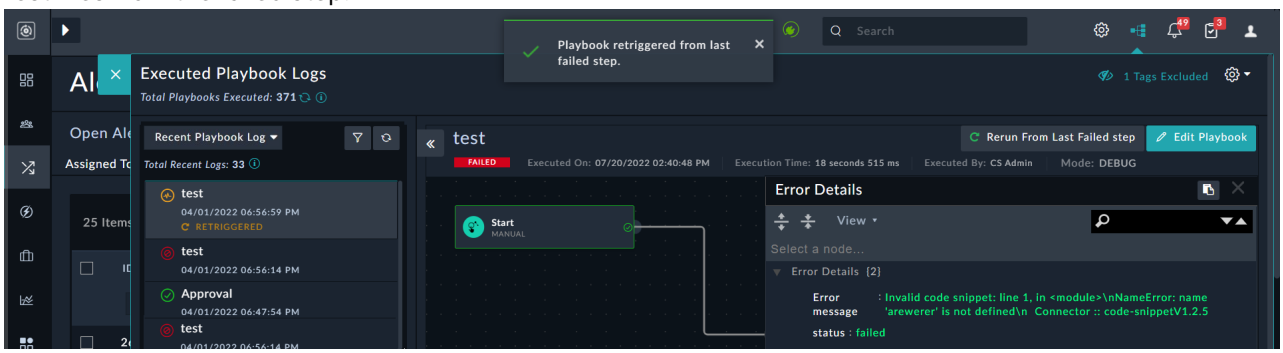
Notes:

- If you have upgraded your FortiSOAR system, then you can resume only those playbooks that were run after the upgrade.
- If you have a playbook that had failed before you upgrade your FortiSOAR system, and post-upgrade you try to resume the execution of that playbook, then that playbook fails to resume its execution.

To resume the running instance of a failed playbook, do the following:

1. Open the `Executed Playbook Logs` dialog.
2. Click the failed playbook that you want to resume, and then click the **Rerun From Last Failed Step** button. FortiSOAR displays the `Playbook retriggered from last failed step` message and the failed playbook

resumes from the failed step:



A playbook that has been rerun will display the **Retriggered** text.

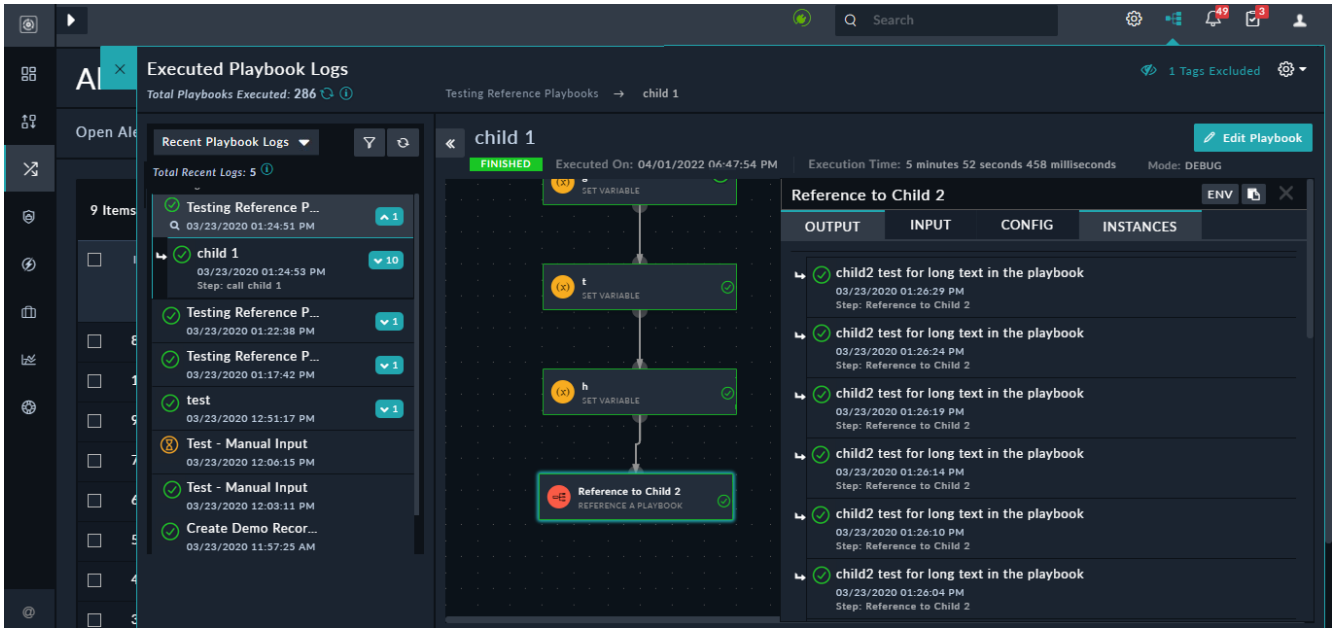
Config Tab

The **Config** tab displays the step variables detailed entered by the user for the particular step and also includes information about whether other variables, such as `ignore_errors`, `MockOutputUsed`, or the `when` condition have been used (`true`/`false`) in the playbook step.

Instances Tab

The **Instances** tab is displayed in case of playbooks containing reference playbook steps. The "Instances" tab allows users to see details such as, name and status of all child instances in a single view.

For example, as displayed in the following image, the "Testing Reference Playbook" references "child 1", which in turn references 10 other child playbooks because of the loop applied on the reference playbook step. When you click the "child 1" step, you can see the "Instances" tab, containing the status of the step "Finished", names of all its child playbooks, and the name of the step that referenced the playbook, which is "Reference to Child 2":



Link to Child Playbooks

Playbooks have been organized by the parent-child relationship. The Parent playbook displays a link that lists the number of child playbook(s) associated with the parent playbook. Clicking the link displays the execution history for the child playbook(s).

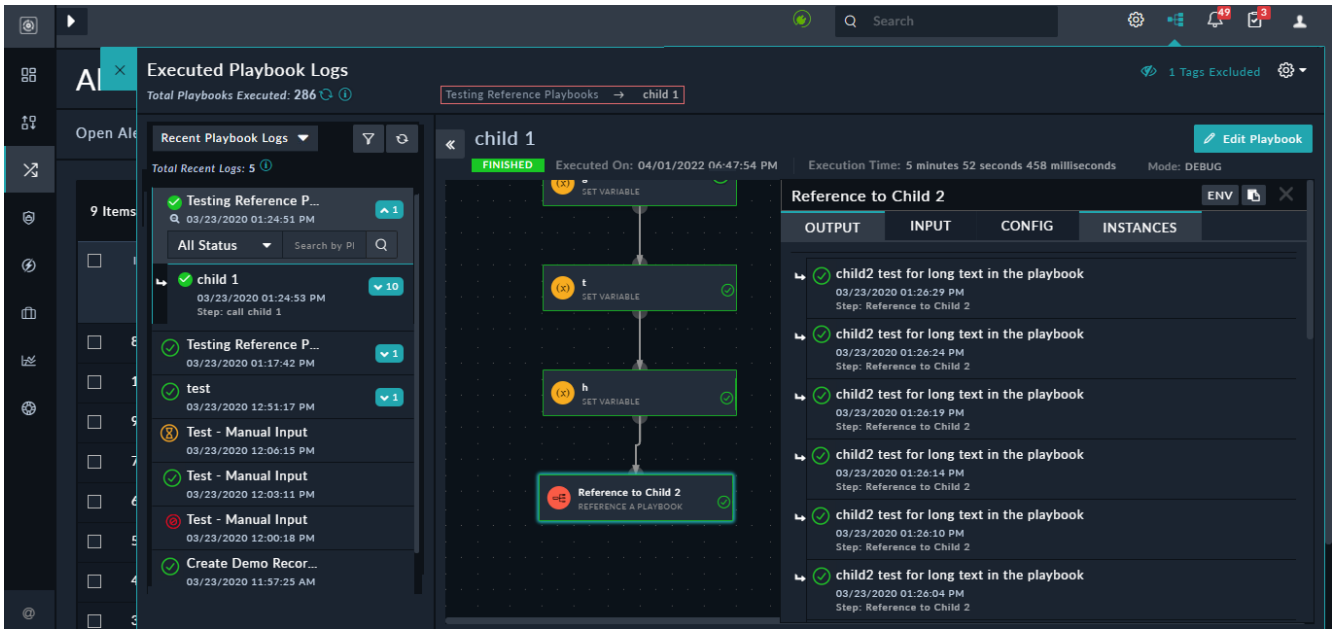
The UI of the execution playbook log displays playbooks that contain various levels of child playbooks in the same visual execution log window. You can click the parent playbook and view its child playbooks, and similarly you can view the children of the child playbook by clicking the child playbook all without losing context of the playbook. You can also see the breadcrumb navigation from parent playbook to the child playbook at the top of the playbook log.

The **Executed Playbook Logs** displays the execution history of the child playbooks, i.e., you can search for the child playbook in Executed Playbooks Logs and the search results will display the child playbook and its execution history and you can also use the Load Env JSON feature in the Jinja Editor making debugging of the child playbooks easier.

Child playbooks inherit the logging level setting from their parent playbook, irrespective of their own setting. For example, if the child playbook's logging level is set as `INFO` and its parent's is set at `DEBUG`; the child playbook's logging level will automatically be set at the `DEBUG` level. For more information on playbook logging levels, and how to set those levels, see the Setting Logging Levels topic.

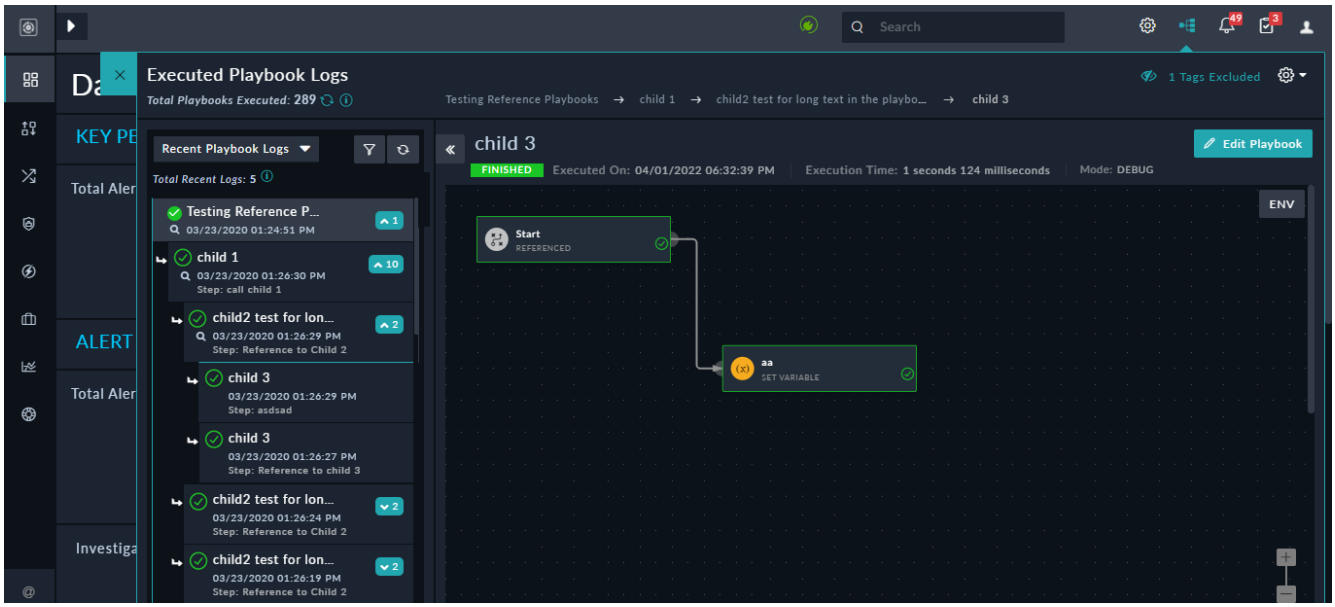
If the parent playbook has a number of child playbooks, you can also search for child playbooks, by clicking the **search** icon that is present beside the child playbook link and then entering the name of the playbook in the **Search by Playbook Name** field. You can also filter the child playbooks on its running status, such as Incipient, Active, Awaiting, etc. by selecting the status from the **All Status** drop-down list.

For example, in the following image, the **Testing Reference Playbook** playbook has 1 child playbook: **child 1**. You can click **child 1** to view its execution history:



As displayed in the above image, you can also see the breadcrumb navigation from parent playbook to the child playbook at the top of the playbook log. You can also view the name of the step at which the child playbook is referenced in the navigator panel. If the playbook contains a reference playbook step then you can click through the child playbooks within the same visual execution log window, allowing you to navigate through the playbook, without losing the context of the playbook. You can also easily navigate back to the parent playbooks using the breadcrumbs present on top of the log.

For example, as displayed in the following image, the "Testing Reference Playbook" contains a child playbook "child 1" at step "call child 1", which in turn contains 10 other child playbooks because of a loop applied on the reference playbook step, such as "child 2 test for long text in playbook", which in turn calls "child 3":



Purging Executed Playbook Logs

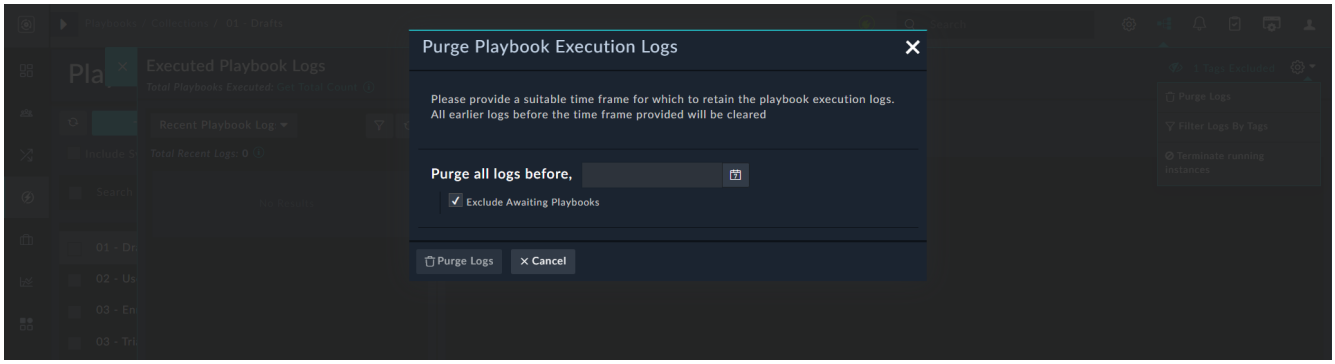
You can purge Executed Playbook Logs by clicking the **Settings** icon on the top-right of the `Executed Playbook Logs` dialog, and then selecting the **Purge Logs** option. Purging executed playbook logs allows you to *permanently* delete old playbook history logs that you do not require and frees up space on your FortiSOAR instance.

Only historical playbook logs will be purged. The purge function excludes 'Recent' playbook logs and playbooks executed on the same day from the 'Historical' logs. Once purged, logs are permanently deleted and cannot be restored. Ensure that you no longer need them before proceeding with the purge.

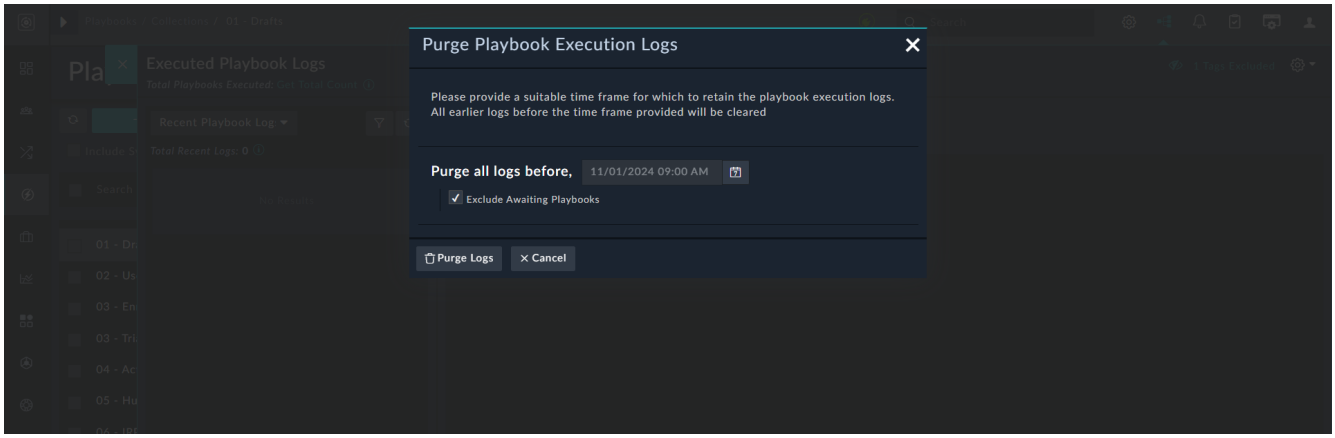
You can also schedule purging, on a global level, for both audit logs and executed playbook logs. For information on scheduling Audit Logs and Executed Playbook Logs, see the `Configuring purge settings for audit logs, executed playbook logs, and recycle bin records, and reclaiming unused disk space` topic in the System Configuration and Management chapter of the "Administration Guide."

To purge Executed Playbook Logs, you must be assigned a role that has a minimum of `Read` permission on the `Security` module and `Delete` permissions on the `Playbooks` module.

To purge Executed Playbook Logs, click the **Settings** icon and select the **Purge Logs** option, which displays the `Purge Playbook Execution Logs` dialog:

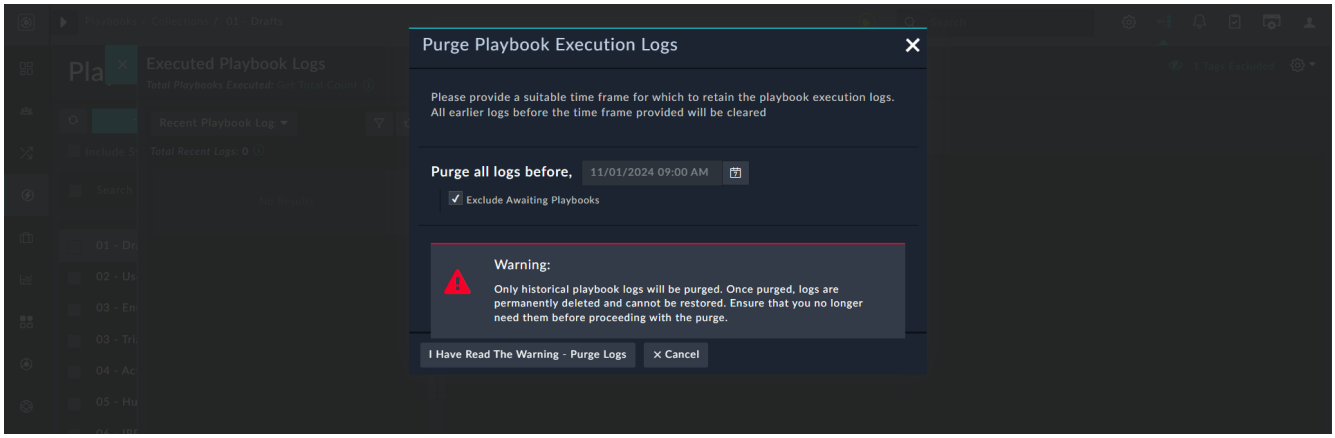


In the **Purge All logs before,** field, select the time frame (using the calendar widget) before which you want to clear all the executed playbook logs. For example, if you want to clear all executed playbook logs before `November 01st, 2024, 9:00 AM`, then select this date and time using the calendar widget.



Click the **Exclude Awaiting Playbooks** checkbox (default) to exclude the playbooks that are in the "Awaiting" state from the purging process.

To purge the logs, click the **Purge Logs** button, which displays a warning as shown in the following image:



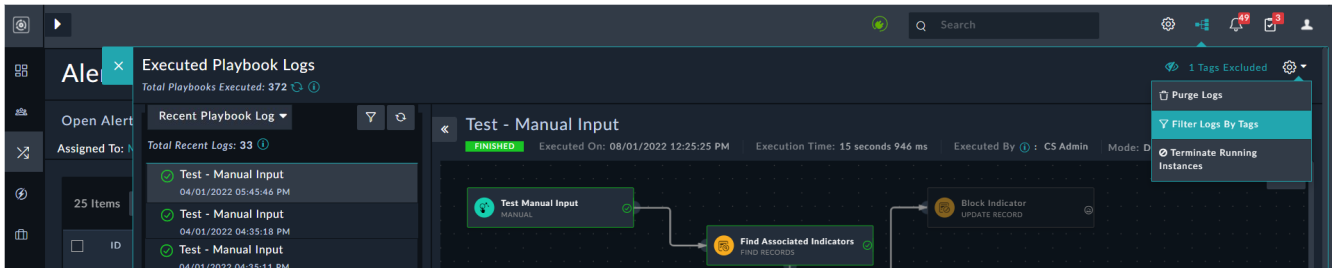
Click the **I Have Read the warning - Purge Logs** to continue the purging process.

Filtering playbook logs by tags

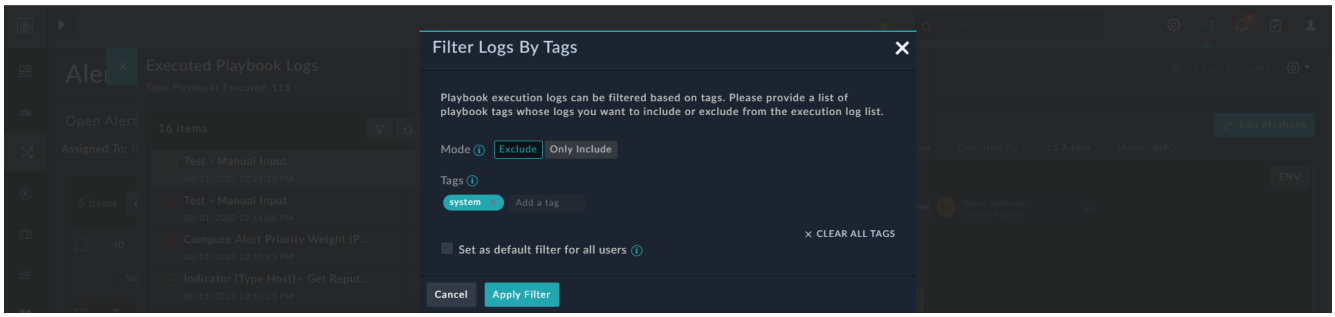
You can filter playbook execution logs by tags or keywords that you have added in your playbooks.

A user who has a role with a minimum of `Update` permission on the `Security` module can save tags, which will be applied as a default filter for playbook execution logs to all other user. A user who does not have such a role can add a tag to filter playbook execution logs and view the filtered playbook execution logs but cannot save that filter.

Click the **Settings** icon on the top-right of the `Executed Playbook Logs` dialog to view tags that have been added by default to filter the playbook execution logs. You can see a message `1 Tags Excluded`, which means that playbook logs with one specific tag is being excluded by default.



You can either click the **1 Tags Excluded** link or the **Filter Logs By Tags** option to open the `Filter Logs by Tags` popup as shown in the following image:



To filter playbook logs based on tags, add a comma-separated list of tags in the **Tags** field.

In the `Mode` section, choose **Exclude** to exclude playbook logs with the specified tags. You will observe that the `#system` tag is already added as a tag in the Exclude mode, which means the any playbook with the `system` tag will be excluded from the playbook logs. To include only those playbook logs with the specified tags, click **Only Include**. For example, if you only want to view the logs of phishing playbooks, i.e., logs of playbooks that have `phishing` tag, click **Only Include** and type `phishing` in the **Tags** field. You must also remove the **system** tag from the **Only Include** mode, since otherwise playbook logs with both the `phishing` and `system` tags will be included.

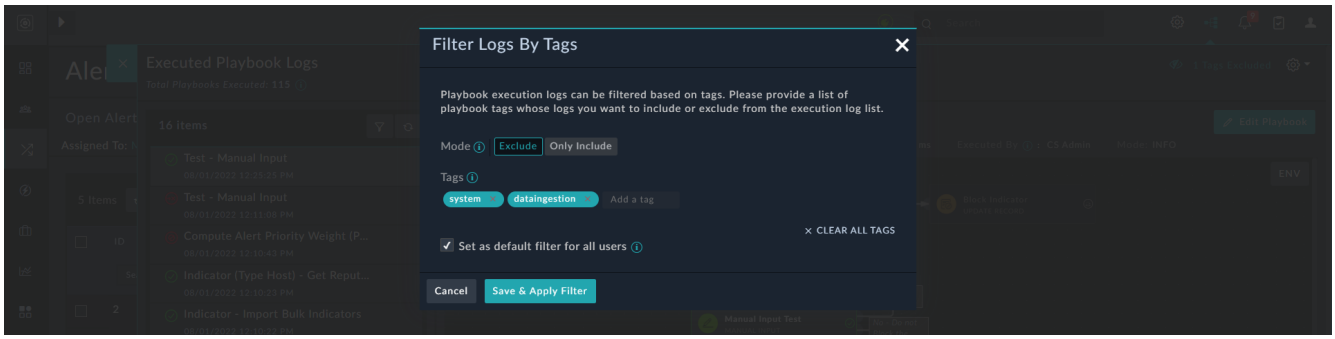
You can specify a comma-separated list to Include all tags or Exclude all tags. You cannot have a mix of Include and Exclude tags.

Filters will apply from the time you have set the filter, i.e., if you have added a `phishing` tag in the **Exclude** list at 16/05/2019 17:00 hours, then the filter will apply only from this time. The historical logs, logs before 16/05/2019 17:00 hours will continue to display in the Executed Playbooks Logs.

From version 7.0.1 onwards, the settings of the playbook execution history logs that are filtered using tags have been updated as follows:

- The 'global' filter will be applicable on only on the global execution log list.
- Record level playbook execution history log will show all playbook logs by default; users can apply temporary filters to filter the result. The **Set as default filter for all users** option has been removed from the record level playbook execution log filter settings.
- Playbook-level playbook execution log will show all logs. When you open a playbook in the playbook designer and view its execution history, you will see all the logs. There is no UI option in playbook execution history to filter the logs. For example, the default exclude filter that is applied for '`system`' playbooks will not be applicable when you open the playbook execution log for a specific playbook by clicking **Tools** > **Execution History** in the playbook designer.

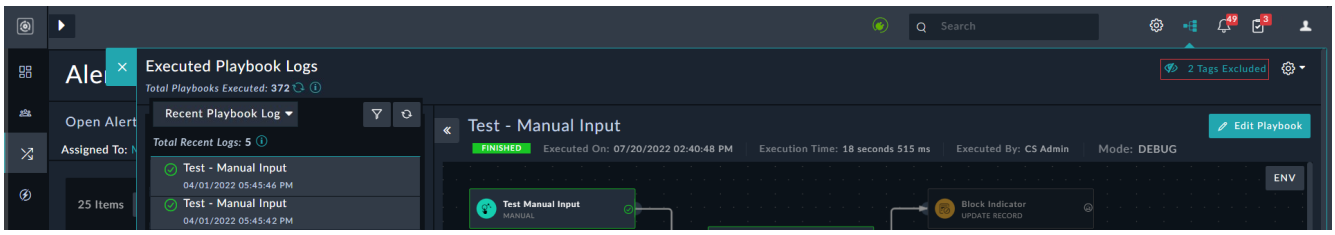
An example of excluding playbook logs by tag follows:
If you have added tags such as `dataIngestion` in your playbooks, then you can filter out the data ingestion logs by clicking **Exclude** and typing `dataIngestion` in the **Tags** field. If an administrator with `Update` rights on the `Security` module wants this filter to be visible to all users, then the administrator can save this filter as a default for all users, by clicking the **Set as default filter for all users** checkbox and then clicking the **Save & Apply Filter** button.



From version 7.0.1, the **Set as default filter for all users** checkbox has been removed from the record level playbook execution log filter settings.

If you do not have appropriate rights, you can apply the filter for only yourself by clicking the **Apply Filter** button and view the filtered playbook executed logs.

This applies the filter and displays text such as **2 Tags Excluded** on the top-right corner of the Executed Playbook Logs dialog. Now, the Executed Playbook Logs will not display logs for any *system* playbook or for any *data ingestion* playbook.



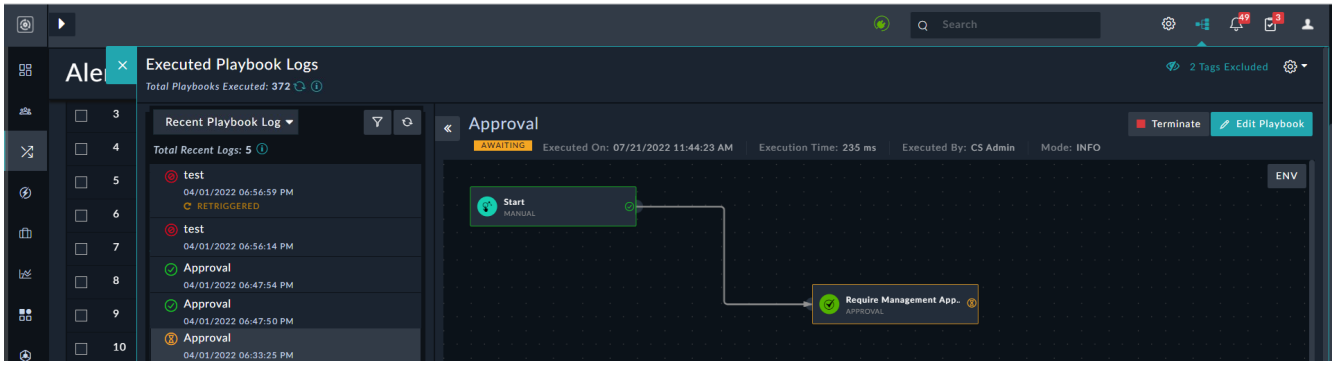
Users (without administrative rights) can remove filters by clicking **Settings** > **Filter Logs by Tags** or clicking the `<number of Tags included>` link to display the `Filter Logs By Tags` dialog and click **Clear All Tags** to

remove the tags added and add their own tags. However, these changes will only be applicable till that instance of the log window is open. If the page refreshes or the window reloads, then the tags specified by the administrator will again be applied.

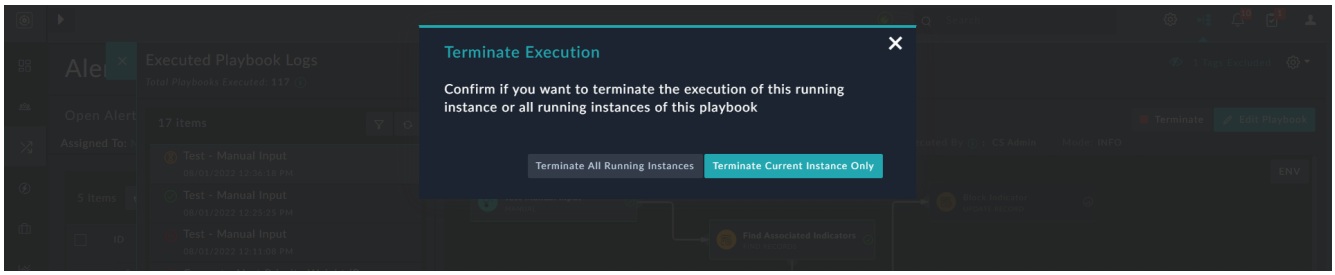
Terminating playbooks

You can terminate playbooks that are in the **Active**, **Incipient**, or **Awaiting** state. Users who have `Read` and `Execute` permissions on the `Playbooks` module can terminate a running instance of their own playbook instance. Administrators who have `Read` permissions on the `Security` module and `Execute` permissions on the `Playbooks` module can terminate running instances of any playbook.

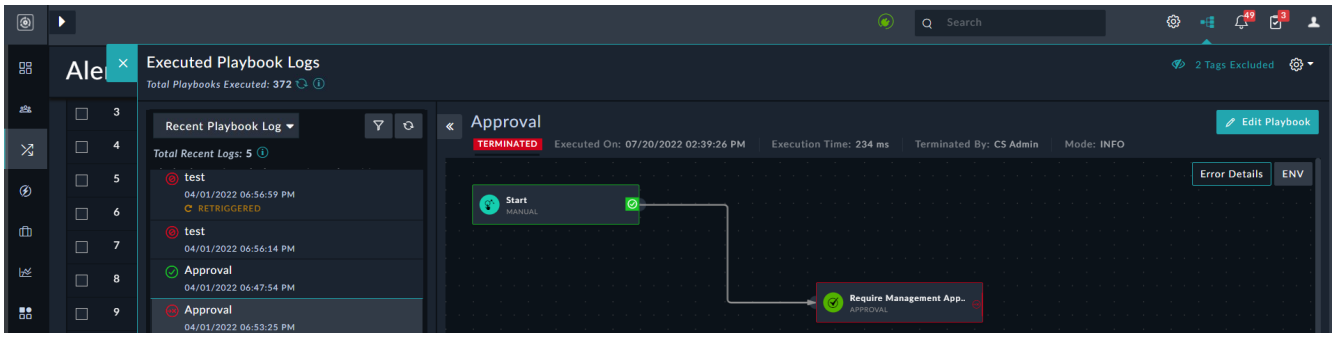
To terminate a running playbook instance, open the Executed Playbook Logs and click the instance that you want to terminate and click **Terminate** as shown in the following image:



Once you click **Terminate**, the `Terminate Execution` dialog is displayed in which you can choose to either terminate only the particular running instance, by clicking **Terminate Current Instance Only** or terminate all running instances, by clicking **Terminate All Running Instances**.



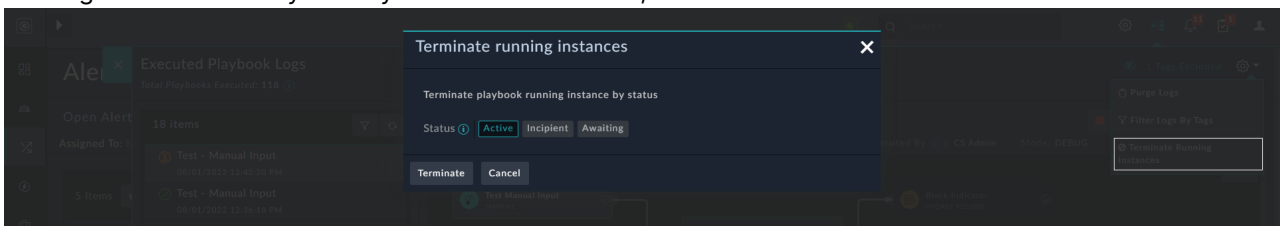
If you click **Terminate Running Instance Only**, then the state of that playbook changes to **Terminated**:



You can also choose to terminate the running instances of all playbooks that are in the Active, Incipient, or Awaiting state.

To terminate the running instances of all playbooks based on the status of the playbooks, do the following:

1. Click the **Settings** icon on the top-right of the `Executed Playbook Logs` dialog.
2. Select the **Terminate Running Instances** option, which displays the `Terminate Running Instances` dialog.
3. In the `Terminate Running Instances` dialog, select the status (**Active**, **Incipient**, or **Awaiting**) whose running instances of Playbooks you want to terminate, and click **Terminate**.



You can rerun the playbook from the step it was terminated by clicking the **Rerun Pending Steps** button on the terminated playbook.

Auto-Cleanup of Workflow Execution History

Workflow execution history is extensively persisted in the database for debugging and validating the input and output of playbooks at each step. However, a large execution history can lead to issues such as increased disk space usage and longer upgrade times. To address this, you can cleanup your workflow execution history using full vacuum or, starting from release 7.6.0, utilize advanced PostgreSQL utilities, 'pg_squeeze' and 'pg_repack', for optimizing disk space reclamation.

Cleaning up your workflow execution history using 'pg_squeeze' and 'pg_repack'

The configuration file (`/opt/cyops-workflow/sealab/sealab/config.ini`) contains keys (`USE_PG_SQUEEZE` and `USE_PG_REPACK`) to specify whether to use full vacuum or the advanced utilities.

If you update any parameter in the `config.ini`, you need to restart the services using the `sudo systemctl restart uwsgi celeryd fsr-workflow` command.

In fresh installations from release 7.6.0 onwards, the default settings for 'USE_PG_SQUEEZE' and 'USE_PG_REPACK' parameters are set to '**true**'. However, in upgrades to release 7.6.0, the 'USE_PG_SQUEEZE' and 'USE_PG_REPACK' keys are set to '`false`', which means that the upgraded systems will still use full vacuum for cleaning up workflow execution history, which can block of all workflow operations. Before changing the workflow execution cleanup behavior by setting the 'USE_PG_SQUEEZE' and 'USE_PG_REPACK' keys to '`true`', for upgraded instances, it is recommended to first run a full vacuum.
To modify the default settings for the 'USE_PG_SQUEEZE' and 'USE_PG_REPACK' parameters you need to edit the

`config.ini` file and then restart the services by using the `sudo systemctl restart uwsgi celeryd fsr-workflow celerybeatd` command.

Additionally, an audit entry is generated each time the 'pg_squeeze' and 'pg_repack' tasks are initiated, and another entry is added upon completion of the task, indicating success or failure. The audit log will include details such as the total time taken, number of retries, and space reduction achieved in case of success.

If you want to use the pg_squeeze' and 'pg_repack' utilities in the case of an externalized FortiSOAR PostgreSQL database, additional steps need to be performed, see the Externalization of your FortiSOAR PostgreSQL database topic in the "Best Practices Guide" for details.

Troubleshooting 'pg_squeeze' and 'pg_repack' errors

The following table list errors you might encounter while using 'pg_squeeze' and 'pg_repack' and steps to resolve these errors:

Error	Troubleshooting steps
Duplicate key value violates unique constraint "pg_replication_origin_<name>"	1. Get data from pg_replication_origin: *Query* : `select * from pg_replication_origin;` 2. Get the 'roname' column data for squeez- related data. Roname start with "`pf_squeeze_slot`" 3. Delete the above slots *Query* : `select pg_replication_origin_drop('origin_<slot_name>);`
The table "public.<table_name>" already has a trigger called "repack_trigger"	Delete and recreate the 'pg-repack' extension *Query*: 1. `DROP EXTENSION IF EXISTS pg_repack CASCADE;` 2. `CREATE EXTENSION pg_repack;`
The PostgreSQL service stops when running advanced utilities, i.e., 'pg_squeeze' or 'pg_repack'	Wait for about five seconds and then restart the PostgreSQL service using the following command: `sudo systemctl restart postgresql-<postgresql version>`
UTCERROR: temporary file size exceeds temp_file_limit (5242880kB)	1. Open the `/var/lib/pgsql/16/data/postgresql.conf` file. 2. Double the size of the `temp_file_limit` parameter. 3. Restart the PostgreSQL service using the following command: `sudo systemctl restart postgresql-<postgresql version>` If the previous steps do not resolve the issue, 1. Set the value the `temp_file_limit` parameter to `-1`. This allows the 'pg_squeeze' temporary tables to occupy all the size allocated to PostgreSQL for the operation. 2. Restart the PostgreSQL service using the following command: `sudo systemctl restart postgresql-<postgresql version>`

Cleaning up your workflow execution history using full vacuum

If you choose to use full vacuum, then it highly recommended to set up an auto-cleanup of the workflow execution history using a weekly cron schedule.

To delete the workflow run history keeping the last 'X' entries, `ssh` to your FortiSOAR VM and run the following command:

```
# sudo /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py cleandb --keep
X
```

For example, to delete all workflow run history, apart from the last 1000 entries, use the following command:

```
# sudo /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py cleandb --keep
1000
```

To set up a weekly schedule delete workflow history, to the above command, add a cron expression entry in the `/etc/crontab` file that would schedule a workflow execution history cleanup as per your requirements. Command to edit a cron job is `sudo crontab -e`.

For example, the command to add an entry in the `/etc/crontab` file that would schedule a workflow execution history cleanup to every Saturday night and delete all workflow run history, apart from the last 1000 entries, would be as follows:

```
sudo vi /etc/crontab
0 0 * * SAT /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py cleandb --
keep 1000
```

Note that running the above command deletes the workflow entries but does not release the disk space back to the OS, i.e., it keeps it reserved for the Postgres process. This is the desired behavior, and no further action is required if the execution history cleanup is scheduled because the Postgres process would need the freed-up disk space to store further workflows. If, however, you also wish to reclaim disk space for backup or restore or other activities, you would additionally need to run a "full vacuum" on the database after you `ssh` to your FortiSOAR VM and run the following commands:

```
sudo psql -U cyberpgsql sealab
psql (10.3)
Type "help" for help.

sealab=# vacuum full;
VACUUM
sealab=# \q
```

Known Issue: If you do not schedule the workflow execution cleanup, and you are deleting a very large set of entries in one go, then the `db cleanup` command might fail due to the failure in loading the large set of entries into memory. In this case, you will have to run the command in batches.

For example:

```
# sudo /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py cleandb --keep
100000
# sudo /opt/cyops-workflow/.env/bin/python /opt/cyops-workflow/sealab/manage.py cleandb --keep
90000
```

Advanced Playbook Development

FortiSOAR supports a variety of Jinja filters that can be used to manipulate, access, and transform data within playbooks. These filters can help with data extraction, transformation, and make your playbooks more dynamic and flexible.

Reserved Keywords

Following is the list of reserved words that must not be used as variable names:

- 'items'
- 'result'
- 'input'
- 'request
- 'values'
- 'keys'
- 'files'
- 'env'
- 'message'
- 'resources'
- 'step_variables'
- 'do_until'
- 'ignore_errors'
- 'when'
- 'for_each'
- 'cyops_playbook_iri'
- 'cyops_playbook_name'
- 'collaborationNote'
- 'inputVariables'
- 'displayConditions'
- 'task_id'
- 'wf_id'

Dynamic Values

Use Dynamic Values to generate Jinja dynamically within the Playbook Designer. To make your playbook dynamic use Jinja templates to define various conditions within steps in a playbook. However, you must have some knowledge of Jinja (see Jinja Documentation), and must understand the workflow of the playbook steps in the JSON format to create Jinja templates.

Using the Jinja editor, you can apply a Jinja template to a JSON input and then render the output, thereby checking the validity of the Jinja and the output before you add the Jinja to the Playbook.

Using Dynamic Values, you can dynamically add Jinja to steps within a Playbook. Click a step, within the playbook that takes Jinja as an input and Dynamic Values is displayed. Choose from the options presented to add Jinja to the step. In FortiSOAR release 7.4.0, the Dynamic Values window has been redesigned for ease of usability and searchability. Ease of usability includes displaying of 'keys' instead of the complete Jinja expression in fields that use the Dynamic Values options by introducing a 'Simplified Expression View' setting. This setting renders a tag-based simplified expression in the playbook designer instead of the complete Jinja expression. However, if your administrator disables this setting, then complete Jinja expressions are displayed in the playbook designer. *Note that this chapter aims in giving examples of usage of Jinja Expressions in playbooks, and therefore, complete Jinja expressions are displayed in the playbook steps*. For information on the 'Simplified Expression View' and some examples of tag-based simplified expressions, see the Simplified Expression View topic.

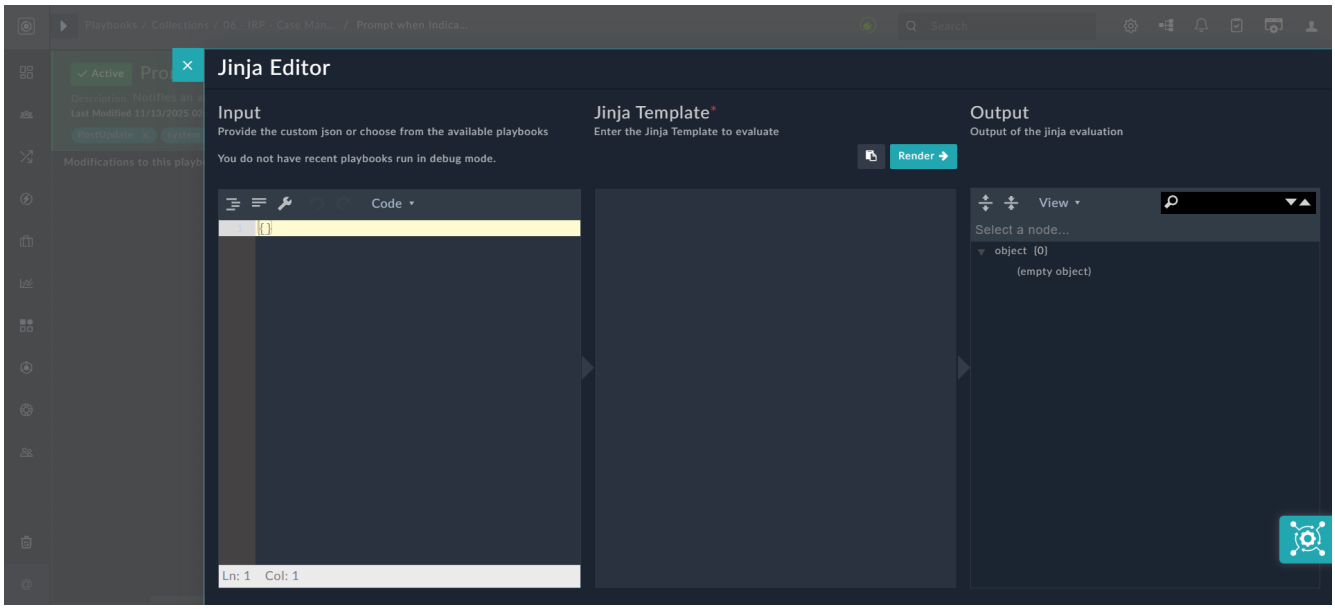
Variable names can contain letters, numbers, and underscores only.

FortiSOAR simplifies the process of building playbooks without requiring to have in-depth Jinja or Python knowledge. Use the "Functions" tab on Dynamic Values to build playbooks with medium-level complexity without a lot of programming experience, with the option to use Jinja or Code Snippets to build playbooks that are very complex. For more information, see Functions.

Jinja Editor

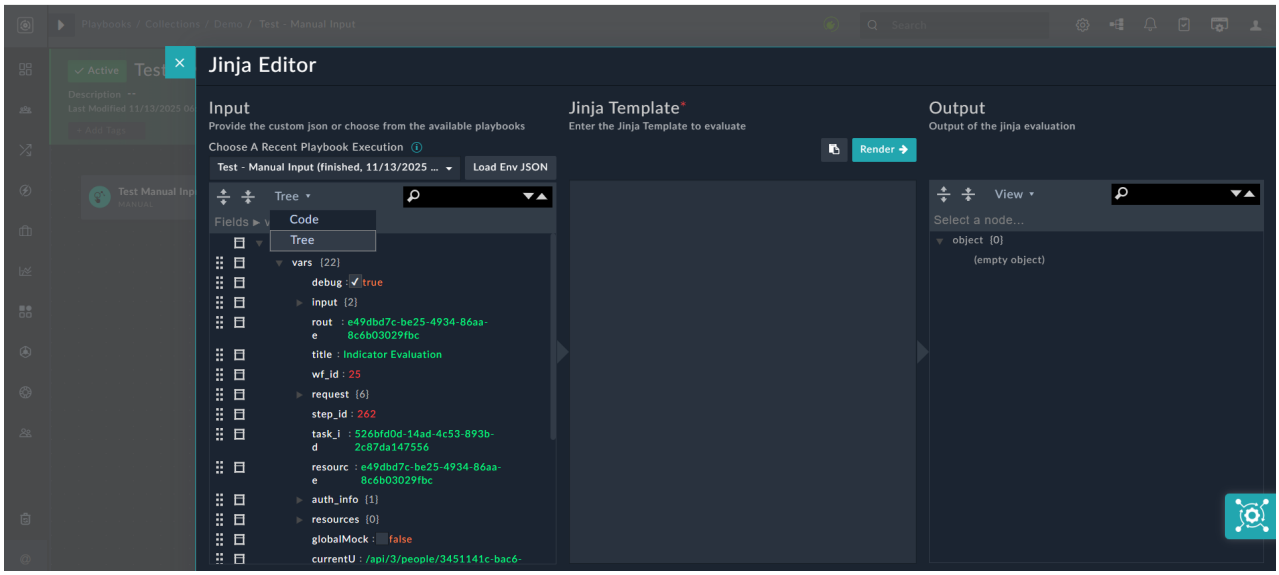
Use the Jinja Editor to apply a Jinja template to a JSON input and render the output. You can validate the Jinja syntax and view the output before adding the Jinja template to the Playbook.

To open the Jinja Editor, click **Tools** > **Jinja Editor** in the Playbook Designer:



The Jinja Editor consists of three areas: Input, Jinja Template, and Output.

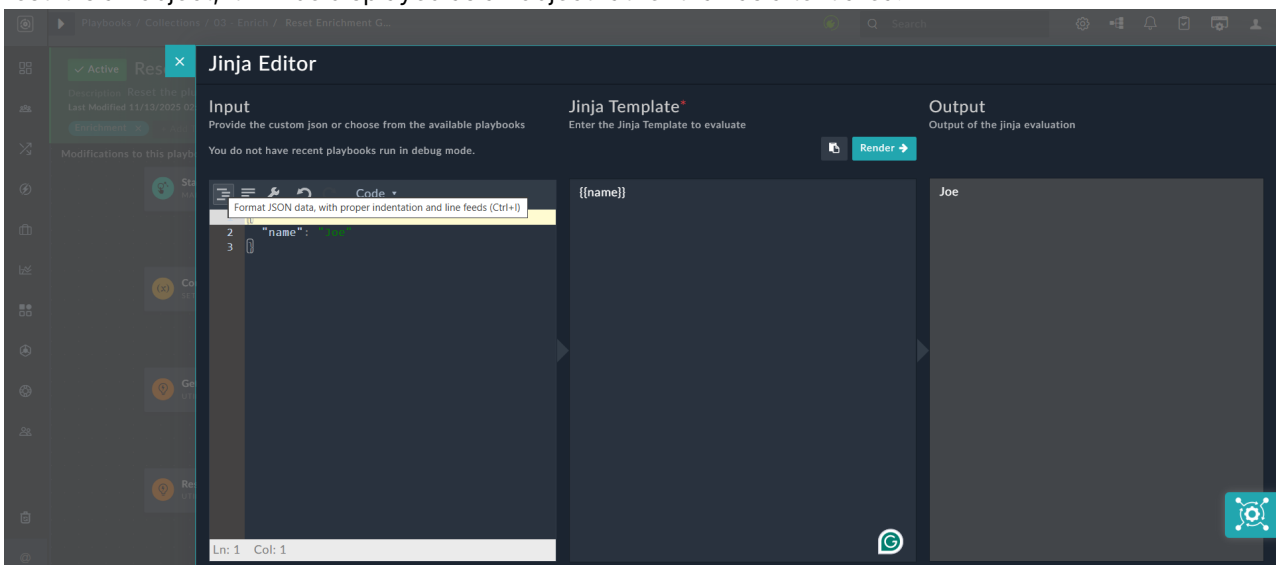
- **Input**: In the `Input` area, provide the JSON input in a `"Key":"Value"` pair format. The Jinja editor will display a `"Bad String"` prompt if there are syntax errors in the JSON. You can also define nested key-value pairs. The JSON data can be viewed in either '**Tree**' view or '**Code**' view.
  In '**Tree**' view, you can expand or collapse fields and search through the JSON data:

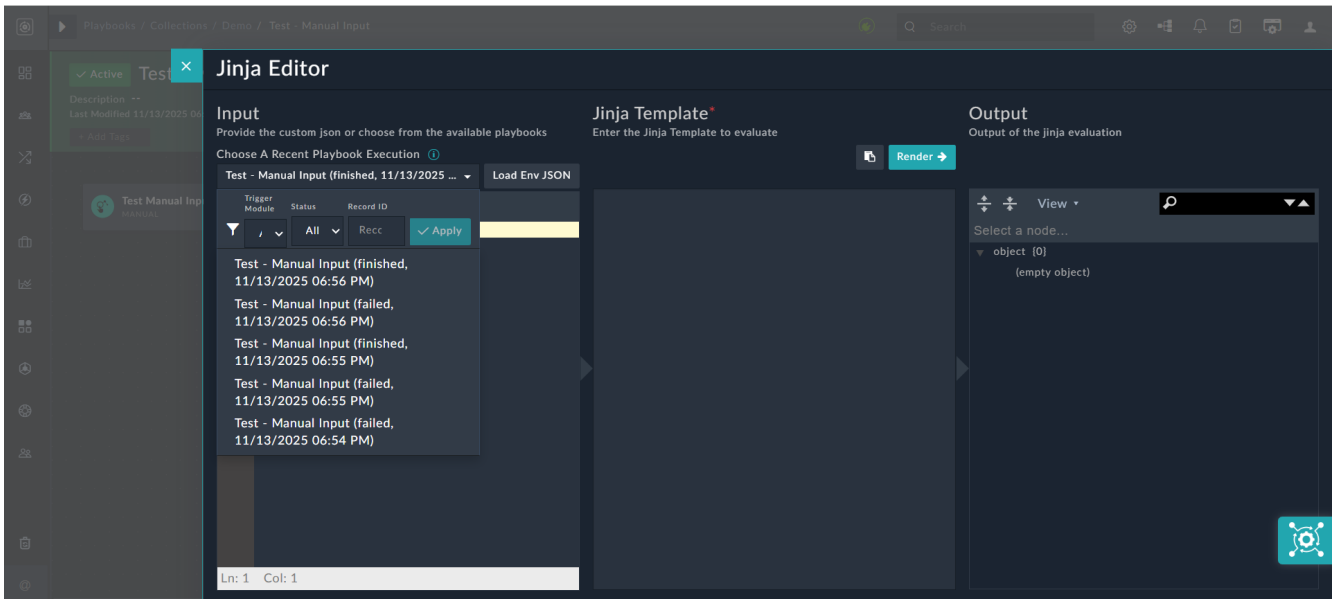
  The top bar in the JSON area in **Code** view includes the following buttons:

  - **Format JSON Data**: Formats the data with proper indentation and line breaks.
  - **Compact JSON Data**: Removes unnecessary whitespace and compacts the data.
  - **Repair JSON**: Fixes issues in the JSON, such as correcting escape characters, quotes, and removing comments.
  - **Undo**: Reverts the last action.
  - **Redo**: Redoes the last undone action.
    After entering the Jinja template and JSON, click **Render** to display the output.

- **Jinja Template**: In the `Jinja Template` area, specify the Jinja code inside *curly* brackets.
- **Output**: The `Output` area shows the result generated by combining the Jinja template and JSON input. If the result is an object, it will be displayed as an object rather than as a text area.

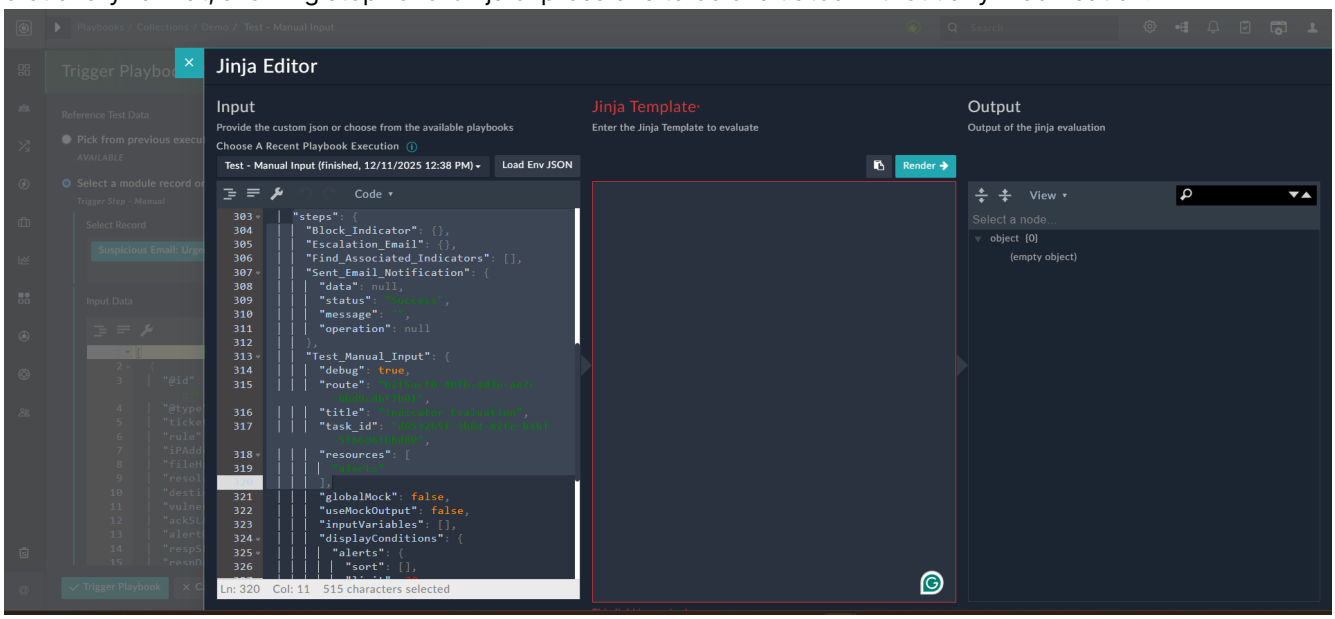


If the playbook has been executed in the 'DEBUG' mode (see the Setting the logging levels for playbooks topic for details on changing modes), you will see a **Choose A Recent Playbook Execution** drop-down list displaying the 30 most recent executions:

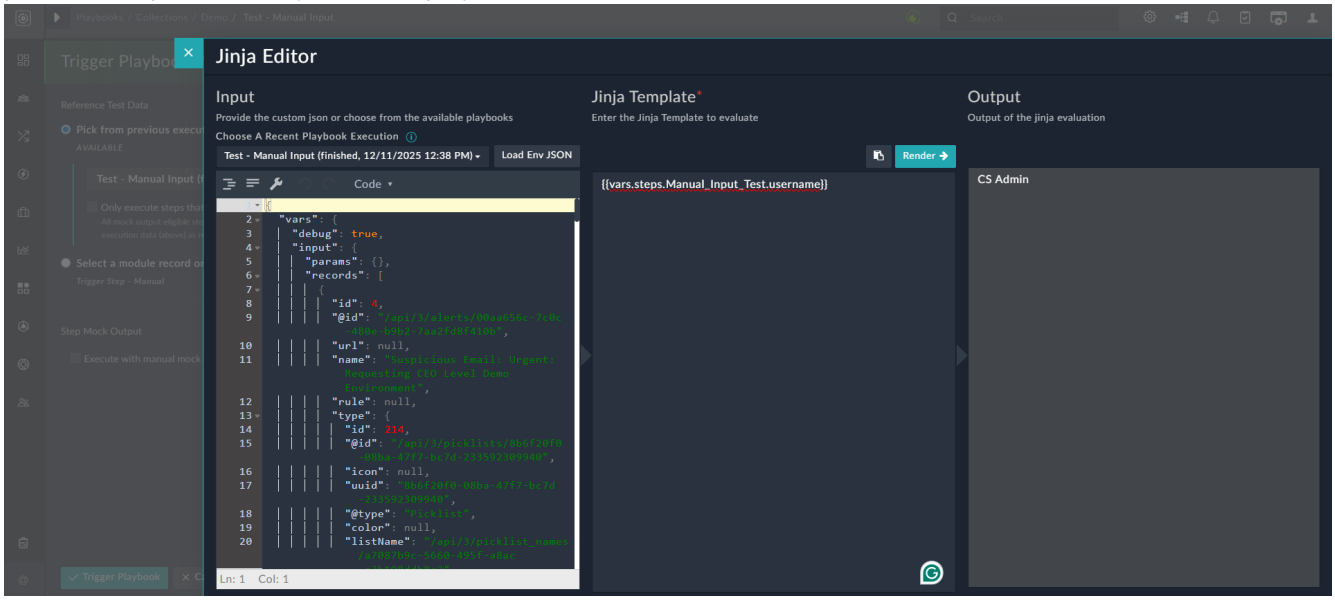


You can select an execution from the **Choose A Recent Playbook Execution** drop-down list and click **Load Env JSON** to view the environment JSON for that specific execution. The playbook step data (`vars.steps`) has been included in the 'ENV' data, which helps with playbook debugging. The output of `vars.steps` is provided in

dictionary format, allowing step-level Jinja expressions to be evaluated without any modification:

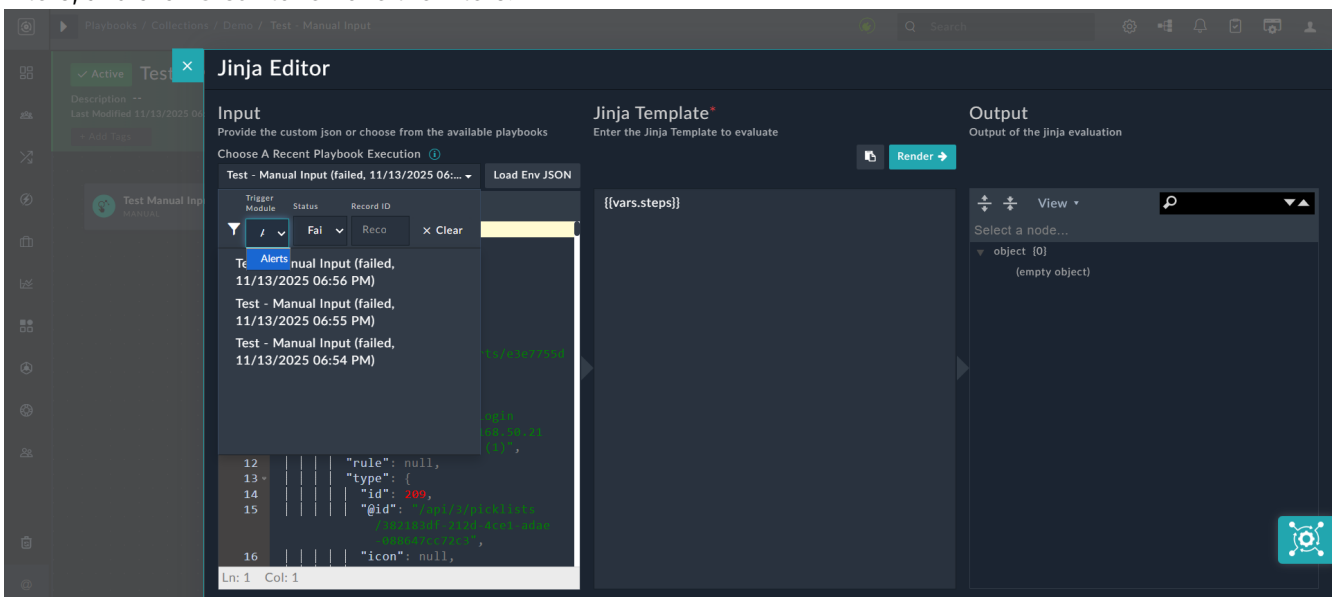


Additionally, the Jinja Editor correctly handles step-result data, i.e., the Jinja Template map results from previous steps accurately once the playbook environment JSON is loaded:



You can filter the list of executions by module, status, and record ID (if the playbook was triggered on a specific record). Use the **Trigger Module** drop-down list and the **Status** drop-down list to filter playbook executions by module and status, respectively. You can also filter the executions for a specific record by entering the ID of the record in the **Record ID** field, allowing you to retrieve playbook logs for that record. Click **Apply** to apply the

filters, and click **Clear** to remove the filters:



Notes for filtering playbook executions:

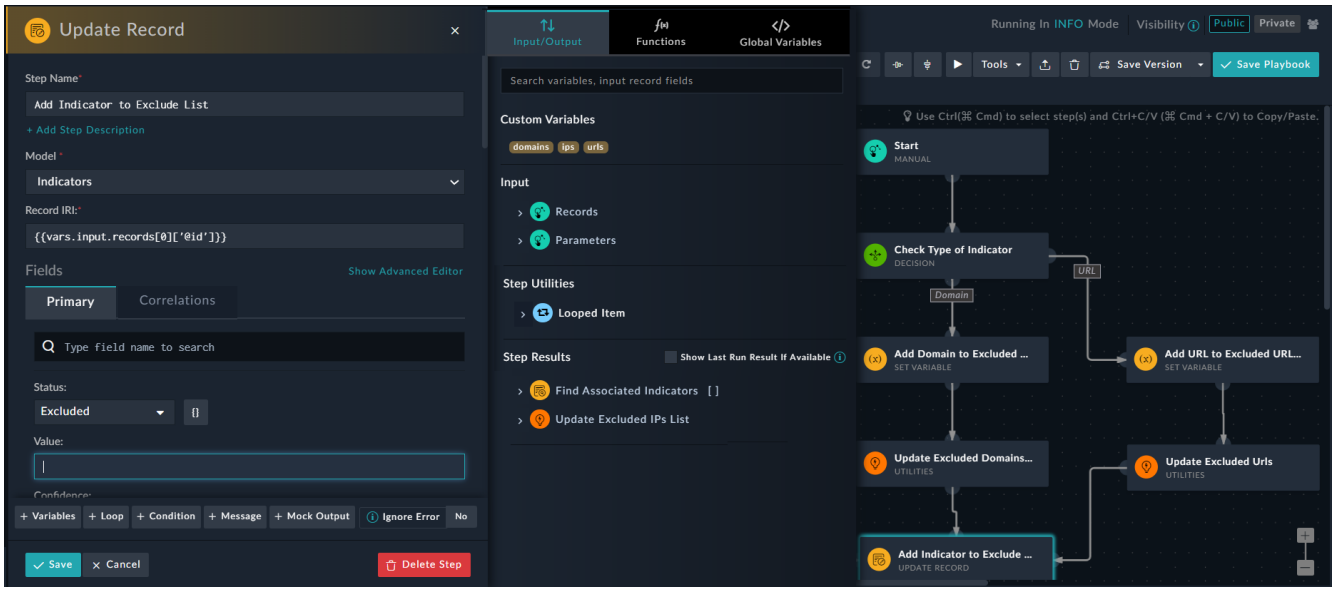
- The Jinja Editor only displays the latest 30 filtered playbook executions and shows the execution time in '`MM/dd/yyyy hh:mm a`' format.
- If the playbook trigger step has its **Execution Behavior** set to **Does not require a record input to run**, the **Record ID** field will not be available for searching. For more information on playbook triggers, see the Playbook Triggers chapter.
- If the playbook trigger is set to '**Referenced**' or '**Custom API Endpoint**', you can only filter playbook executions by '**Status**'.

Dynamic Values Window Usage (I/O, Functions, Global Variables)

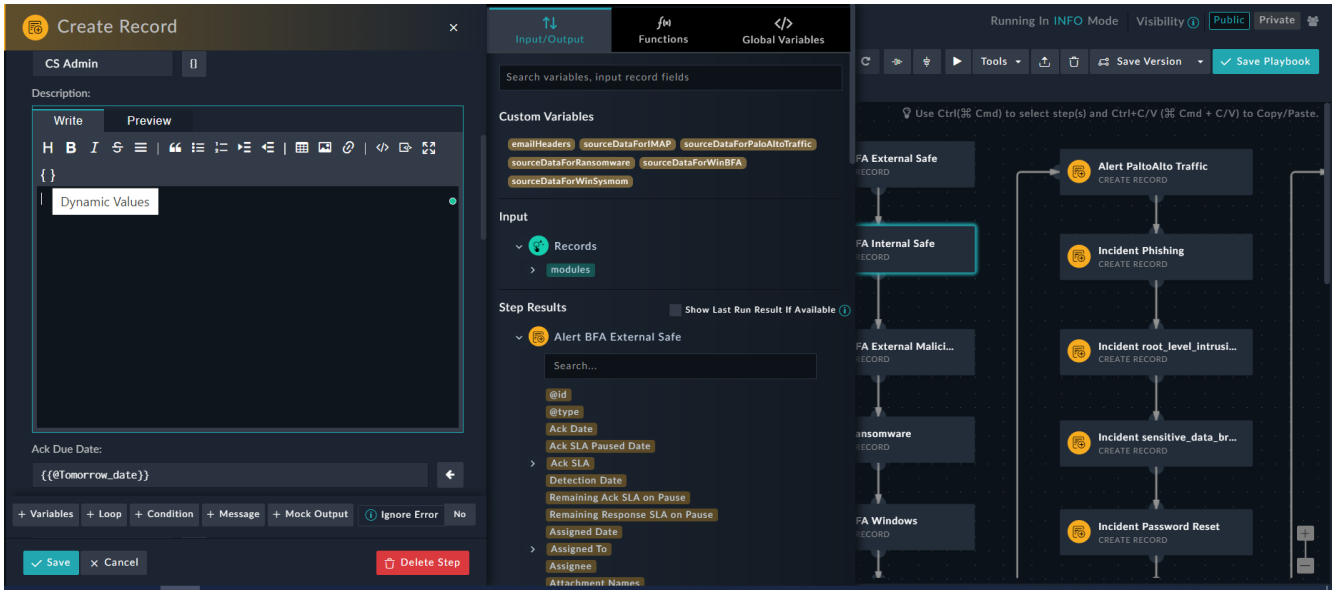
Dynamic Values is used within the Playbook Designer. Use the Dynamic Values directly within steps of your playbook to dynamically add Jinja to those steps. Click a step within the playbook that takes Jinja as an input and Dynamic Values is displayed. Choose from the options presented to add Jinja to the step.

Dynamic Values is visible for input fields such as text fields, rich text, date/time fields, picklists, checkboxes, etc. You can use the **Add Custom Expression** button to toggle fields, such as drop-down lists and checkboxes, and add custom jinja expressions for fields such as picklists, Lookups, File selectors, rich text, text fields, etc. Clicking the **Dynamic Values** button also displays the **Dynamic Values** window using which also you can add expressions to these fields. The ability to add Jinja expressions to these fields enables you to customize your playbooks further.

In release 7.0.0, FortiSOAR has updated the arrow library due to which the `timestamp` attribute has been changed into `int_timestamp` for *DateTime* jinja expressions. Therefore, new playbooks must use the `int_timestamp` for any *DateTime* jinja expressions. For more information, see the Dynamic Variables chapter.



You can also use Dynamic Values within a `Text` field type that has a subtype of either `Rich Text (Markdown)`, which is the default, or `Rich Text (HTML)`. To display Dynamic Values within a `Rich Text` type field, click the **Dynamic Values** ({}) icon in the formatting toolbar as shown in the following image:



The Dynamic Values Window displays the following tabs:

- Input/Output
  - Custom Variables
  - Input
  - Step Utilities (introduced in release 7.4.1)
  - Step Results
- Functions, including IRI Lookup
- Global Variables

Before you delete or modify any global variable or variable(s) ensure that you have removed or updated the variable in the Playbook to ensure that the change does not affect the functionality of the playbook.

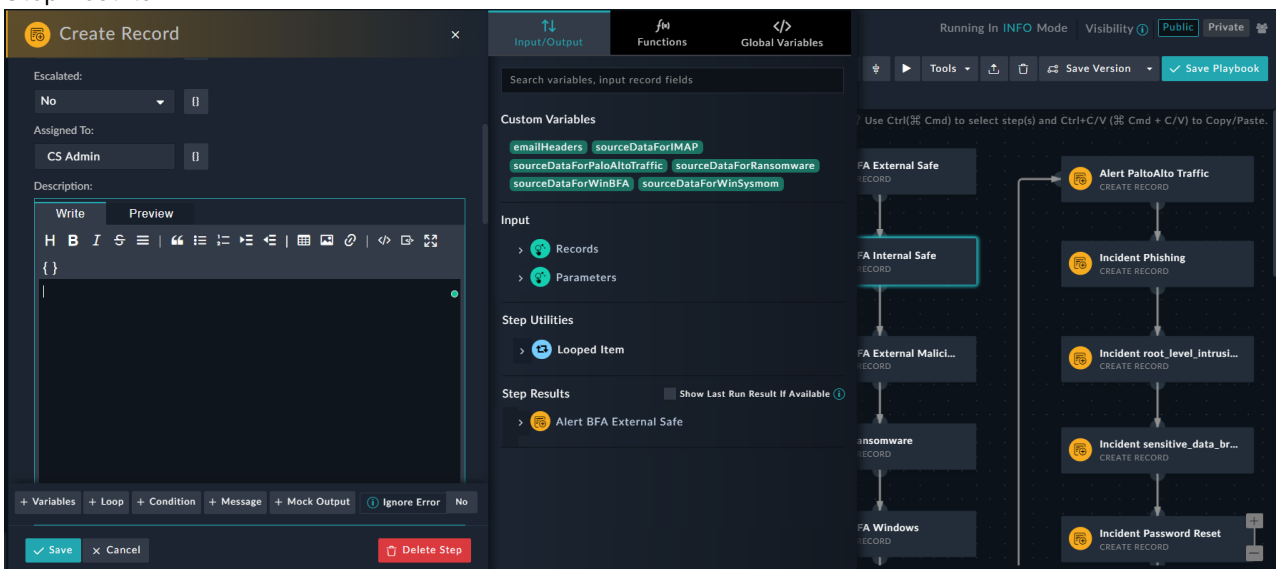
Use the “Functions” tab on Dynamic Values to build medium-level complexity playbooks based on your requirements without a lot of programming experience. For more information, see Functions.

Input/Output Tab

Use the **Input/Output** tab to effectively build your playbook using various available options for content, variables, parameters, previous step results (output), etc.

The Input/Output tab contains the following options:

- Custom Variables
- Input
- Step Utilities (introduced in release 7.4.1)
- Step Results

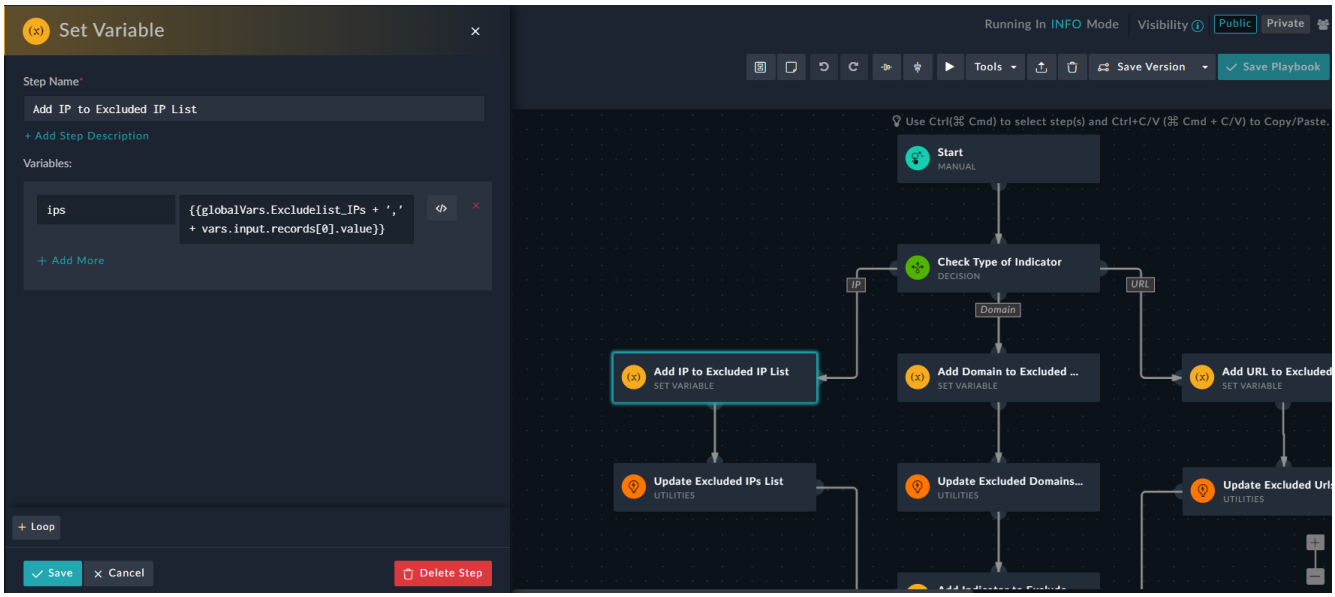


You can search for 'Custom Variables' and 'Input Record' fields, using the search box present in the **Input/Output** tab. In the case of 'Step Results', search works individually for each step, i.e., you can search for fields within each individual step.

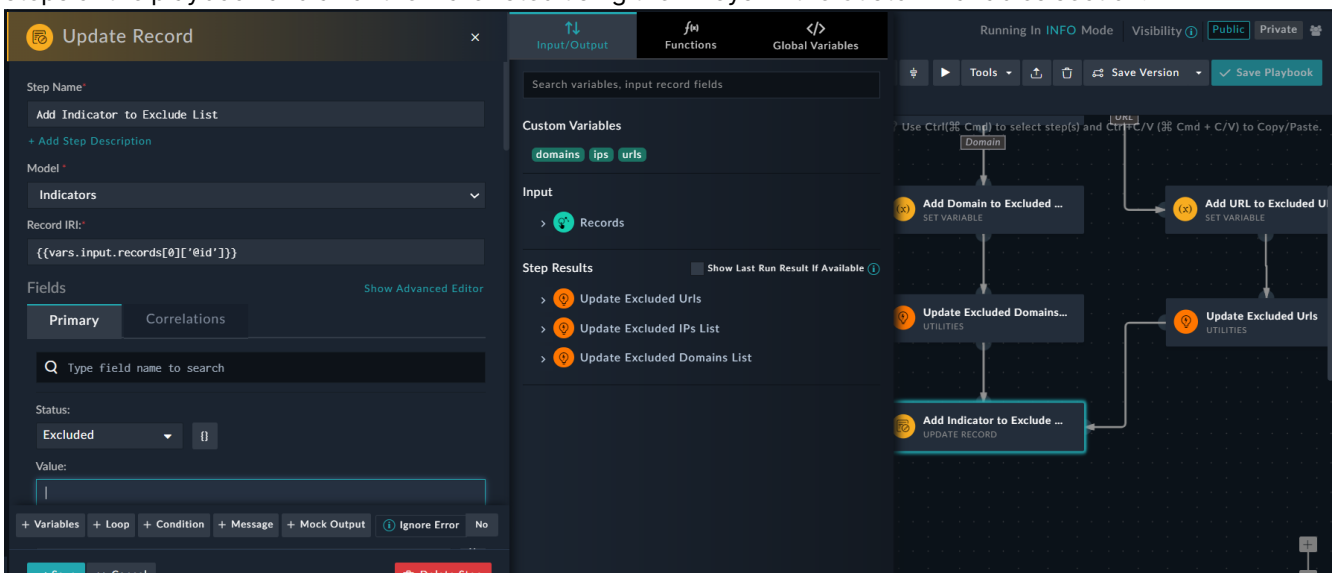
Custom Variables

Custom variables are variables that can be used only in the playbook in which it is defined. Therefore, the scope of a variable is *local*. To create a custom variable, in the Playbook Designer, click the **Set Variable** step and add the variable name in the `Name` field and the jinja expression that represents the variable in the `Value` field, and

then click **Save**:



You can add multiple 'Set Variables' in a playbook to define multiple custom variables that can be used in the steps of the playbook and all of them are listed using their 'keys' in the Custom Variables section:



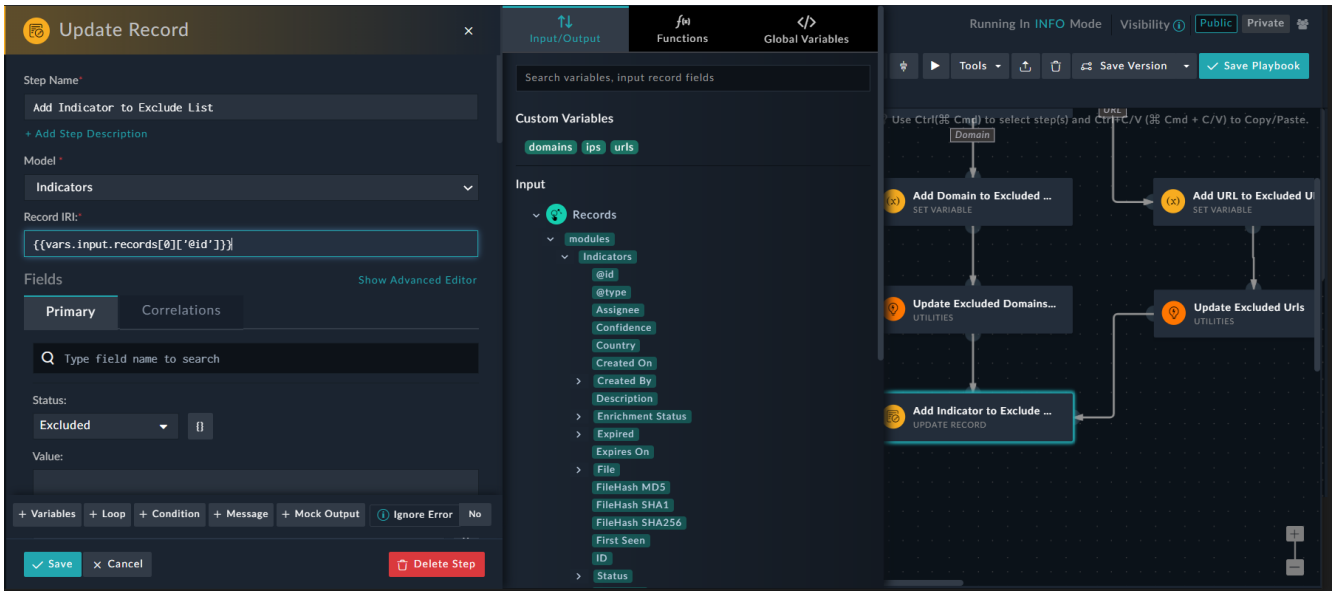
Custom Variables can be used like Global Variables, except that custom variables can be used only in the remaining steps of the playbook in which they have been defined or in any child playbook, regardless of how many levels deep the child playbooks are called.

If there are no variables defined, the **Input/Output** tab displays the following message: `No custom variables available`.

Input

The **Input** section in the Input/Output tab provides you with content and parameters that you can use in your current step. The content and parameters are represented using their 'keys', such as 'Assignee', 'Created On',

etc:



The **Input** section has the following options:

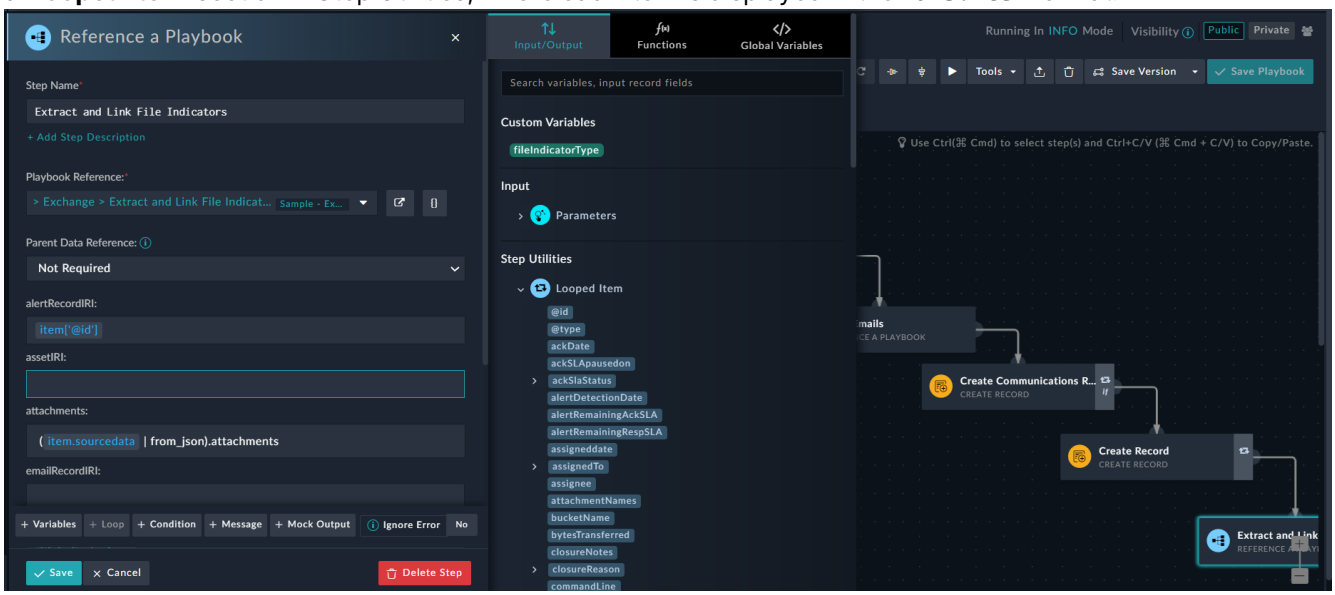
- **Records**:
  Trigger step data has been added as part of the **Inputs** so that you can use the variables and data from the module on which trigger has been added. Data of the trigger step appears when you click **Records**.
- **Parameters**:
  - Parameters that you have defined using **Tools** > **Parameters** in the playbook designer, appear when you click **Parameters** in the **Inputs** section.
  - Data of the trigger step, in case of `Custom API Endpoint Trigger`, appear when you click **Parameters > api_body** in the **Inputs** section.

While importing playbooks that were created using an older version of FortiSOAR, before you use the "Input" option in any step, ensure that you open and save the trigger step and then save the playbook.

Step Utilities

The **Step Utilities** option enables you to use the values of list items if the step contains a loop. This option makes it easier to use the values of the items that are part of a list in the playbook step. Dynamic Values displays

a '**Looped item**' section in Step Utilities, where each item is displayed in the `vars.item` format:



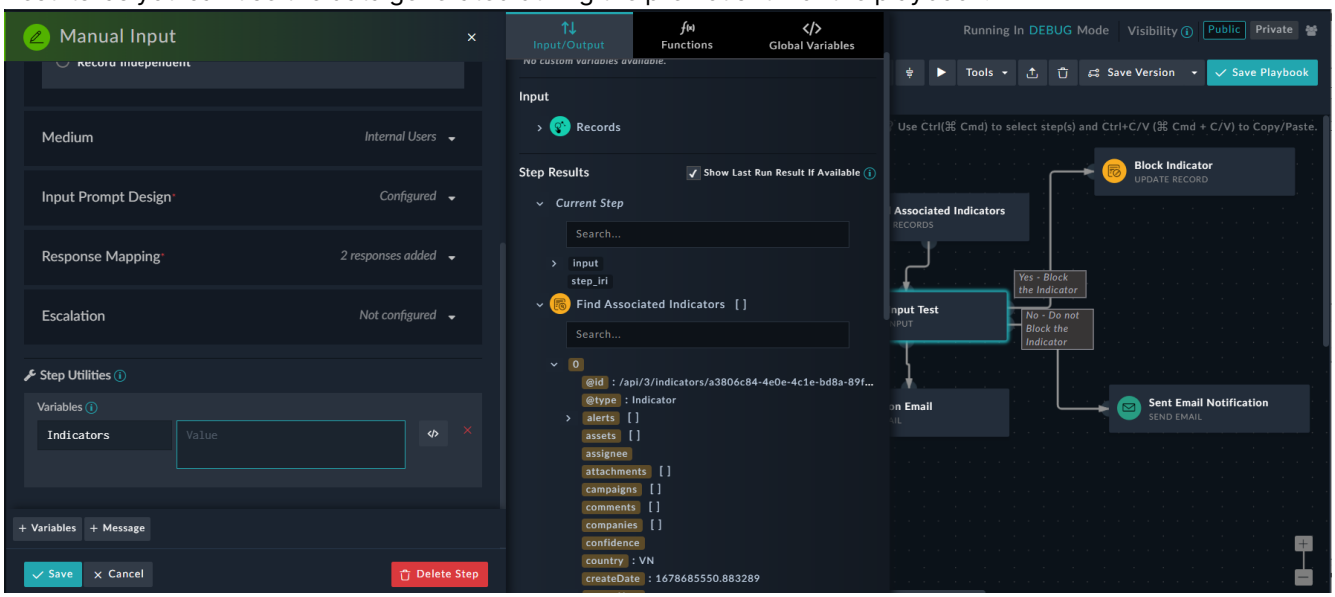
Note that if a playbook's trigger condition is set for more than one module, then the looped item for `vars.input.records` suggests fields across all the modules.

Be cautious while using variables from **Input > Records** while using a 'for each' loop. When users add any specific item from `vars.record` in the 'for each' loop, the '**Looped Item**' section in Step Utilities displays all fields of `vars.record` instead of fields of the selected item.

Step Results

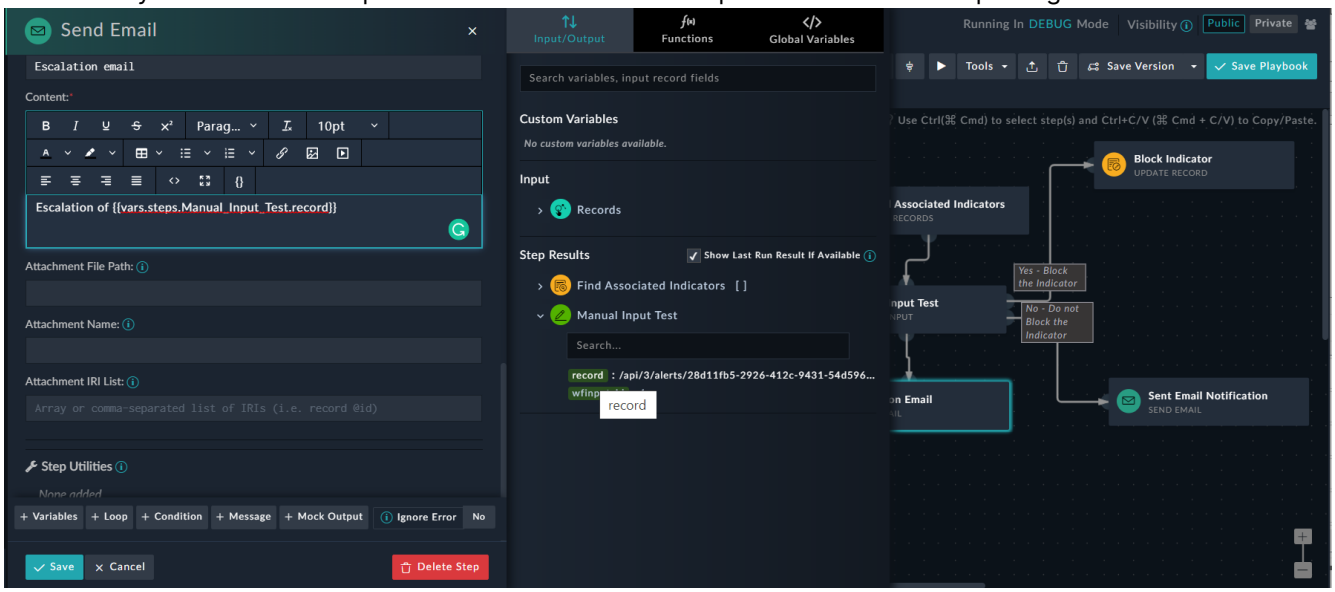
The **Step Results** option enables you to use the output of the steps that have been executed, in the current step. Selecting the **Show Last Run Result if Available** option dynamically pulls the last available execution data to create a reference schema compared to the pre-defined output schema. This enhances the usability of 'Step

Results' as you can use the data generated during the previous run of the playbook:



The data generated during the previous run of the playbook is available only if you run the playbook in the 'DEBUG' mode. For more information on the how to set a playbook to run in the DEBUG mode, see the Setting Logging Levels topic.

To use the output of the steps that have been executed, click the step in which you want to the use the executed steps' output, which then displays the Dynamic Values window. The output schema, with all its attributes, for all the executed steps are visible in the **Step Results** section. You can then use the output schema and attributes of any of the executed steps as an input to the current step based on the logic or functionality of the current step. You can also search the output schema of the step using the search box:



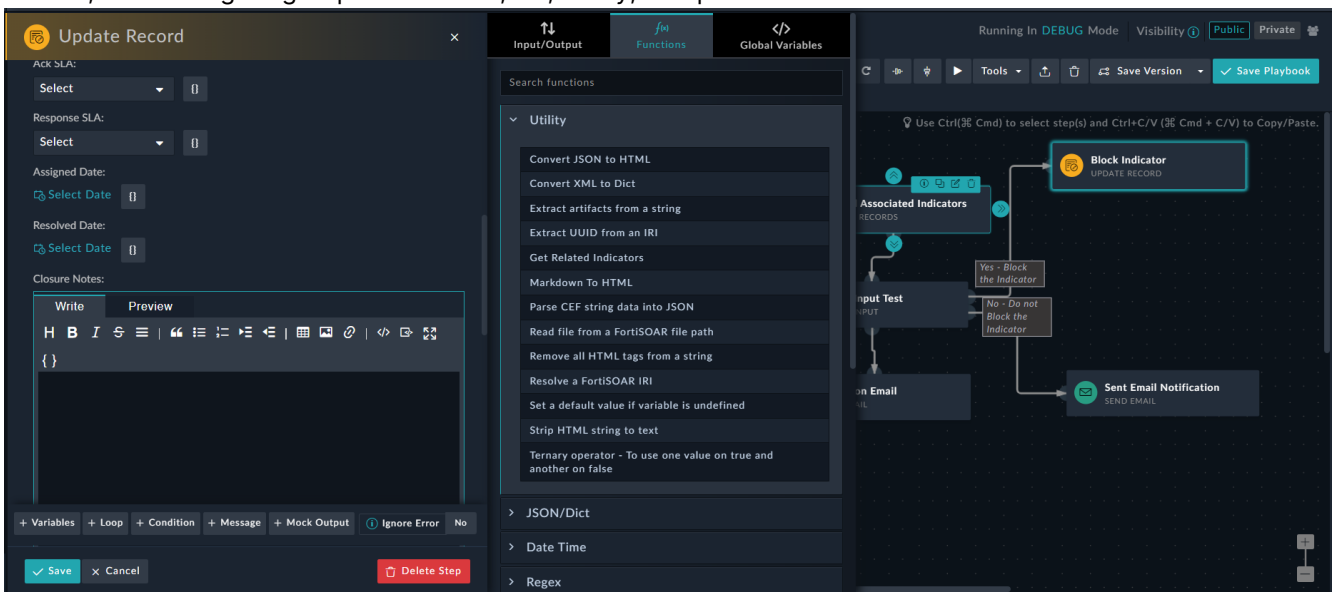
Step result output schemas are represented using 'keys', for example 'record'. The Jinja value of the 'keys translate to `{{vars.steps.<step_name>.keyname}}`, for example, `{{vars.steps.Manual_Input_Test.record}}`.

To use an array element in an executed steps' output, then you must specify the position of the element (index `[i]`) in the Jinja that is generated. The index value of an array starts from zero `[0]`. For example, if you want to fetch the `name` property from the `Find_all_Open_Alerts []` array from the executed steps' output, then the Jinja that is generated is as follows: `{{vars.steps.Find_all_Open_Alerts[0].name}}`.
Therefore, before you run the playbook and require to fetch any element other than the first element in the array, you must provide the position of the element. For example, if you want to fetch the `name` property of the 4th element of the `Find_all_Open_Alerts []` array then your Jinja must be written as `{{vars.steps.Find_all_Open_Alerts[3].name}}`.

If there is no step output available or if you are at the first step in the Playbook or if the step is not connected to another step, then the Jinja generator displays the following message: `There are no input defined`.

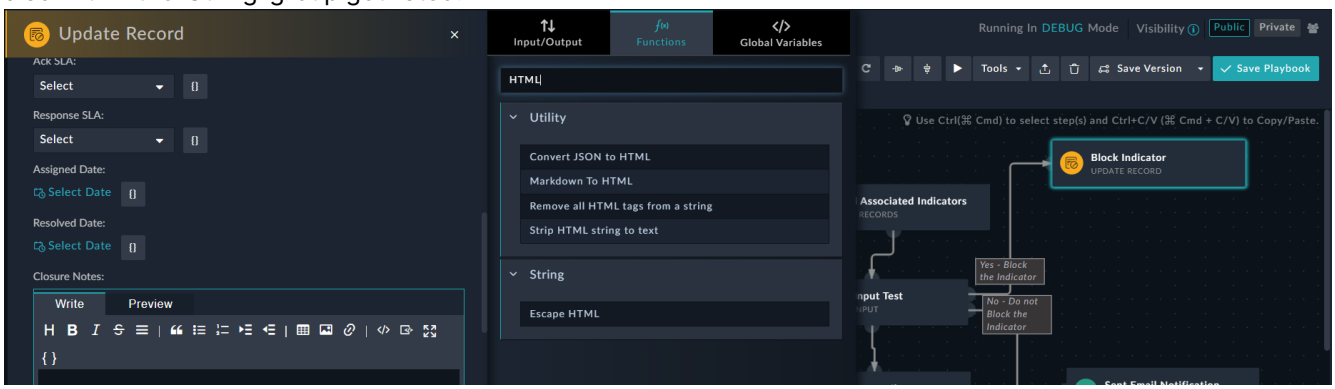
Functions

Use the "Functions" tab in Dynamic Values to build playbooks based on your requirements and without programming knowledge, and add various operations and expressions to playbooks. The **Functions** tab contains easy-to-understand operations that cover most aspects of playbook development. The operations are grouped logically as per their functionality, for example, if you want to convert datetime into a specific format or to a different time zone then these operations will be listed in the 'Date Time Operations' list. Similarly, if you want to replace text in a string with a regex, then these operations will be listed in the 'Regex Operations' list. By default, the first logical group of functions, i.e., Utility, is expanded:

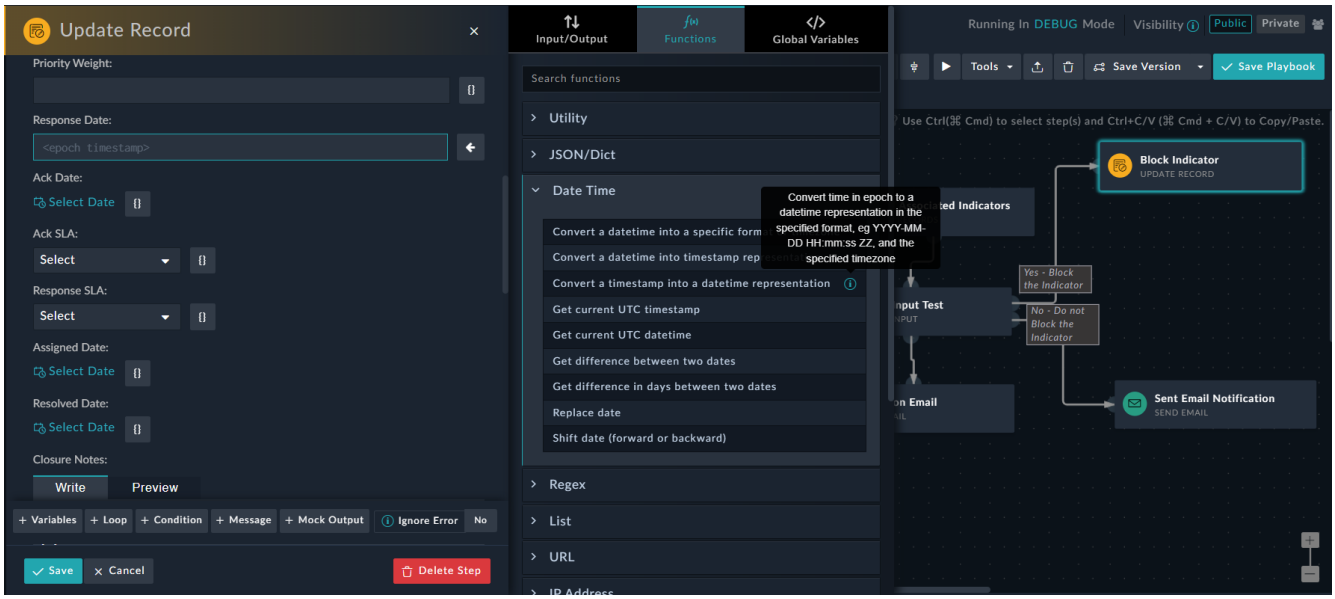


You can search for functions using the search box in the **Functions** tab. On search of a result, all functions across the logical groups get listed, for example if you search for `HTML`, functions within the 'Utility' group and

also within the 'String' group get listed:

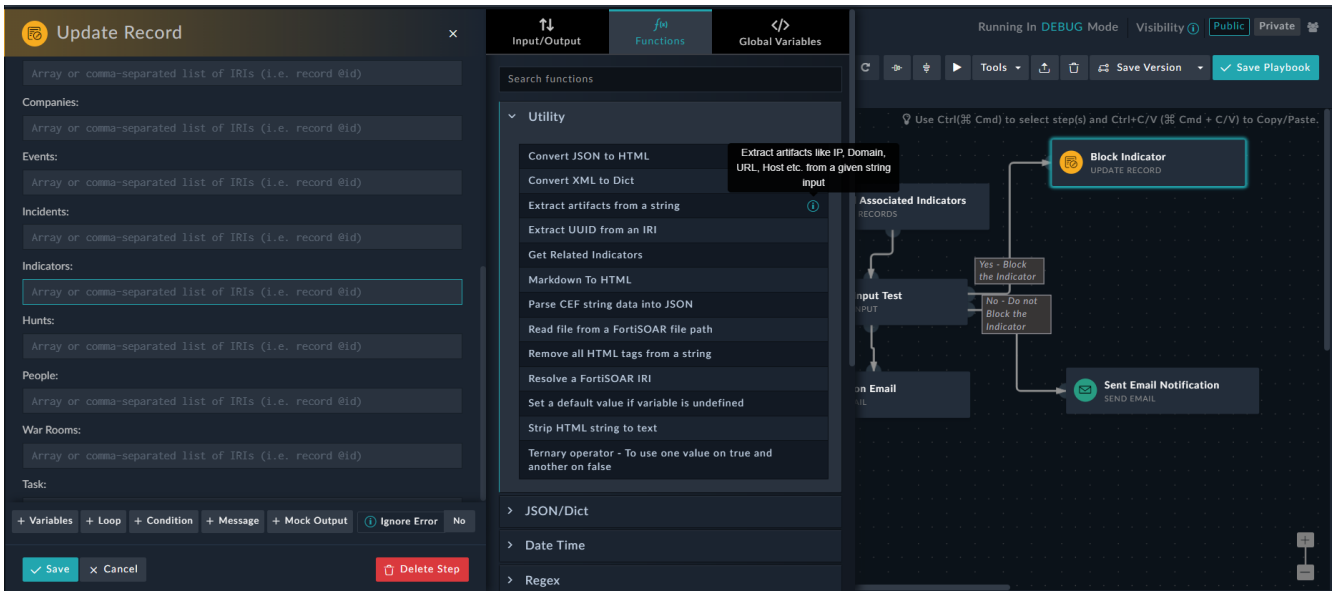


Each function has an information icon that you can hover over to view more information about that particular operation.

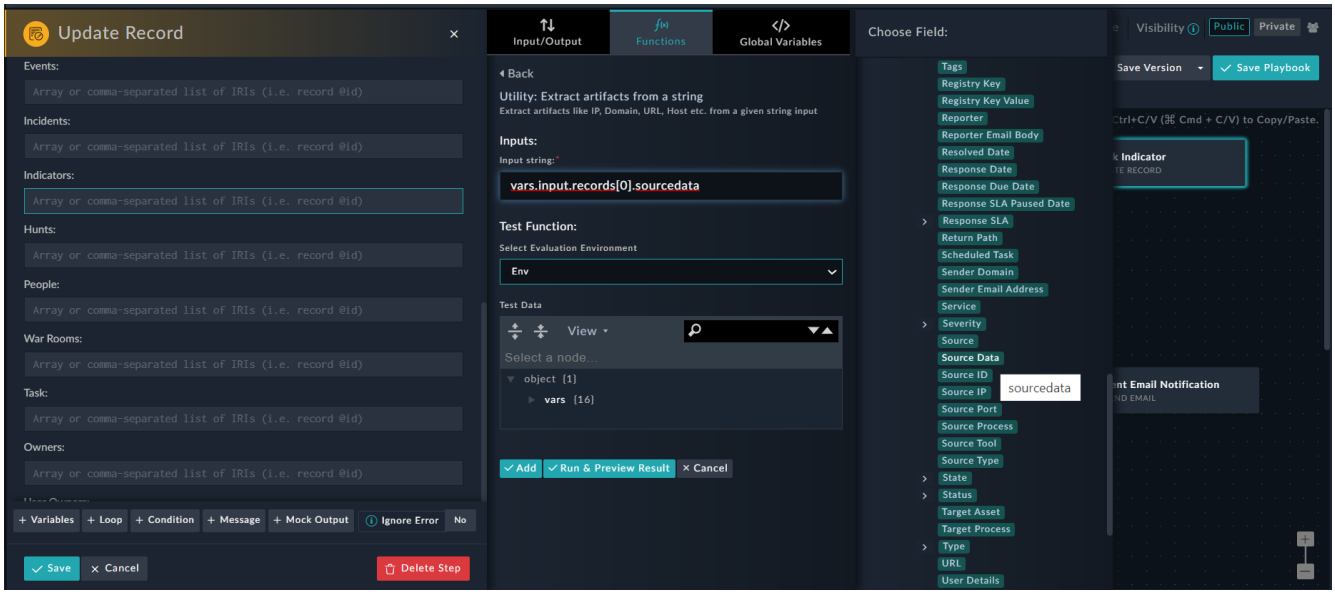


An example of using an expression would be requiring to extract artifacts from the source data of an alert that has been created in FortiSOAR from a SIEM and update that record with the extracted artifacts. You can use **Utility Operation** > **Extract artifacts from a string** function to extract artifacts from the source data.

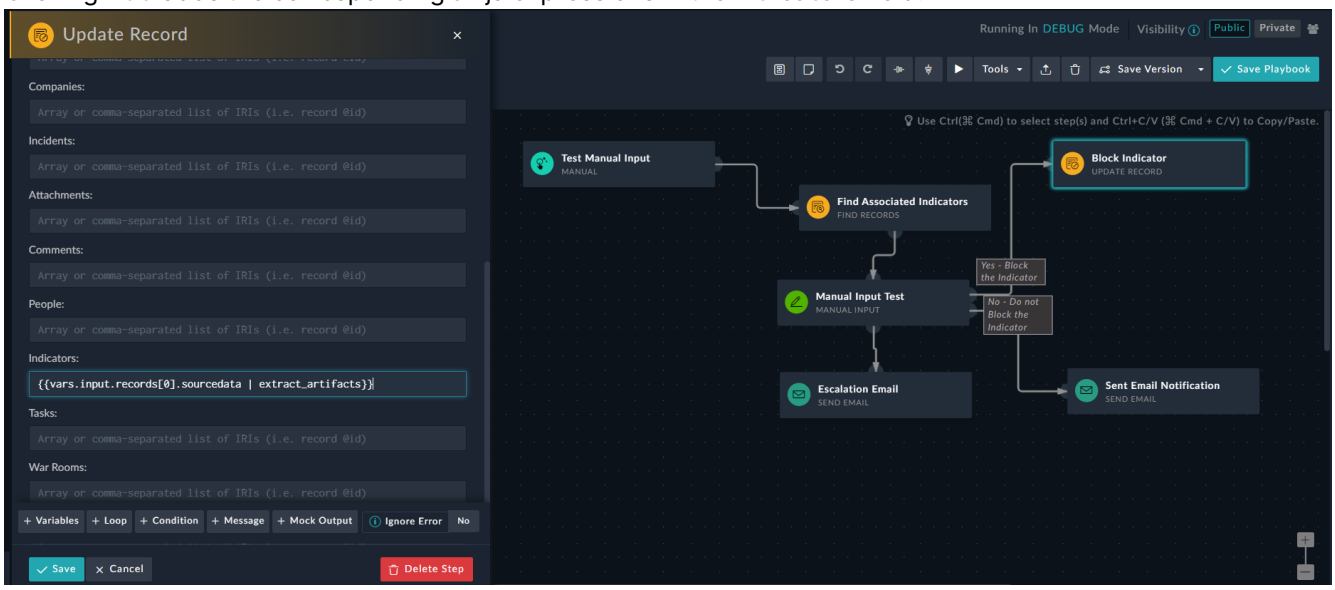
To use the Extract artifacts from a string function, in the "Update Record" step, click a field in which you want to use the function. For example, the **Indicators** field in the **Correlations** tab of an alert record, which displays the Dynamic Values window. Click the **Functions** tab and then click the **Utility Operations** list:



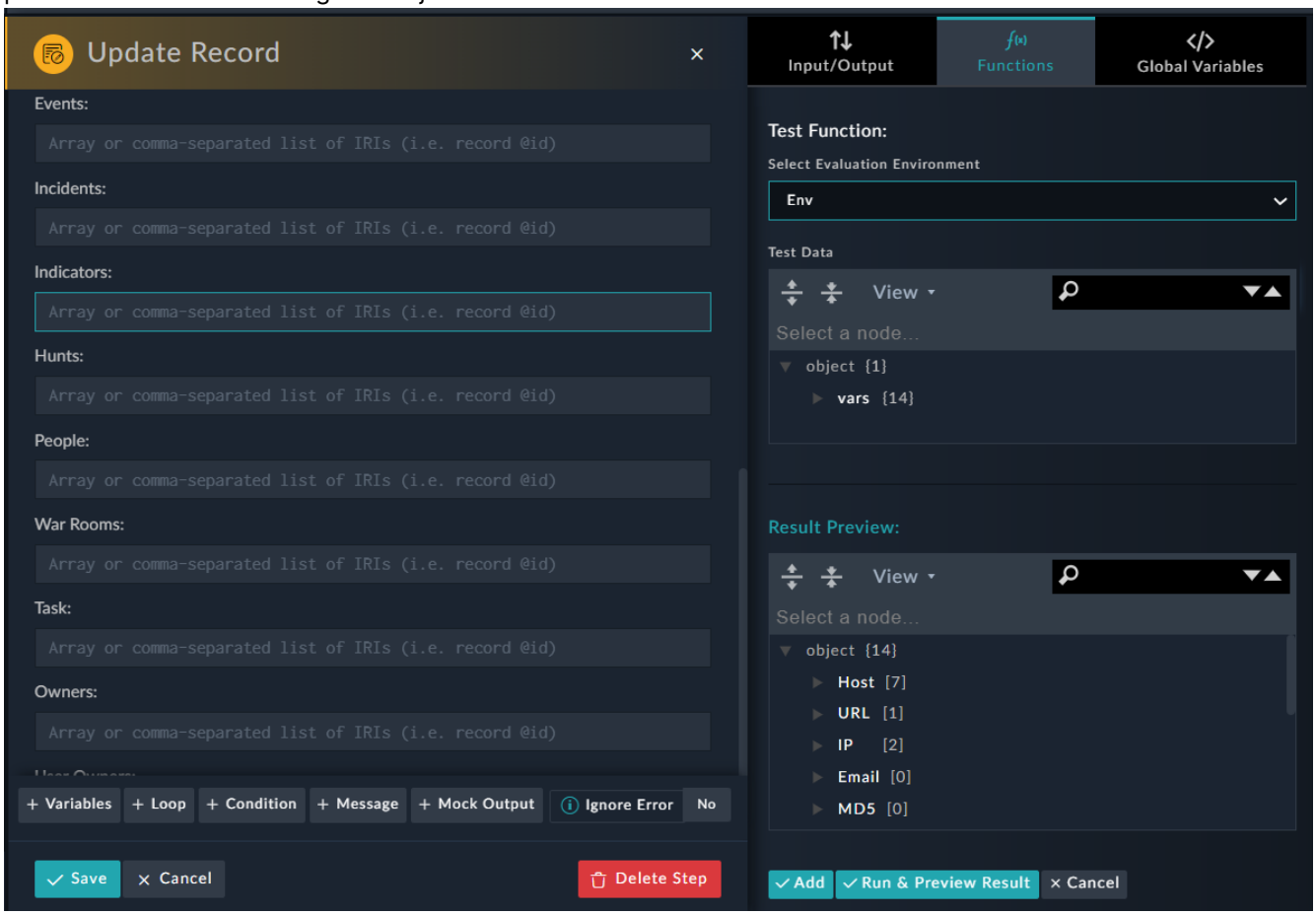
Click the **Extract artifacts from a string** function, which in turn displays an **Inputs string** text box. From the **Choose Field** list, you can select **Source Data** by clicking **Input > Records > modules > Alerts** and clicking the **Source Data** field, which adds the corresponding Jinja value in the **Input string** field:



Clicking **Add** adds the corresponding Jinja expressions in the **Indicators** field:

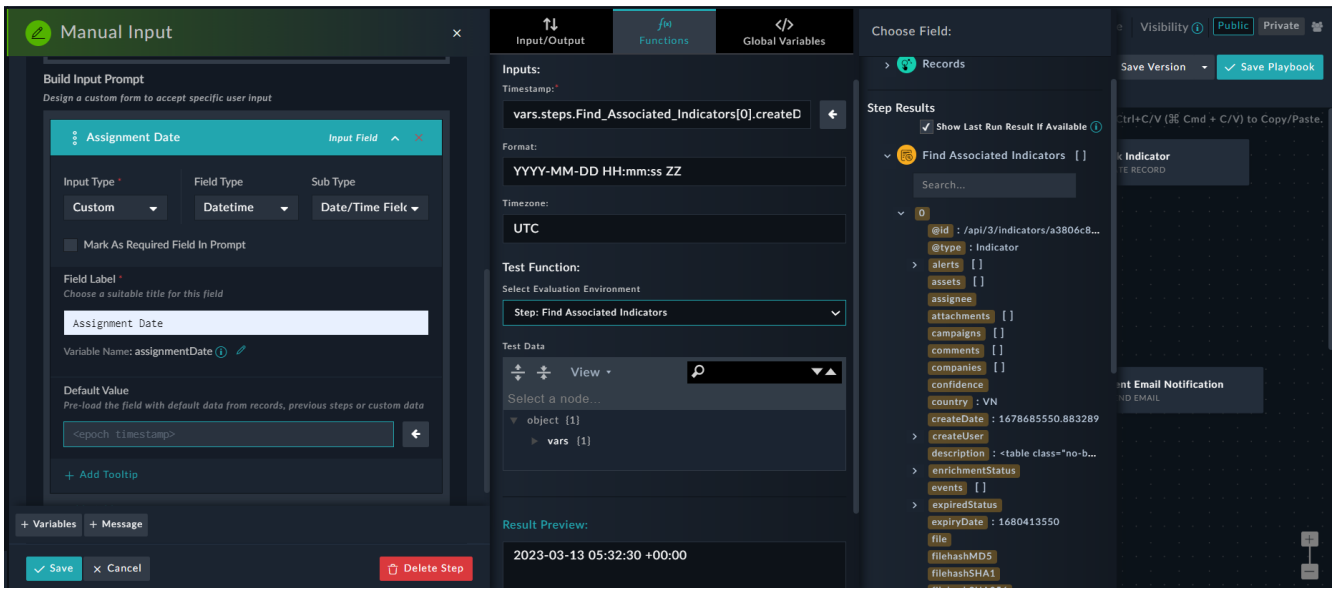


You can also choose to test the added expression and preview the result before you add the expression to your playbook. To test the expression, you can choose an Environment from the **Select Evaluation Environment** list such as, the complete environment (Env) or any of the steps of the playbook, and click **Run & Preview Result** to preview the result of adding this Jinja to the selected **Indicator** field:



Click **Save** to save the Update Record step and then save the playbook. Now, when you run this playbook for an alert that contains source data, this step will extract artifacts from the source data and update the indicators associated with the alert record with the extracted artifacts.

Another example of testing an added expression and preview the result before you add the expression to your playbook would be to test an expression where you want to get convert an epoch timestamp to represent datetime, open Dynamic Values and click **Functions** > **DateTime Operations** and select **Convert a timestamp into datetime representation**, then select the evaluation environment, can select 'Env' or any of the previous steps and then clicking **Run & Preview Result**. You will be able to view the result of the expression in **Result Preview**:

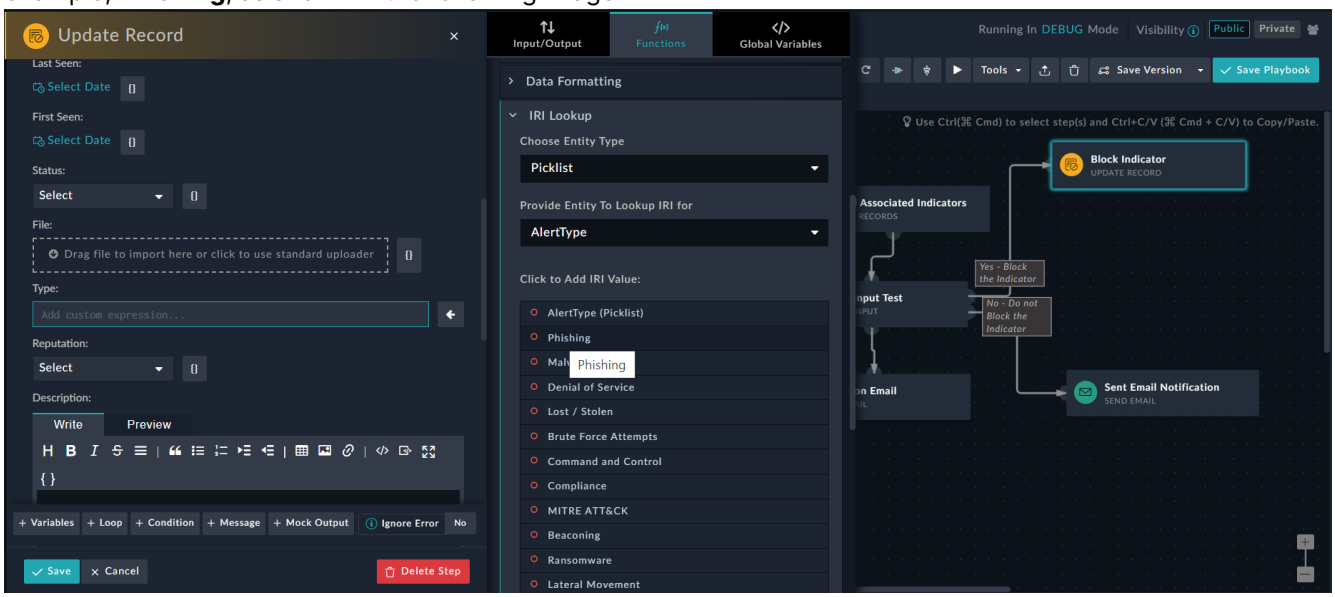


IRI Lookup

All foreign key references use International Resource Identifiers (IRIs) to reference records within the system. IRIs are generated automatically when FortiSOAR inserts records. FortiSOAR uses IRI values in multiple places for referencing picklists, playbooks, attachments, etc. Use the **IRI Lookup** option in the **Functions** tab to efficiently use the IRI values of picklists, attachments, or playbooks configured in your system.

To use the IRI lookup, click the step in which you want to insert an IRI value, which then displays the Dynamic Values window. Click the **Functions** tab and click the **IRI Lookup** option. Click **IRI Lookup** and from the **Choose Entity Type** drop-down menu, choose the entity, either a **Picklist**, **Attachment**, or **Playbook**. For example, if you choose **Picklist**, then from the **Provide Entity To Lookup IRI for** drop-down menu, select the `Picklist` whose IRI value you want to add to the step. In our example, we want to add the IRI value for the picklist that retrieves the types of alerts. Therefore, from the **Provide Entity To Lookup IRI for** drop-down menu, select **AlertType**, and then in the `Click to Add IRI Value` section, select the alert type that you want to add, for

example, **Phishing**, as shown in the following image:



Once you click **Phishing**, the IRI value (jinja) of 'Phishing' alert type is added to the playbook as a 'key', whose Jinja translates to `{{("picklistName" | picklist("itemvalue of picklist","@id")}}`, for example `{{"AlertType" | picklist("Phishing", "@id")}}`

Adding your own functions

You can also create your own functions or use existing Jinja2 filters (https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html), and then add them to the "Functions" tab in Dynamic Values. To add your own functions, use the steps mentioned in the *Building your own connector* chapter in the "Connectors Guide" for adding functions or exposing an action as a function to Dynamic Values.

Useful Functions

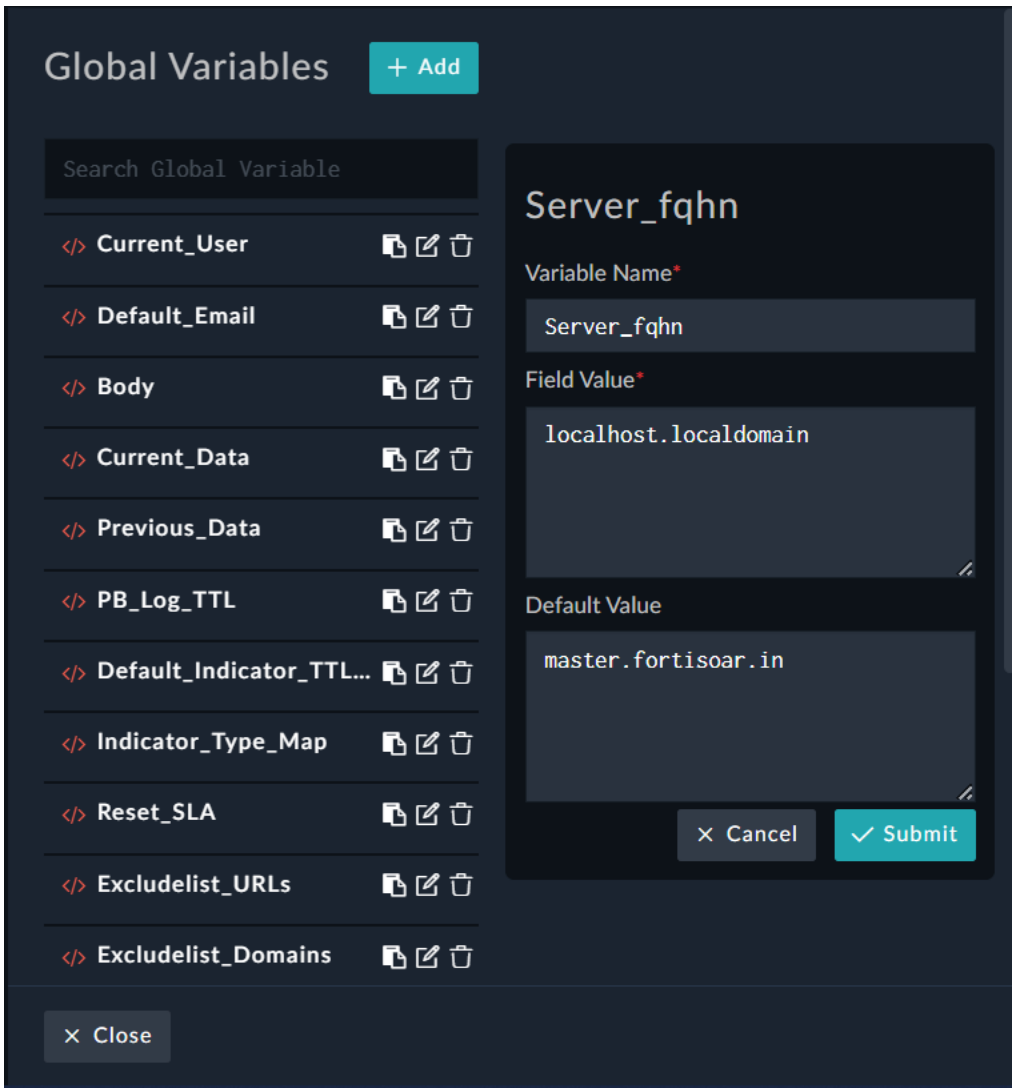
- **Title**: Get Related Indicators
  Expression: `find_indicators`
  Description: Gets a list of related indicators that are linked to the given alert or incident based on the type and reputation specified. Returns the IRI, type, value, and reputation fields for the linked indicators.
  Example:
  Input:
  Record iri = /api/3/alerts/aaaa0a52-340f-4412-bc79-94d6c4b08365
  Indicator Type = IP Address
  Indicator Reputation = Malicious
  Output:
  This function will fetch all indicators with type 'IP Address' and reputation 'Malicious' that are linked to the given record (alert or incident) IRI. Users can choose not to apply 'Type' and 'Reputation' filters by keeping these fields blank.
- **Title**: Check if a string is null or empty
  Expression: `{{not (vars.input.records[0].description and vars.input.records [0].description.strip())}}`
  Description: Validates if the given input string is null or empty.
- **Title**: Check if a key present in a JSON
  Expression: `{{vars.input.records[0].emailBody[vars.input.records[0].domain] is defined}}`
  Description: Returns 'true' if the given key exists in the specified JSON object.
- **Title**: Check if a key is present in a JSON and its value is not null
  Expression: `{{(vars.input.records[0].emailBody[vars.input.records[0].domain] is defined) and (vars.input.records[0].emailBody[vars.input.records[0].domain] != None)}}`
  Description: Returns 'true' if the given key exists, and if its value is 'not null', in the specified JSON object.
- **Title**: Check if a key is not present in a JSON, or if its value is null
  Expression: `{{(vars.input.records[0].emailBody[vars.input.records[0].domain] is undefined) or (vars.input.records[0].emailBody[vars.input.records[0].domain] == None)}}`
  Description: Returns 'true' if the given key does not exists, or if its value is 'null', in the specified JSON object.
- **Title**: Find all elements from a list matching a given condition
  Expression: `{{vars.steps.find_alert_indicator | selectattr("name", "eq", "alert-name") | list}}`
  Description: Returns a filtered list of items that match the given condition.

Global Variables

Global Variables are variables that can be used across playbooks. You can declare a global variable once and then use it across all playbooks, instead of having to redefine the variable every time in each playbook. You can create global variables only in the Playbook Designer. FortiSOAR includes some pre-defined global variables.

To create a global variable, in the Playbook Designer, click **Tools** > **Global Variable**. Click **New Global Variable** and enter appropriate content in the `Variable Name` and `Field Value` fields for the variable and click **Submit** to create a global variable. You can optionally also add the default value for the variable in the `Default Value` field.

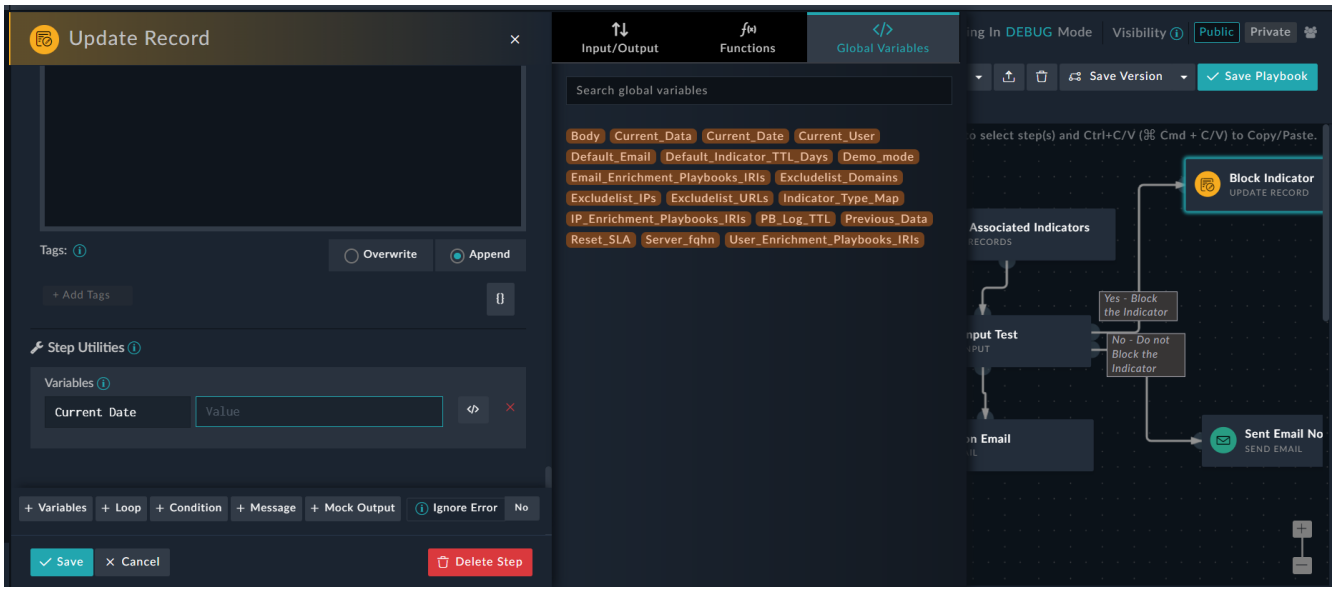
Variable Names must always begin with a character when you are creating global variables and the name can contain only alphabets and numerals. Special characters and spaces are not allowed.



To ensure that the correct hostname is displayed in links contained in emails sent by System Playbooks, you must update the `Server_fqhn` global variable in **Global Variables**. Click the `Edit` icon in the `Server_fqhn` global variable, and in the `Field Value` field add the appropriate hostname value, and then click **Submit**. If you have not specified the hostname in global variables, then the hostname that you had specified or that was present when you installed FortiSOAR will be the default hostname and this will be added in the email. In this case, ensure that you have used the `Server_fqhn` global variable in the `Send Email` step in the playbook that is sending the email.

Example of using a global variable: In the `Set Variable` step, you need to set a name and value. When you click the `Value` field, the Dynamic Values window is displayed. Click the **Global Variables** tab, and you will see a list of global variables that have been created; you can also search for a global variable in the search box. Click the global variable that you require, for example 'Current Date' to add the Jinja value of the global variable in the

`Value` field:



Jinja Template Security Enhancements and Configuration Options

Restriction on Access to Private Members in Jinja Templates

From FortiSOAR release 7.4.0 onwards, Jinja templates are blocked from accessing private functions, modules, and classes. This restriction is imposed to prevent unwanted usage, such as the using malicious Jinja templates in playbooks. The following Jinja, for instance, is restricted:
`{{ self.__init__.__globals__.__builtins__.__import__('os').popen('<sensitive file>').read() }}`

Because of this, playbooks containing Jinja that try to access private members such as, `__setitem__` cause playbooks to fail with the "`CS-WF-7: Invalid Jinja template: 'access to attribute '__setitem__' of 'dict' object is unsafe.' template :: …`" error. Therefore, it is recommended to modify such private references in playbooks. For example, `__setitem__(key, value)` can be changed to `update({key: value})`

Alternatively, if you want existing playbooks (created prior to 7.4.0) to work and want to allow Jinja templates to access private members, which is *NOT Recommended*, then you can remove the restrictions by adding `USE_SANDBOX_ENV: false` to the `[application]` section in the `/opt/cyops-workflow/sealab/sealab/config.ini` file.

Configurable Blocking of Sensitive Jinja Filters

You have an option in FortiSOAR to block sensitive filters by using the `DISALLOWED_JINJA_FILTERS` parameter in the `/opt/cyops-workflow/sealab/sealab/config.ini` file. Currently, the '`readfile`' operation is blocked:
`vi /opt/cyops-workflow/sealab/sealab/config.ini`
`DISALLOWED_JINJA_FILTERS = ['readfile']`

Depending on your needs, you can add other filters to the list of `DISALLOWED_JINJA_FILTERS`.

Dynamic Variables

Syntax, Implementation, Scope, Functionality

Dynamic variables are objects that can be set and accessed within a playbook. Any valid Python object can be a dynamic variable. This includes ints, strings, dictionaries, etc. Variables themselves have no type information associated with them; however, playbook steps do. Steps may attempt to coerce dynamic variables into the expected data type; however, it is mostly on the caller to pass the correct types.

Variable names can contain letters, numbers, and underscores only.

Dynamic variables can be passed to playbook steps as arguments directly, or they may be embedded in a larger string, where they will act more as global variables (or macros), getting replaced by a string representation of themselves.

Syntax

Double curly braces ({{ }}) demarcate dynamic variables from the surrounding text. Anything that goes between the braces is a dynamic variable. The most basic use of the dynamic variable is as a simple dictionary lookup.

The general data structure you are using matters within the usage of the dynamic variable. JSON is the easiest data format to consume and use. XML may be converted into JSON directly so that it may also be used. The following examples assume that you are able to use a JSON format.

Let's look at some examples. Say you have an object (array) named `users` which has the following structure:

```
{
    'Alvian': 42,
    'Kreb': 413,
    'Mandu': 1
}
```

Example 1

You can then use dynamic variables to access the values of that object.

```
{{ vars.users.Alvian }}
```

This statement will evaluate to 42.

Example 2

```
There are {{ vars.users.Kreb }} Krebs in FortiSOAR.
```

This statement will evaluate to the string "There are 413 Krebs in FortiSOAR."

Example 3

```
{{ vars.users.does_not_exist }}
```

This statement would evaluate to an error and would be displayed as:

```
no such element: users['does_not_exist']
```

Example 4

Say you modified the object (array) named `users` to have the following structure:

```
{
    'Alvian': 42,
    'Kreb': {
       'original': 413,
        'pi': 3.14
    },
    'Mandu': 1
}
```

To access the secondary array is as easy as adding an additional key for the key-value pair you desire to access.

```
{{ vars.users.Krebs.pi }}
```

This statement would evaluate to 3.14. An alternative format for accessing the variable, which may be used in case of special characters present, is:

```
{{ vars.users['Krebs']['pi'] }}
```

This statement would evaluate identically to the previous, 3.14.

Implementation

The major driving force behind dynamic variables is Jinja2 templates. A general overview of how Jinja2 works can be found here.

Specifically, to render a template, Jinja takes two arguments: a context and a template string. The template string is the dynamic variable itself, which is provided by users in the playbook. The context object, on the other hand, will be created automatically before each and every playbook step. It contains various helper functions as well as the internal representation of the dynamic variable data.

Scope

Scope for dynamic variables is defined by the `COPY_ENV_FOR_REFERENCE_WORKFLOW` setting. Use this setting to pass variables to a reference playbook.

By default, the `COPY_ENV_FOR_REFERENCE_WORKFLOW` is set to `false`.

Functionality

There are several top-level objects that can be accessed within a dynamic variable.

Dictionary-like Objects

Most ordinary variables are stored under the vars namespace. Whenever a variable is declared using: class:`workflow.tasks.set_variable`, it will go under `vars`. Additionally, the playbook engine will automatically set the following variables:

- `vars.steps.<step_name>.keyname`, for example, `vars.steps.<step_name>.data`: This contains the return value of the previous playbook step.
- `vars.input.records`: This contains information about what triggered a playbook, i.e., the body of the inbound request.
- `vars.request.headers`: This contains the metadata of all the headers that are part of the playbooks environment, and which can be used in the playbooks, such as `X-RUNBYUSER` which is a jinja template to retrieve the name of the user who triggered the playbook. Some other parameters are, trigger type, authorization, accept, host, content-type, etc.
- `vars.input.params.api_body`: This contains the data passed from a Custom API Endpoint trigger.

Built-in Functions & Filters

Functions

- `arrow`: Datetime functions:

```
{{ arrow.utcnow().int_timestamp }}
```

In version 7.0.0, FortiSOAR has updated the arrow library, due to which the `timestamp` attribute has been changed into `int_timestamp` for *DateTime* jinja expressions, . For more information see, https://arrow.readthedocs.io/en/latest/releases.html#id4

New playbooks must use the `int_timestamp` for any *DateTime* jinja expressions.

- `uuid`: returns a uuid using python's `uuid.uuid4()` function

```
{{ uuid() }}
```

Filters

See the Jinja Filters, Functions, Expressions, and Extensions chapter for information.

Jinja Filters, Functions, Expressions, and Extensions

Use jinja2 filters to design and manipulate playbook step outputs. Jinja operations are supported in the Playbook Engine and you can also use the documented Custom Functions and Filters.

All filters are case-sensitive.

These examples present in this chapter provide a reference to common and very useful string operations that may be leveraged within the Playbook engine.

Some examples of jinja expressions that can be used while creating or adding notifications are included in the `Usage examples of Jinja Expressions in Notifications` section in the 'Notifications' topic in the *System Configuration and Management* chapter of the "Administration Guide."

You can also use Jinja extensions to enrich expressions, for more information, see the Jinja Extensions topic.

Jinja Filters and Functions

Filters

FortiSOAR supports the following filters:

- **`fromIRI`**: Resolves an IRI (Internationalized Resource Identifier) and returns the associated object(s).
  - Resolving an IRI from a string:
    `{{ '/api/3/events/8' | fromIRI }}`
    `{{ vars.event.alert_iri | fromIRI }}`
  - Accessing nested fields after using fromIRI. Use dot access for values returned by fromIRI.
    For example: To get a person record and return their 'name' field you can use the following:
    `{{ (vars.person_iri | fromIRI).name }}`
  - Recursively accessing related data. For example:
    `{{ ((vars.event.alert |fromIRI).owner| fromIRI).name }}`
  - Retrieving related objects for a record. For example:
    `{{('/api/3/alerts/<alert_IRI>?$relationships=true' | fromIRI).indicators}}`
    Use `fromIRI` when you need to fetch full object data from an IRI or resolve related data from a record.

- **toDict**: Coerces a string into a dictionary for easier access to its key-value pairs.
  `{{ (request.data.incident_string | toDict).id }}`
  Use `toDict` when handling string-based data that needs to be parsed into a dictionary format to access specific fields.
- **xml_to_dict**: Converts an XML string into a dictionary for easy access to the elements.
  `{{ '<?xml version="1.0" ?><person><name>john</name><age>20</age></person>' | xml_to_dict }}`
  Use `xml_to_dict` when you need to parse XML strings and work with their contents in dictionary form.
- **extract_artifacts**: Parses and extracts a list of Indicators of Compromise (IOCs) from a given string:
  `{{'abc.com 1xx.xxx.xx.xx' | extract_artifacts}}`
  Use `extract_artifacts` when parsing strings to identify IOCs like IP addresses, domains, or URLs.
- **parse_cef**: Parses a given CEF (Common Event Format) string and converts it into a dictionary:
  `{{ 'some string containing cef' | parse_cef }}`
  Use `parse_cef` when working with CEF logs, enabling you to convert the event data into a more accessible dictionary format for further processing.
- **readfile**: Fetches the contents of a file that is downloaded in FortiSOAR:
  `{{ vars.steps.<step_name>.keyword| readfile}}`
  Use `readfile` to read the contents of files downloaded or stored in a particular step, such as logs or reports, for use in subsequent steps.
  For example, `vars.steps.<step_name>.data` will contain the contents of the file.
- **ip_range**: Checks if an IP address is in the specified range:
  `{{vars.ip | ip_range('1xx.xx.x.0/24')}}`
  Note: Use `ip_range` when you need to verify if an IP falls within a specific network range, which can be useful for filtering or validating incoming connections.
- **counter**: Returns the count of each item's occurrence in an array.
  `{{data| counter}}`
  Use `counter` when you need to count occurrences of items in a list, useful for aggregating and analyzing repeated values.
  For example:
  `data: ['apple','red','apple','red','red','pear']`
  `{{data| counter}}`
  Result: `{'red': 3, 'apple': 2, 'pear': 1}`
- **resolveRange**: Maps a given value to a corresponding value in a dictionary based on exact matches or specified ranges.
  `{{data| resolveRange(vars)}}`
  Use `resolveRange` for conditional logic based on values or ranges. Useful for dealing with ranges of numeric data or predefined categories.
  For example:
  Data:

```
{"vars": {
   ""<2": "Less than 2",
   ">2000": "Greater than 2000",
   "key2": "exact key match",
   "5,6,7": "Comma separated",
   "50..100": "Range",
   "key3,key4,key5": "Comma separated keys"
  }
}
```

  Result:
  Input => Output

```
{{ "1.5" | resolveRange(vars)}} => "Less than 2"
{{ "3567" | resolveRange(vars)}} => "Greater than 2000"
```

```
{{ "key2" | resolveRange(vars)}} => "exact key match"
{{ "5" | resolveRange(vars)}} => "Comma separated"
{{ "6" | resolveRange(vars)}} => "Comma separated"
{{ "87" | resolveRange(vars)}} => "Range"
{{ "key4" | resolveRange(vars)}} => "Comma separated keys"
```

FortiSOAR also supports following filters, more information for which is present at http://docs.ansible.com/ansible/latest/playbooks_filters.html.

Filters for formatting data

The following filters take a data structure in a template and render it in a slightly different format. These are occasionally useful for debugging:

```
{{ some_variable | to_json }}
{{ some_variable | to_yaml }}
```

- **Convert to human-readable output**:
  ```
  {{ some_variable | to_nice_json }}
  {{ some_variable | to_nice_yaml }}
  ```
- **Change the indentation of variables**:
  ```
  {{ some_variable | to_nice_json(indent=2) }}
  {{ some_variable | to_nice_yaml(indent=8) }}
  ```
- **Read from formatted data**:
  ```
  {{ some_variable | from_json }}
  {{ some_variable | from_yaml }}
  ```

Filters that operate on list variables

Getting Minimum and Maximum Values: Retrieve the smallest or largest number from a list of numbers.
Examples:

- To get the smallest number from a list of numbers:
  ```
  {{ list1 | min }}
  ```
- To get the largest number from a list of numbers:
  ```
  {{ [3, 4, 2] | max }}
  ```

Use `min` or `max` when you need to quickly extract the smallest or largest value from a numerical list.

Filters that return a unique set from sets or lists

- **Get Unique Values from a List**: Removes duplicates from a list and returns a unique set of items:
  ```
  {{ list1 | unique }}
  ```
- **Union of Two Lists**: Combines two lists, ensuring unique items across both.
  ```
  {{ list1 | union(list2) }}
  ```
- **Intersection of Two Lists**: Returns only the items that exist in both lists:
  ```
  {{ list1 | intersect(list2) }}
  ```
- **Difference of Two Lists**: Returns items in the first list that do not exist in the second:
  ```
  {{ list1 | difference(list2) }}
  ```
- **Symmetric Difference of Two Lists**: Returns items that are exclusive to each list:
  ```
  {{ list1 | symmetric_difference(list2) }}
  ```

Use these set operations (unique, union, intersect, difference, symmetric_difference) when working with lists that need to be manipulated to reflect common, exclusive, or combined items.

Random Number and Item Filter

The following filters can be used similar to the default jinja2 random filter (returning a random item from a sequence of items), but they can also be used to generate a random number based on a range.

- **Random Item from a List**: Randomly selects a single item from a list.
  `{{ ['a','b','c'] | random }} # Could return 'a', 'b', or 'c'`
- **Random Number Generation**: Generates a random number from 0 to a supplied end, e.g., random number from 0 to 59:
  `{{ 59 | random }} # Returns a number between 0 and 59`
- **Random Number in Steps**: Generates a random number in steps, e.g., 0 to 100 in steps of 10.
  `{{ 100 | random(step=10) }} # Returns a random number like 30, 40, 50...`
- **Random Number from a Range with a Specific Start and Step**: e.g., a random number between 1 and 100 in steps of 10:
  `{{ 100 | random(start=1, step=10) }} # Returns a number like 10, 20, 30...`
- **Seeded Random Number Generator**: Initializes the random number generator from a seed. Ensures idempotentcy in random number generation by using a seed:
  `{{ 59 |random(seed=inventory_hostname) }}`

Use these random filters to generate random values for use cases like sampling, testing, or creating idempotent playbooks with seeded randomness.

Shuffle Filter

The following filters randomize the order of items in a list, giving a different order every invocation:

- **Random Order**:

```
{{ ['a','b','c']|shuffle }}
# => ['c','a','b']

{{ ['a','b','c']|shuffle }}
# => ['b','c','a']
```

- **Idempotent Shuffle**: Shuffle a list in a reproducible manner using a seed (ensures the same shuffle each time):

```
{{ ['a', 'b', 'c'] | shuffle(seed=inventory_hostname) }} # Returns the same shuffled order
every time for the same inventory_hostname
```

When this filter is used with a non 'listable' item it is a noop. Otherwise, it always returns a list.

Use the `shuffle` filter when you need to randomize the order of a list for use cases like sampling, testing, and use `shuffle(seed=...)` for predictable, idempotent results.

Filters for Mathematical Operations

- **Logarithm**: Returns the logarithm of a number. By default, this is the natural logarithm (base e):
  `{{ myvar | log }} # Returns the natural logarithm of myvar`

Logarithm with Base 10:
`{{ myvar | log(10) }} # Returns the base-10 logarithm of myvar`

- **Power Operation**: Raises a number to a specified power, e.g., power of 2 (or 5):
  `{{ myvar | pow(2) }}`
  `{{ myvar | pow(5) }}`
- **Root Operation**: Returns the square root of a number or any other specified root.
  Square Root:
  `{{ myvar | root }} # Square root of myvar`
  5th Root:
  `{{ myvar | root(5) }} # 5th root of myvar`

Use mathematical filters like log, pow, and root when you need to perform mathematical operations on variables in your playbooks.

IP Address Filters

- **Check Valid IP Address**: Verifies if a string is a valid IP address:
  `{{ myvar | ipaddr }}`
- **Extract IP Version**: Gets the IP address in a specific IP protocol version:
  IPv4:
  `{{ myvar | ipv4 }}`
  IPv6:
  `{{ myvar | ipv6 }}`
- **Extract Specific IP Information from CIDR**: For example, to get the IP address itself from a CIDR, use:
  `{{ '1xx.0.x.1/24' | ipaddr('address') }} # Extracts the IP address '192.0.2.1' from CIDR`
- **Filter a list of IP addresses**:
  - `test_list = ['192.24.2.1', 'host.fqdn', '::1', '192.168.32.0/24', 'fe80::100/10']`
    `{{ test_list | ipv4 }} # Filters out valid IPv4 addresses`
  - `['192.24.2.1/32', '::1/128', 'fe80::100/10']`
    `{{ test_list | ipaddr('host') }} # Retrieves a host IP address from a list of IP addresses`
  - `['192.24.2.1', '2001:db8:32c:faad::/64']`
    `{{ test_list | ipaddr('public') }} #Retrieves a public IP address from a list of IP addresses`
  - `['192.24.2.1/32', '::1/128', 'fe80::100/10']`
    `{{ test_list | ipaddr('private') #Retrieves a private IP address from a list of IP addresses`
  - `['192.24.2.1', '192.168.32.0/24']`
    `{{ test_list | ipaddr('192.0.0.0/8') }} #Network range as a query`

Use IP filters when dealing with network-related data, such as verifying, extracting, or filtering valid IP addresses, and distinguishing between IPv4 and IPv6.

Hashing Filters

Hashing is commonly used to generate fixed-length representations of strings, useful for data integrity, encryption, and checking changes. FortiSOAR supports several popular hashing algorithms.

- Use `password_hash` when dealing with sensitive data (e.g., passwords) to ensure they are securely hashed with salt.

- Avoid using MD5 for security-sensitive operations due to its vulnerability to collision attacks, and prefer SHA256 or SHA512 for stronger security.
- Use hash for generating checksums or non-security-sensitive hashed values.

Filters:

- **SHA1 Hashing**: Generate a SHA1 hash of a string:
  `{{ 'test1' | hash('sha1') }} # Returns the SHA1 hash of 'test1'`
- **MD5 Hashing**: Generate an MD5 hash of a string.
  Note: MD5 is considered insecure for cryptographic purposes, but it is still used for checksums or non-cryptographic purposes:
  `{{ 'test1' | hash('md5') }} # Returns the MD5 hash of 'test1'`
- **Checksum**: Calculate a checksum (e.g., CRC32 or other methods) of a string:
  `{{ 'test2' | checksum }} # Returns the checksum of 'test2'`
- **Other hashes (platform dependent)**: FortiSOAR supports additional hashing algorithms like 'Blowfish', depending on the platform:
  `{{ 'test2' | hash('blowfish') }} # Returns a Blowfish hash of 'test2'`
- **SHA512 Password Hashing (with Salt)**: Generate a password hash using SHA512 with a random salt, making it more secure for password storage):
  `{{ 'passwordsaresecret' | password_hash('sha512') }} # Returns a SHA512 hash of 'passwordsaresecret' with random salt`
- **SHA256 Password Hashing (with Specific Salt)**: Generate a password hash using SHA256 with a specific salt:
  `{{ 'secretpassword' | password_hash('sha256', 'mysecretsalt') }} # SHA256 hash with the salt 'mysecretsalt'`

FortiSOAR uses the `haslib` library for `hash` and `passlib` library for `password_hash`.

Filters for combining hashes and dictionaries

The `combine` filter allows merging of multiple hashes (dictionaries) into a single dictionary, with the possibility of recursively combining nested dictionaries.

- Use combine when you need to merge two or more dictionaries.
- Use `recursive=True` when merging nested dictionaries and you want to combine all nested keys as well.

Filters:

- **Combine Hashes**: Merge one or more dictionaries, where keys in subsequent dictionaries overwrite those in previous ones:
  `{{ {'a': 1, 'b': 2} | combine({'b': 3}) }} # Result: {'a': 1, 'b': 3}`
- **Recursive Combine**: Merge nested dictionaries recursively using the `recursive=True` parameter to not only override keys in the first hash, but also recursively into nested hashes and merge their keys too:
  `{{ {'a': {'foo': 1, 'bar': 2}, 'b': 2} | combine({'a': {'bar': 3, 'baz': 4}}, recursive=True) }} # Result: {'a': {'foo': 1, 'bar': 3, 'baz': 4}, 'b': 2}`
- **Combining Multiple Hashes**: Merges multiple dictionaries. The order of dictionaries matters; the last dictionary will overwrite previous ones:
  `{{ a | combine(b, c, d) }} # Result: combines a, b, c, and d, with d overwriting c, and so on`

Filters for extracting values from containers

The extract filter is useful for pulling specific values from lists or dictionaries. This can be particularly useful for dynamically accessing nested data structures.

- Use the extract filter when you need to extract values from lists or nested data structures, especially when the data can vary or is dynamic in nature.
- Combine map (`'extract', ...`) with dynamic lists of keys or indices to access complex datasets efficiently.

Filters:

- **Extract Values from Lists or Dictionaries**: Extract values from a list of indices or from specific keys in a dictionary:
  From List of Indices:
  `{{ [0, 2] | map('extract', ['x', 'y', 'z']) | list }} # Result: ['x', 'z']`
  From Dictionary:
  `{{ ['x', 'y'] | map('extract', {'x': 42, 'y': 31}) | list }} # Result: [42, 31]`
- **Extract Nested Values**: Perform a recursive lookup by specifying an additional argument (`groups`) in the extract filter:
  `{{ groups['x'] | map('extract', hostvars, 'ec2_ip_address') | list }} # Takes the list of hosts in group ‘x,’ looks them up in hostvars, and then looks up the ec2_ip_address and then extracts IP addresses for the hosts in group 'x'`
- **Extract from Nested Dictionaries**: Extract values from deeply nested dictionaries by chaining keys or using lists:
  `{{ ['a'] | map('extract', b, ['x', 'y']) | list }} # Extracts the value of b['a']['x']['y']`

Comment filter

The `comment` filter allows you to decorate the text with a chosen comment style.
For example, the following data:

```
{{ "Plain style (default)" | comment }}
```

produces the following output:

```
#
# Plain style (default)
#
```

Similarly, you can apply style to the comments for `C (//...)`, `C block (/*...*/)`, `Erlang (%...)` and `XML (<!--...-->)`:

```
{{ "C style" | comment('c') }}
{{ "C block style" | comment('cblock') }}
{{ "Erlang style" | comment('erlang') }}
{{ "XML style" | comment('xml') }}
```

It is also possible to fully customize the comment style:

```
{{ "Custom style" | comment('plain', prefix='#######\n#', postfix='#\n#######\n ###\n #') }}
```

Creates the following output:

```
#######
#
# Custom style
#
#######
```

```
###
 #
```

URL Split filter

The `urlsplit` filter extracts the fragment, hostname, netloc, password, path, port, query, scheme, and username from an URL. If you do not provide any arguments to this filter then it returns a dictionary of all the fields:

- **Extract Hostname**:
  ```
  {{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" | urlsplit
  ('hostname') }}
  # => 'www.acme.com'
  ```
- **Extract Netloc (User info + Hostname + Port)**:
  ```
  {{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" | urlsplit
  ('netloc') }}
  # => 'user:password@www.acme.com:9000'
  ```
- **Extract Username**:
  ```
  {{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" | urlsplit
  ('username') }}
  # => 'user'
  ```
- **Extract Path**:
  ```
  {{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" | urlsplit
  ('path') }}
  # => '/dir/index.html'
  ```
- **Extract Port**:
  ```
  {{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" | urlsplit
  ('port') }}
  # => '9000'
  ```
- **Extract Scheme (Protocol like HTTP, HTTPS)**:
  ```
  {{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" | urlsplit
  ('scheme') }}
  # => 'http'
  ```
- **Extract Query**:
  ```
  {{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" | urlsplit
  ('query') }}
  # => 'query=term'
  ```
- **Extract All Components (Return as a dictionary)**:
  ```
  {{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#frament" | urlsplit }}
  # =>
  # {
  # "fragment": "fragment",
  # "hostname": "www.acme.com",
  # "netloc": "user:password@www.acme.com:9000",
  # "password": "password",
  # "path": "/dir/index.html",
  # "port": 9000,
  # "query": "query=term",
  # "scheme": "http",
  # "username": "user"
  # }
  ```

Regular Expression filters

The Regular Expression (regex) filters provide powerful tools for pattern matching and text manipulation within strings.

- **`regex_search`**: Searches for a pattern within a string. It returns the first match found or an empty result if no match is found:
  Syntax: `{{ string | regex_search('<pattern>') }}`

  ```
  # search for "foo" in "foobar" {{ 'foobar' | regex_search('(foo)') }} # 'foo' (matches the first occurrence of "foo")
  ```

  - Search for a non-existing pattern "foobar" in "ansible":
    `{{ 'ansible' | regex_search('(foobar)') }} # Output: '' (empty as no match found)`
- **`regex_findall`**: Finds all occurrences of the regex pattern in a string and returns them as a list:
  Syntax: `{{ string | regex_findall('<pattern>') }}`
  - Find all IPv4 addresses in a string:
    `{{ 'Some DNS servers are 8.8.8.8 and 8.8.4.4' | regex_findall('\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b') }} # Output: ['8.8.8.8', '8.8.4.4']`
- **`regex_replace`**: Replaces text in a string based on a regex pattern. You can use it to modify or format a string:
  Syntax: `{{ string | regex_replace('<pattern>', '<replacement>') }}`
  - Convert "ansible" to "able" by capturing part of the string:
    `{{ 'ansible' | regex_replace('^a.*i(.*)$', 'a\\1') }}`
    `# Output: 'able'`
  - Convert "foobar" to "bar" by removing the leading 'f' and 'o':
    `{{ 'foobar' | regex_replace('^f.*o(.*)$', '\\1') }}`
    `# Output: 'bar'`
  - Convert "localhost:80" to "localhost, 80" using named groups:
    `{{ 'localhost:80' | regex_replace('^(?P<host>.+):(?P<port>\\d+)$', '\\g<host>, \\g<port>') }}`
    `# Output: 'localhost, 80'`
  - Remove ":80" from "localhost:80" to get just "localhost":
    `{{ 'localhost:80' | regex_replace(':80') }}`
    `# Output: 'localhost'`
- **`regex_escape`**: Escapes special characters in a regex pattern, which is useful when you need to use a string as a literal pattern instead of a regular expression.
  Syntax: {{ string | regex_escape() }}
  - Escape a regex pattern
    `{{ '^f.*o(.*)$' | regex_escape() }} # Output: '\^f\.\*o\(\.\*\)\$' (escaped version of the regex pattern)`

Using regex in Playbooks:

- **Validating Input**: Use `regex_search` to validate if a user input matches a specific pattern. For example, validating an IP address or email address.
  `{{ 'user@example.com' | regex_search('^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$') }}`
  `# Output: 'user@example.com' (if it matches the email pattern)`
- **String Manipulation**: Use `regex_replace` to format strings in specific ways, such as changing date formats or removing unwanted characters.
  `{{ '2023-04-01' | regex_replace('(\d{4})-(\d{2})-(\d{2})', '\\3/\\2/\\1') }}`
  `# Output: '01/04/2023' (reformat date)`

- **Extracting Multiple Values**: If you need to extract all matches from a text, use `regex_findall` to gather all the relevant values.
  ```
  {{ 'The servers are 192.168.1.1 and 10.0.0.2' | regex_findall('\d+\.\d+\.\d+\.\d+') }}
  # Output: ['192.168.1.1', '10.0.0.2'] (extract IP addresses)
  ```

Other useful filters

The additional useful filters in FortiSOAR offer a wide range of functionality to work with strings, paths, dates, Base64 encoding/decoding, UUIDs, and more.

- **`quote`**: Adds quotes around a string, which is useful when preparing strings for shell usage or commands:
  Syntax: `{{ string_value | quote}}`
  Example: `{{ "example" | quote }} # Output: '"example"'`
- **`ternary`**: This is a conditional filter that allows you to use one value if a condition is true and another value if the condition is false:
  Syntax: `{{ condition | ternary(value_if_true, value_if_false) }}`
  Example: `{{ (name == "John") | ternary('Mr','Ms') }} # Output: 'Mr' (if name is "John")`
- **`join`** : Concatenates a list into a string, with an optional separator between elements:
  Syntax: `{{ list | join(separator) }}`
  Example: `{{ ['a', 'b', 'c'] | join(", ") }} # Output: 'a, b, c'`
- **`basename`**: Extracts the last part of a file path, such as the file name from the full path.
  Syntax: `{{ path | basename }}`
  Example: `{{ '/etc/asdf/foo.txt' | basename }} # Output: 'foo.txt'`
- `win_basename`: Extracts the last part of a Windows-style file path:
  Syntax: `{{ path | win_basename }}`
  Example: `{{ 'C:\\path\\to\\file.txt' | win_basename }} # Output: 'file.txt'`
- **`win_splitdrive`**: Splits a Windows-style file path into two parts: the drive letter and the rest of the path:
  Syntax: `{{ path | win_splitdrive }}`
  Example: `{{ 'C:\\path\\to\\file.txt' | win_splitdrive }} # Output: ['C:', '\\path\\to\\file.txt']`
  - **`win_splitdrive| first`**: Returns only the windows drive letter:
    Syntax: `{{ path |win_splitdrive| first }}`
  - **`win_splitdrive| last`**: Returns rest of the path without the drive letter:
    Syntax: `{{ path |win_splitdrive| last}}`
- **`dirname`** : Extracts the directory part of a path, removing the file name:
  Syntax: `{{ path | dirname }}`
  Example: `{{ '/etc/asdf/foo.txt' | dirname }} # Output: '/etc/asdf'`
- **`win_dirname`** : Extracts the directory from a Windows-style path:
  Syntax: `{{ path | win_dirname }}`
  Example: `{{ 'C:\\path\\to\\file.txt' | win_dirname }} # Output: 'C:\\path\\to'`
- **`expanduser`**: Expands a path that contains the tilde (~), which represents the user's home directory:
  Syntax: `{{ path | expanduser }}`
  Example: `{{ '~/myfile.txt' | expanduser }} # Output: '/home/user/myfile.txt' (or similar depending on the OS)`
- **`realpath`**: Resolves a symbolic link to its real path:
  `{{ path | realpath }}`
  Example: `{{ '/path/to/symlink' | realpath }} # Output: '/path/to/realfile'`

- **`relpath`** : Returns the relative path between the current path and a reference path:
  Syntax: `{{ path | relpath (start_path)}}`
  Example: `{{ '/etc/ssh/sshd_config' | relpath('/etc') }} # Output: 'ssh/sshd_config'`
- **`splitext`**: Splits a path into its root and extension:
  Syntax: `{{ path | splitext }}`
  Example: `{{ 'nginx.conf' | splitext }} # Output: ('nginx', '.conf')`
- **`b64decode`** and **`b64encode`**:Used for Base64 encoding and decoding.
  Syntax:
  `{{ encoded | b64decode }}`
  `{{ decoded | b64encode }}`
  Examples:
  Decoding Base64: `{{ 'aGVsbG8gd29ybGQ=' | b64decode }} # Output: 'hello world'`
  Encoding Base64: `{{ 'hello world' | b64encode }} # Output: 'aGVsbG8gd29ybGQ='`
- **`to_uuid`** : Generates a UUID (Universally Unique Identifier) from a string:
  Syntax: `{{ string | to_uuid }}`
  Example: `{{ 'hostname' | to_uuid }} # Output: a unique UUID based on 'hostname'`
- **`to_datetime`**: Converts a string into a date object. It can also be used to calculate the difference between two dates:
  Syntax: `{{ string | to_datetime(format) }}`
  Example: Calculate the difference in seconds between two dates. The default date format is `%Y-%m-%d %H:%M:%S` but you can pass your own format:
  `{{ (("2016-08-14 20:00:12" | to_datetime) - ("2015-12-25" | to_datetime ('%Y-%m-%d'))).seconds }} # Output: seconds between the two dates`

Permutations and Combination filters

These filters are helpful for generating ordered or unordered sets from a list of elements.

- **`permutations`**: Returns all possible ordered arrangements. If a length is specified, only permutations of that length are returned:
  `{{ [1,2,3] |permutations|list }}`
  `# Output: [(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)]`

  To get the permutations of sets of two:
  `{{ [1,2,3] |permutations(2)|list }}`
  `# Output: [(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)]`
- **`combinations`**: Returns unordered subsets of a specified size; they always require a set size.
  To get the combinations for sets of two:
  `{{ [1,2,3,4] |combinations(2)|list }}`
  `# Output: [(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)]`
- **`zip`**: Combine elements of multiple lists into tuples by *index*. `zip` stops at the shortest list (like Python's built-in `zip()`).
  `{{ [1,2,3] |zip(['a','b','c'])|list }}`
  `# Output: [(1, 'a'), (2, 'b'), (3, 'c')]`
- **`zip_longest`**: Continues until the longest list is exhausted; missing values are filled with `fillvalue`, such as 'X' in the following example:
  `{{ [1,2] |zip_longest(['a','b','c'], ['10', '11', '12'], fillvalue='X')|list }}`
  `# Output: [(1, 'a', 10), (2, 'b', 11), ('X', 'c', 12), ('X', 'X', 13)]`

Date Formatting with `strftime`

The `strftime` filter formats a date string from an epoch time (UNIX timestamp) or the current time.

Examples:

- Display year-month-day:
  `{{ '%Y-%m-%d' | strftime }} # Output (today's date): '2025-10-01'`
- Display hour:min:sec:
  `{{ '%H:%M:%S' | strftime }} # Output: '13:45:22' (current time)`
- Use ansible_date_time.epoch:
  `{{ '%Y-%m-%d %H:%M:%S' | strftime(1441357287) }} # Output: '2015-09-04 08:41:27'`
- Use arbitrary epoch value:
  `{{ '%Y-%m-%d' | strftime(0) }} # Output: '1970-01-01' (UNIX epoch start)`
  `{{ '%Y-%m-%d' | strftime(1441357287) }} # Output: ' 2015-09-04'`

Debugging filters

Use the 'type_debug' filter to display the underlying Python type of a variable. This can be useful in debugging in cases where you might need to know the exact type of a variable:
`{{ myvar | type_debug }}`

FortiSOAR also supports following built-in filters from Jinja, more information for which is present at Template Designer Documentation — Jinja Documentation (2.11.x).

- **`abs`** (*number*): Returns the absolute value of the argument.
- **`attr`** (*obj*, *name*): Gets an attribute of an object. `foo|attr("bar")` works like `foo.bar` just that always an attribute is returned and items are not looked up.
  See Notes on subscriptions for more details.
- **`batch`** (*value*, *linecount*, *fill_with=None*): Batches items. It works pretty much like slice just the other way around. It returns a list of lists with the given number of items. If you provide a second parameter, this is used to fill up missing items. For example:
  ```
  {% for row in items|batch(3, 'FillerString') %}
     {% for column in row %}
       {{ column }}
     {% endfor %}
  {% endfor %}
  ```
- **`capitalize`**(*s*): Capitalizes a value. The first character will be uppercase, all others lowercase.
- **`center`**(*value, width=80*): Centers the value in a field of a given width.
- **`default`**(*value, default_value=u'', boolean=False*): If the value is undefined it will return the passed default value, otherwise the value of the variable:
  `{{ my_variable|default('my_variable is not defined') }}`
  This would output the value of `my_variable` if the variable was defined. Otherwise, `my_variable is not defined`.
  If you want to use default with variables that evaluate to false, you have to set the second parameter to true:
  `{{ ''|default('the string was empty', true) }}`
- **`dictsort`**(*value, case_sensitive=False, by='key'*): Sorts a dict and yields (key, value) pairs. Because python dicts are unsorted you might want to use this function to order them by either key or value:
  - Sort the dict by key, case insensitive:
    `{% for item in mydict|dictsort %}`

  - Sort the dict by key, case sensitive:
    `{% for item in mydict|dictsort(true) %}`
  - sort the dict by value, case insensitive:
    `{% for item in mydict|dictsort(false, 'value') %}`
- **`escape`**(*s*): Converts the characters &, <, >, ‘, and ” in strings to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.
- **`filesizeformat`**(*value, binary=False*): Formats the value like a human-readable file size (i.e. 13 kB, 4.1 MB, 102 Bytes, etc). Per default decimal prefixes are used (Mega, Giga, etc.) if the second parameter is set to `True` the binary prefixes are used (Mebi, Gibi).-
- **`first`**(*seq*): Returns the first item of a sequence.
- **`float`**(*value, default=0.0*): Converts the value into a floating point number. If the conversion doesn’t work, it will return `0.0`. You can override this default using the first parameter.
- **`forceescape`**(*value*): Enforces HTML escaping. This will probably double escape variables.
- **`format`**(*value, args,*`**kwargs`): Applies python string formatting on an object:
  ```
  {{ %s"|format("Hello?", "Foo!") }}
     -> Hello? - Foo!
  ```
- **`groupby`**(*value, attribute*): Groups a sequence of objects by a common attribute. If you, for example, have a list of dicts or objects that represent persons with `gender`, `first_name`, and `last_name` attributes and you want to group all users by genders you can do something like the following snippet:
  ```
  {% for group in persons|groupby('gender') %}}
       {{ group.grouper }}
    {% for person in group.list %}
      {{ person.first_name }} {{ person.last_name }}
  {% endfor %}
  ```
  Additionally, it is possible to use tuple unpacking for the grouper and list:
  ```
  {% for grouper, list in persons|groupby('gender') %}
        ...
  {% endfor %}
  ```
  As you can see the item, we are grouping by are stored in the `grouper` attribute, and the list contains all the objects that have this grouper in common. You can also use dotted notation to group by the child attribute of another attribute.
- **`indent`**(*s, width=4, indentfirst=False*): Returns a copy of the passed string, each line indented by four spaces. The first line is not indented. If you want to change the number of spaces or indent the first line too you can pass additional parameters to the filter:
  ```
  {{ mytext|indent(2, true) }}
    indent by two spaces and indent the first line too.
  ```
- **`int`**(*value, default=0, base=10*): Converts the value into an integer. If the conversion does not work, it will return 0. You can override this default using the first parameter. You can also override the default base (10) in the second parameter, which handles input with prefixes such as 0b, 0o and 0x for bases 2, 8 and 16 respectively. The base is ignored for decimal numbers and non-string values.
- **`join`**(*value, d=u'', attribute=None*): Returns a string which is the concatenation of the strings in the sequence. The separator between elements is an empty string per default; you can define it with the optional parameter:
  ```
  {{ [1, 2, 3] |join('|') }}
       -> 1 |2|3

  {{ [1, 2, 3] |join }}
       -> 123
  ```
  It is also possible to join certain attributes of an object:
  ```
  {{ users|join(', ', attribute='username') }}
  ```

- **`last`**(*seq*): Returns the last item of a sequence.
- **`length`**(*object*): Returns the number of items of a sequence or mapping.
  Aliases: `count`
- **`list`**(*value*): Converts the value into a list. If it were a string, the returned list would be a list of characters.
- **`lower`**(*s*): Converts a value to lowercase.
- **`map()`**: Applies a filter on a sequence of objects or looks up an attribute. This is useful when dealing with lists of objects, but you are only interested in a certain value of it.
  The basic usage is mapping on an attribute. Imagine you have a list of users, but you are only interested in a list of usernames:
  Users on this page: `{{ users |map(attribute='username')|join(', ') }}`
  Alternatively, you can let it invoke a filter by passing the name of the filter and the arguments afterward. A good example would be applying a text conversion filter on a sequence:
  Users on this page: `{{ titles |map('lower')|join(', ') }}`
- **`pprint`**(*value, verbose=False*): Pretty print a variable. Useful for debugging. With Jinja 1.2 onwards you can pass it a parameter. If this parameter is truthy, the output will be more verbose (this requires pretty).
- **`random`**(*seq*): Returns a random item from the sequence.
- **`reject()`**: Filters a sequence of objects by applying a test to each object, and rejecting the objects whose tests succeed.
  If no test is specified, each object will be evaluated as a boolean. For example:
  `{{ numbers|reject("odd") }}`
- **`rejectattr ()`**: Filters a sequence of objects by applying a test to the specified attribute of each object, and rejecting the objects whose tests succeed.
  If no test is specified, the attribute's value will be evaluated as a boolean.
  `{{ users|rejectattr("is_active") }}`
  `{{ users|rejectattr("email", "none") }}`
- **`replace`**(*s, old, new, count=None*): Returns a copy of the value with all occurrences of a substring replaced with a new one. The first argument is the substring that should be replaced; the second is the replacement string. If the optional third argument count is given, only the firstcount occurrences are replaced:

```
{{ "Hello World"|replace("Hello", "Goodbye") }}
     -> Goodbye World

{{ "aaaaargh"|replace("a", "d'oh, ", 2) }}
     -> d'oh, d'oh, aaargh
```

- **`reverse`**(*value*): Reverses the object or returns an iterator that iterates over it the other way round.
- **`round`**(*value, precision=0, method='common'*): Round the number to a given precision. The first parameter specifies the precision (default is `0`), the second the rounding method:
  - `'common'` rounds either up or down: Default method.
  - `'ceil'` always rounds up
  - `'floor'` always rounds down

```
{{ 42.55|round }}
     -> 43

 {{ 42.55|round(1, 'floor') }}
     -> 42.5
```

  Note that even if rounded to `0` precision, a float is returned. If you need a real integer, pipe it through `int`

```
{{ 42.55 |round|int }}
     -> 43
```

- **safe**(*value*): Marks the value as safe which means that in an environment with automatic escaping enabled this variable will not be escaped.
- **select()**: Filters a sequence of objects by applying a test to each object, and only selecting the objects whose tests succeed.
  If no test is specified, each object will be evaluated as a boolean. For example:
  `{{ numbers|select("odd") }}`
- **selectattr()**: Filters a sequence of objects by applying a test to the specified attribute of each object, and only selecting the objects whose tests succeed.
  If no test is specified, the attribute's value will be evaluated as a boolean. For example:
  ```
  {{ users|selectattr("is_active") }}
  {{ users|selectattr("email", "none") }}
  ```
- **slice**(*value, slices, fill_with=None*): Slices an iterator and returns a list of lists containing those items.
  Useful if you want to create a div containing three ul tags that represent columns:
  ```
  <div class="columwrapper">
    {% for column in items|slice(3) %}
      <ul class="column-{{ loop.index }}">
        {% for item in column %}
          <li>{{ item }}</li>
        {% endfor %}
      </ul>
    {% endfor %}
  </div>
  ```
  If you pass it a second argument, it is used to fill missing values on the last iteration.
- **sort**(*value, reverse=False, case_sensitive=False, attribute=None*): Sorts an iterable. Per default it sorts ascending, if you pass it true as the first argument, it will reverse the sorting.
  If the iterable is made of strings, the third parameter can be used to control the case sensitiveness of the comparison which is disabled by default.
  ```
  {{% for item in iterable|sort %}}
        ...
  {% endfor %}
  ```
  It is also possible to sort by an attribute (for example to sort by the date of an object) by specifying the attribute parameter:
  ```
  {% for item in iterable|sort(attribute='date') %}
        ...
  {% endfor %}
  ```
- **string**(*object*): Makes a string unicode if it isn't already. That way a markup string is not converted back to unicode.
- **striptags**(*value*): Strips SGML/XML tags and replace adjacent whitespace by one space.
- **sum**(*iterable, attribute=None, start=0*): Returns the sum of a sequence of numbers plus the value of parameter 'start' (which defaults to 0). When the sequence is empty, it returns to start.
  It is also possible, to sum up only certain attributes:
  Total: `{{ items|sum(attribute='price') }}`
  The *attribute* parameter was added to allow summing up over attributes. Also, the *start* parameter was moved on to the right.
- **title**(*s*): Returns a titlecased version of the value, i.e., words will start with uppercase letters, all remaining characters are lowercase.
- **tojson** or **toJSON** (*value, indent=None*): Dumps a structure to JSON so that it's safe to use in `<script>` tags. It accepts the same arguments and returns a JSON string. Note that this is available in templates through the `|tojson` filter which will also mark the result as safe. Due to how this function escapes certain

characters this is safe even if used outside of `<script>` tags.
The following characters are escaped in strings: `<>&'`
This makes it safe to embed such strings in any place in HTML with the notable exception of double quoted attributes. In that case single quote your attributes or HTML escape also.
The indent parameter can be used to enable pretty printing. Set it to the number of spaces that the structures should be indented with.
Note that this filter is for use in HTML contexts only.

- **`trim`**(*value*): Strips the leading and trailing whitespace.
- **`truncate`**(*s, length=255, killwords=False, end='...', leeway=None*): Returns a truncated copy of the string. The length is specified with the first parameter which defaults to `255`. If the second parameter is `true`, the filter will cut the text at length. Otherwise, it will discard the last word. If the text was in fact truncated, it would append an ellipsis sign (`"..."`). If you want a different ellipsis sign than `"..."` you can specify it using the third parameter. Strings that only exceed the length by the tolerance margin given in the fourth parameter will not be truncated.

```
{{ "foo bar baz qux"|truncate(9) }}
     -> "foo..."

{{ "foo bar baz qux"|truncate(9, True) }}
      -> "foo ba..."

 {{ "foo bar baz qux"|truncate(11) }}
      -> "foo bar baz qux"

{{ "foo bar baz qux"|truncate(11, False, '...', 0) }}
      -> "foo bar...
```

  The default leeway on newer Jinja2 versions is 5 and was 0 before but can be reconfigured globally.
- **`upper`**(*s*): Converts a value to uppercase.
- **`urlencode`**(*value*): Escape strings for use in URLs (uses UTF-8 encoding). It accepts both dictionaries and regular strings as well as pairwise iterables.
- **`urlize`**(*value, trim_url_limit=None, nofollow=False, target=None, rel=None*): Converts URLs in plain text into clickable links.
  If you pass the filter an additional integer, it will shorten the URLs to that number. Also, a third argument exists that makes the URLs “nofollow”:

```
{{ mytext|urlize(40, true) }}
     links are shortened to 40 chars and defined with rel="nofollow"
```

  If the *target* is specified, the `target` attribute will be added to the `<a>` tag:

```
{{ mytext|urlize(40, target='_blank') }}
```

- **`wordcount`**(*s*): Counts the words in that string.
- **`wordwrap`**(*s, width=79, break_long_words=True, wrapstring=None*): Returns a copy of the string passed to the filter wrapped after `79` characters. You can override this default using the first parameter. If you set the second parameter to *false* Jinja will not split words apart if they are longer than the *width*. By default, the newlines will be the default newlines for the environment, but this can be changed using the `wrapstring` keyword argument.
- **`xmlattr`**(*d, autospace=True*): Creates an SGML/XML attribute string based on the items in a dict. All values that are neither none nor undefined are automatically escaped

```
<ul{{ {'class': 'my_list', 'missing': none,
     'id': 'list-%d' |format(variable)}|xmlattr }}>
...
</ul>
```

  Results in something like this:

```
<ul class="my_list" id="list-42">
...
</ul>
```

As you can see it automatically prepends a space in front of the item if the filter returned something unless the second parameter is false.

json_query filter

Use the `json_query` filter when you have a complex data structure in the JSON format from which you require to extract only a small set of data. The `json_query` filter enables you to easily query nested and complex JSON structure. The filter is built using `JMESPATH`, and you can use the same syntax in the `json_query` filter. For details on `JMESPATH`, see JMESPath Examples.

Examples

- The result of your playbook step is as follows:
  ```
  [ {"name": "a", "state": "running"},
  {"name": "b", "state": "stopped"},
   {"name": "b", "state": "running"} ]
  ```
  From this result, you want to query only the names of those objects who are in the `running` state. For this query the valid jinja expression would be:
  ```
  {{ vars.steps.<step_name>.keyname | json_query(" [?state=='running'].name") }} #Output:
  ["a","b"]
  ["a","b"]
  ```
  Replace `<step_name>` and `keyname` with your actual step name and key used in FortiSOAR.
- **Filter and extract specific field**:
  ```
  {{ my_json_data | json_query("[?status=='success'].id") }}
  ```
- **Access nested fields**:
  ```
  {{ my_json_data | json_query("users[*].details.email") }}
  ```
- **Select objects with multiple conditions**:
  ```
  {{ my_json_data | json_query("[?type=='admin' && active==`true`].username") }}
  ```

To create a valid JSON query, you can refer to JMESPath Tutorial.

Comprehensive list of filters

The following table contains a comprehensive list of filters that you can use:

Filter	Description	Source
`abs`	Absolute value of a number	jinja2_docs
`attr(x)`	Gets attribute 'x' of an object. Does not return *items*, which are dictionary keys	jinja2_docs
`b64decode`	Decodes a base64 value	ansible
`b64encode`	Encodes a value as base64	ansible
`basename`	Last name in a filepath	ansible

`batch(n)`	Separates a list into various lists with size 'n'	jinja2_docs
`bool`	Casts a string as a boolean value (i.e. "True" or "False" to a boolean value)	ansible
`capitalize`	Capitalizes the first letter of a value	jinja2_docs
`center(n)`	Centers the value in a field of 'n' characters by adding spaces on either side	jinja2_docs
`checksum`	Get a string checksum	ansible
`cidr_merge`	Merges a list of subnets or IP addresses to their minimal representation	ansible_ipaddr
combinations(n)	Returns an iterator of n-size combinations of items from an input list	ansible
`combine(dict_x, recursive=False)`	Merges dect_x into input dictionary, overwriting any values that overlap. Setting recursive=True also allows for nested keys to be merged.	ansible
`comment`	Converts a string into a python-style comment	ansible
`comp_type5`	ansible.netcommon comp_type5 filter plugin	ansible.netcommon
`count`	Alias of length	jinja2_docs
`count_occurrence`	Retrieves the number of times each element appears in the list.	FortiSOAR
`counter`	Gets the count of each item's occurrence in an array of items.	FortiSOAR
`d(x)`	Alias of "default"	jinja2_docs
`default(x)`	Outputs default value 'x' if the passed input is not defined.	jinja2_docs
`dict2items`	Turn a dictionary into a list of items suitable for looping	ansible
`dictsort (false,reverse=false)`	Sorts a dict and yields key, value pairs. Set the first argument to true for case sensitive sort. Provide 'value' as second argument to sort by value instead of key.	jinja2_docs
`difference(list_x)`	Gets items from an input list that are not present in 'list_x'.	ansible
`dirname`	Gets the directory from a path	ansible
`e`	Alias of escape	jinja2_docs
`escape`	Escapes characters &, M, >, ', and " with HTML-safe sequences	jinja2_docs

`expanduser`	Expands a path containing a telde(~) character	ansible
`expandvars`	Expands a path containing environment variables	ansible
`extract()`	Maps a list of indices to a list of values from a container (hash or array)	FortiSOAR
`extract_artifacts`	Parses and extracts a list of IOCs from a given string	FortiSOAR
`extract_cef/parse_cef`	Parses a given CEF string and converts the CEF string into a dictionary	FortiSOAR
`fileglob`	Provides an output list of matching files from the given input path (can include wildcards, for example '/tmp/*.txt').	ansible
`filesizeformat`	Formats a number into "human readable" file size, for example, 13 KB	jinja2_docs
`first`	Returns the first item of a sequence	jinja2_docs
`flatten`	Flattens a list	ansible
`float`	Converts a value into floating point number	jinja2_docs
`forceescape`	Enforces HTML escaping. Can lead to double-escaping	jinja2_docs
`format(string_x)`	Formats 'string_x' based on the passed format string	jinja2_docs
`fromIRI`	Resolve an IRI and return the object(s) that live (s) there. This is similar to loading the object by id (IRI)	FortiSOAR
`from_json`	Converts a json-formatted string to dict.	ansible
`from_yaml`	Converts a yaml-formatted string to dict	ansible
`from_yaml_all`	Parses a multi-document yaml string to an iterator of parsed yaml documents	ansible
`groupby(value)`	Groups a sequence of objects by their attribute value	jinja2_docs
`hash(hashtype)`	Gets the hash of a string, using hashtype, for example, 'md5', or 'sha1'	ansible
`html2texthash`	Converts an HTML string to a text string	FortiSOAR
`human_readable`	Asserts whether the given string is human readable or not	ansible
`human_to_bytes`	Returns the given string in bytes format	ansible
`hwaddr`	Checks if a string is a MAC address	ansible_ipaddr

`indent`	Returns the input string with each line indented by four spaces	jinja2_docs
`int`	Converts the value to an integer	jinja2_docs
`intersect(list_x)`	Gets a list of unique items that is present in both the 'input list' and 'list_x'	ansible
`ip4_hex`	Converts IPv4 to an Hexadecimal notation	ansible_ipaddr
`ip_range(ip_range)`	Checks if the IP address is in the specified CIDR range	FortiSOAR
`ipaddr`	Checks if a string is a valid IP address	FortiSOAR
`ipmath(n)`	Gets the next 'n' addresses based on the passed parameter in specified IP address	ansible_ipaddr
`ipsubnet`	Converts an ip address to a subnet	ansible_ipaddr
`ipv4`	Checks if the given IP address is an IPv4 address	ansible_ipaddr
`ipv6`	Checks if the given IP address is an IPv6 address	ansible_ipaddr
`ipwrap`	Wraps any IPv6 addresses in brackets in the provide a list of strings, leaving other items intact	ansible_ipaddr
`items2dict(key=k, value=v)`	Reverse of dict2items, i.e., maps key and value into a dictionary	ansible
`join(delim_x)`	Returns a string that is the concatenation of the strings in the passed sequence. If 'delim_x' is provided, it is used to separate the items in the string	jinja2_docs
`json2html`	Converts JSON data into HTML	FortiSOAR
`json_query`	Allows the use of jmespath expressions (see jmespath.org) to manipulate input data	ansible
`last`	Returns the last item of a sequence	jinja2_docs
`length`	Returns the number of items in a sequence	jinja2_docs
`list`	Converts the value into a list	jinja2_docs
`loadRelationships`	Fetches details of a related (correlation) record	FortiSOAR
`log(base=e)`	Gets the log (default base e) of the passed values	ansible
`lower`	Converts a value to lowercase	jinja2_docs
`mandatory`	Raises an error if the passed variable is undefined	ansible

`map`	Applies a filter on a sequence of objects or looks up an attribute	jinja2_docs
`max`	Returns the largest item from the sequence	jinja2_docs
`md5`	Gets the md5 hash of a string	ansible
`min`	Returns the smallest item from the sequence	jinja2_docs
`network_in_network`	Returns whether 'address_x' is in the passed network	ansible_ipaddr
`network_in_usable`	Returns whether an address passed as an argument is usable in a network	ansible_ipaddr
`next_nth_usable(n)`	Returns the next 'n' usable IP addresses in relation to the passed IP addresses/ranges	ansible_ipaddr
`nthhost(n)`	Returns the nth IP address in the passed CIDR range	ansible_ipaddr
`parse_cli`	Converts the output of a network device CLI command into a structured JSON output	ansible
`parse_cli_textfsm`	Parses output of a network device CLI command using the TextFSM library	ansible
`parse_xml(path_to_specfile)`	Converts XML output of a network device command into a structured JSON output	ansible
`password_hash(algorithm, salt)`	Gets a password hash with a specified hashing algorithm, and optionally a provided salt value	ansible
`permutations`	Gets an iterator of all permutations of values in a list	ansible
`picklist`	Loads the specified picklist item object	FortiSOAR
`pow(x)`	Returns passed values to the power of 'x'	ansible
`pprint`	Pretty prints the passed variable	jinja2_docs
`previous_nth_usable`	Returns the previous 'n' usable IP addresses in relation to the passed IP addresses/ranges	ansible_ipaddr
`product(iterable_x)`	Returns the cartesian product of the passed iterable with 'iterable_x'	ansible
`quote`	Wrap the passed string in quotes	ansible
`random`	Returns a random item from the passed sequence	jinja2_docs
`random_mac`	Generates a random MAC address from the passed string prefix	ansible
`readfile`	Fetches the contents of a file that is downloaded in FortiSOAR.	FortiSOAR

`realpath`	Gets the real path of a link	ansible
`reduce_on_network(ip_range)`	Checks whether multiple addresses belongs to a network	ansible_ipaddr
`regex_escape`	Escapes special characters within the passed standard pythton regex	ansible
`regex_findall(regex_pattern)`	Searches for all occurrences of regex matches in the passed string	ansible
`regex_replace(regex_to_replace, replacement_regex)`	Replaces text in the passed string using regex	ansible
`regex_search(regex_to_find)`	Finds the first occurrence of regex_to_find in the passed string	ansible
`reject(test)`	Filters the passed list, removing elements where 'test_x' succeeds	jinja2_docs
`rejectattr(attribute_x)`	Filters the passed list removing elements where 'attribute_x' evaluates as true	jinja2_docs
`relpath(start_point)`	Gets the relative path of the passed link from the 'start_point'	ansible
`replace(substr_to_replace)`	Returns a copy of the passed values with all occurrences of 'substr_to_replace' replaced with the 'new_substring'	jinja2_docs
`reverse`	Reverses the passed object or returns an iterator that iterates over the passed object in the reverse order	jinja2_docs
`root(x)`	Returns the 'x' root of the passed value	ansible
`round(precision)`	Rounds the passed number to the given precision (default 0)	jinja2_docs
`safe`	Marks the provided value as safe	
`select(test)`	Filters the passed sequence, keeping only the objects for which the test succeeds	jinja2_docs
`selectattr(attribute_x)`	Filters the passed sequence, keeping only the objects for which 'attribute_x' evaluates as true	jinja2_docs
`sha1`	Gets the sha1 hash of the passed string	ansible
`shuffle`	Randomizes the passed list	ansible
`slaac`	Generates an IPv6 address for a given network and MAC address in stateless configuration	ansible_ipaddr
`slice`	Slices an iterator and return a list containing those items	jinja2_docs

`sort`	Sorts an iterable using Python's sorted() function	jinja2_docs
`splitext`	Gets the root and extension of the passed path or filename	ansible
`strftime`	Formats a date using the passed date format string	ansible
`string`	Converts an object to a string if it is not already a string	jinja2_docs
`striptags`	Strips XML/SGML tags and replaces adjacent whitespaces with one space	jinja2_docs
`subelements`	Produces a product of the passed list and a subelement of the objects in that list	ansible
`sum`	Returns the sum of the passed sequence of numbers	jinja2
`symmetric_difference (list_x)`	Returns the items exclusive to 'list_x' and the passed list	ansible
`ternary(output_1, output_2)`	Returns 'output_2' if the passed value is false, and 'output_1' if it is true	ansible
`title`	Return a titlecased version of the passed value	jinja2_docs
`toDict`	Converts a string into a dictionary	FortiSOAR
`toJSON`	Dumps a structure to a JSON string	FortiSOAR
`to_datetime`	Gets a date object from a string	ansible
`to_json`	Converts a data structure to a JSON format	ansible
`to_nice_json`	Converts a data structure to a human-readable JSON format	ansible
`to_nice_yaml`	Converts a data structure to human-readableYAML format	ansible
`to_uuid`	Creates a UUID from a string	ansible
`to_yaml`	Converts a data structure to a YAML format	ansible
`tojson`	Alias of toJSON	FortiSOAR
`trim`	Strips leading and trailing characters, by default whitespace	jinja2_docs
`truncate(n)`	Returns a truncated copy of the passed string; truncated to length 'n'	jinja2_docs
`type_debug`	Displays the underlying Python type of the passed variable	FortiSOAR

`union(list_x)`	Get the union of 'list_x' with the passed list	ansible
`unique`	The list of unique items in the passed list	jinja2_docs
`upper`	Converts the passed string to uppercase	jinja2_docs
`urldecode`	Decodes the passed URL	FortiSOAR
`urlencode`	Escapes the strings for use in URLs	FortiSOAR
`urlize`	Converts URLs into clickable links	jinja2_docs
`urlsplit`	Extracts the fragment, hostname, netloc, password, patht, port, query, scheme, and username from a URL	ansible
`vlan_parser`	Transforms the passed unsorted list of VLAN integers into a sorted string list of integers according to IOS-like VLAN list rules	ansible
`win_basename`	Gets the last name of a windows-style file path	ansible
`win_dirname`	Gets the directory from a windows path	ansible
`win_splitdrive`	Separate the windows drive letter from the rest of a file path	ansible
`wordcount`	Counts the words in the passed string	jinja2_docs
`wordwrap(n)`	Wraps the given string to width 'n'	jinja2_docs
`xml_to_dict`	Converts an XML string into a dictionary	FortiSOAR
`xmlattr`	Creates an XML attribute string based on the items in the passed dict	jinja2_docs
`yaql`	YAQL (Yet Another Query Language) is an embeddable and extensible query language, which allows users to perform complex queries against arbitrary objects. For more information, see YAQL Filters.	
`zip(list_x)`	Get a list by combining the items from the passed list with those from 'list_x'	ansible
`zip_longest`	Like 'zip' but always exhausts all input lists	ansible

Notes:

- If an iterator is returned, pass the output into the "list" filter to get a list.

Sources

- jinja2 - https://tedboy.github.io/jinja2/templ14.html
- ansible- https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html
- ansible_ipaddr - https://docs.ansible.com/ansible/latest/collections/ansible/utils/docsite/filters_ipaddr.html
- jinja2_docs - https://jinja.palletsprojects.com/en/3.1.x/templates/#builtin-filters

Jinja Expressions

Following are some examples of jinja expressions used in FortiSOAR:

For Loop

At times, it is useful to use a combination of conditional logic and looping to check particular conditions across a list of values. In this case, the `for` and the `if` loop is useful in the Dynamic Variable language.

In the following example, an evaluation of assignment is run across a specific dictionary representing all associated teams within a particular record. These teams have already been assigned to a particular variable from the parent entity:

```
{% for item in vars.teamName %}
       {% if item == '/api/3/teams/97fd5a3f-4eaf-4cc1-a132-1c9274bd8428' %}
       Yes  {% endif %}
{% endfor %}
```

If Condition

```
{% if 1485561600000 > 1484092800000 %}
        {{vars.input.records[0]}}
            {% elif 5==6 %}
        {{vars.input.records[0]}}
{% endif %}
```

An `if` condition can cause a playbook to fail if the jinja that you have added returns an empty string, which is not compatible with the field datatype defined in the database. For example, if you have added the following jinja to a `{% if vars.currentValue == "Aftermath" %}{{@Current_Date}}{% endif %}` field, then the playbook will fail if the jinja returns an empty string.

To ensure that your playbook does not fail due to the issue of jinja returning an empty string, add the following jinja in the field: `{% if vars.currentValue == “Aftermath” %}{{Current_Date}}{% else %} None {% endif %}`.

For Loop along with the If condition

At times, it can be useful to use a combination of conditional logic and looping during a check of particular conditions across a list of values. In this case, the `for` and the `if` loop is useful in the Dynamic Variable language.

```
{% for i in vars.var_list  %}
        { %  if i not in vars.var_response % }
            {{vars.var_response.append(i)}}
        {% endif %}
{% endfor %}
```

The following example uses the `for` and the `if` loop to check if a particular team is tagged to a record. In the following example, an evaluation of assignment is run across a specific dictionary representing all of the associated teams within a particular record. These teams have already been assigned to a particular variable from the parent entity.

```
{% for item in vars.teamName %}
       {% if item == '/api/3/teams/97fd5a3f-4eaf-4cc1-a132-1c9274bd8428' %}
       Yes  {% endif %}
{% endfor %}
```

If Else condition

`If` conditions within the Dynamic Variable templating engine can be very useful to avoid unnecessary decision steps. These conditions allow you to specify values defined within specific conditions such that copying, or value branches are not needed.

The following example uses the if else condition to create mapping between Alert Severity and Incident Category.

Here is an example of a particular case in which the value of an IRI is defined based upon the specific conditions evaluated during the execution. Bear in mind these IRI values must be known in this case.

```
{% if vars.alertSeverity == "/api/3/picklists/ad5eacc1-7c05-3c4e-bba7-eb356c8547c9" %}

       /api/3/picklists/58d0753f-f7e4-403b-953c-b0f521eab759

{% elif vars.alertSeverity == "/api/3/picklists/ddbc7842-dde0-392a-ae94-a1e7a3c7c2f7" %}

       /api/3/picklists/40187287-89fc-4e9c-b717-e9443d57eedb

{% elif vars.alertSeverity == "/api/3/picklists/6c7a8653-a2d5-3611-bc86-8ca307a02e88" %}

       /api/3/picklists/7efa2220-39bb-44e4-961f-ac368776e3b0

{% endif %}
```

Time Operations

- **Parse a timestamp**:
  `{{ arrow.get('2013-05-30 12:30:45', 'YYYY-MM-DD HH:mm:ss') }}`
- **Get current UTC time in epoch (milliseconds)**:
  `{{arrow.utcnow().timestamp*1000 |int| abs}}`
- **Convert a static date to epoch time**:
  `{{ arrow.Arrow(2017, 3, 30).timestamp}}`
- **Convert a date from UTC to another timezone with formatting**: Use the Arrow library to convert dates and times from one-time zone/format to another. The general format is:
  `{{ arrow.get( % VARIABLE % ).to('% TIME ZONE ACRONYM %').format('% FORMAT STRING%') }}`
  The following is an example that is converting the Date of Compromise field into a readable Eastern Standard Time (EST) format:
  `{{arrow.get(vars.input.records[0].dateOfCompromise).to('EST').format('YYYY-MM-DD HH:mm:ss`

ZZ')}}
`to('EST')`: Converts from UTC to Eastern Standard Time.
`.format(...)`: Customizes the output format.

- **Shift time Forward**: To convert timezone from any to UTC and shift the time forward. The general format is:
  `{{ arrow.get( % VARIABLE %).replace( % TIME VALUE % = % OPERATOR + VALUE %) }}`
  An example where the Alert Time value is replaced with a four-hour increase:
  `{{ arrow.get(vars.alertTime).shift(hours=+4) }}`
- **Convert a variable UTC time into epoch for API**: In this example, a particular UTC time is converted into epoch time in order to format it for API insertion. The API will accept times in epoch as the default:
  `{{ arrow.get(vars.utcTime).timestamp }}`

String Operations

Some useful string manipulation operations you can perform in FortiSOAR with Jinja expressions:

- **Find the length of list or string**:
  `{{vars.emails | length }}`
- **Replace a string**: Use the `replace` filter to modify parts of a string. The following example removes the "dict_keys(" and ")" from the string:
  `{{ vars.var_keys.replace("dict_keys(","" ) | replace( ")", "" )}}`
- **Strip the first X characters of a string**: Starting with a string variable, you can slice a string to get only the first X characters. The general format for this Jinja expressions is as follows where # is the number of characters:
  `{{ % VARIABLE %[:#] }}`
  An example here pulls the first 17 characters of the string `timeRange`:

```
timeRange = 2016-09-02 13:45:00 EDT - 2016-09-02 14:00:00 ET
alertTime = {{vars.timeRange[:17]}}

print(alertTime) 2016-09-02 13:45:00
```

- **Remove HTML tags**: The `striptags()` function is used to remove the HTML tags from a string, which may be present in rich text or other HTML formats. The function preserves the data contained within the tags and removes any tagged information contained around the actual values.
  `{{ vars.input.records[0].name.striptags() }}`
  If name contains `<p>This is <b>bold</b> text.</p>`, the output will be: `This is bold text.`

Code in block

```
{% block body %}
        {% for key, value in vars.loop_resource.items() %}
            {{ key }}: {{ value }}
        {% endfor %}
{% endblock %}
```

Set variable based on condition

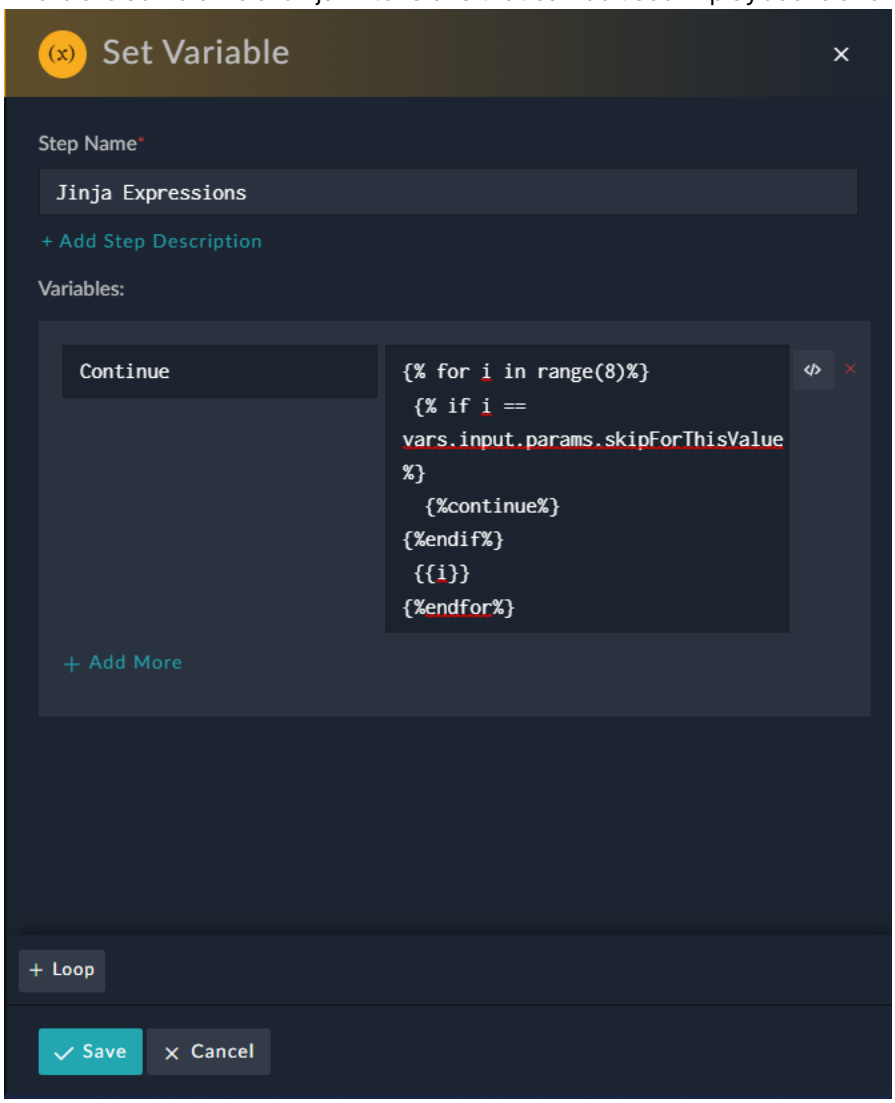
```
{% for i in vars.steps<step_name>['hydra:member'] %}
        {% set id = i['@id'] %}
        {{ vars.inc_fdata.append(id) }}
{%endfor%}
```

Second example:

```
{% for i in vars.steps<step_name>['hydra:member'] %}
        {% set id = i['@id'] %}
        {% set createDate = i.createDate | string %}
        {% set list_item = [id,createDate] %}
            {{ vars.inc_fdata.append(list_item) }}
{%endfor%}
```

Jinja Extensions

There are some official Jinja Extensions that can be used in playbooks and help in enriching expressions:



Note: Support to '**break**' and '**continue**' ansible loops has been added.

Some examples of using expression statements follow:

Appending a list in a loop example:

```
{% for i in range(100000)%}
{{vars.printThis}}
{% do vars.res.append(vars.printThis) %}
{%if i==vars.input.params.breakAfterNumberOfLoops %}
{%break%}
{%endif%}
{%endfor%}
```

Continue Statement example:

```
{% for i in range(8)%}
 {% if i == vars.input.params.skipForThisValue%}
  {%continue%}
{%endif%}
 {{i}}
{%endfor%}
```

Break Statement example:

```
{% for i in range(100000)%}
{{vars.printThis}}

{%if i==vars.input.params.breakAfterNumberOfLoops %}

{%break%}

{%endif%}
```

YAQL and Custom Filters

YAQL Filters

FortiSOAR release 7.2.0 adds support for YAQL as an additional filter language (in addition to JINJA). YAQL (Yet Another Query Language) is an embeddable and extensible query language, which allows users to perform complex queries against arbitrary objects. YAQL contains a comprehensive standard library of frequently used querying functions and can be extended even further with user-specified functions. YAQL is written in python and is distributed via PyPI. For more information on YAQL, see the Getting started with YAQL document.

Usage

In a YAQL query, the `$` symbol is used to refer to "this" object. Therefore, `$.lastname` in a YAQL query pulls the last name of each object in the list passed to the filter.

The `$` symbol can also have different meanings within the same query. For example, in `$.pets.flatten().where($.type='cat')`, the first instance of `$` refers to each of the 'User' objects, while the second instance of `$` refers to each of the 'Pets' objects. In this example, `where()` and `flatten()` are examples of functions available in YAQL. For a list of available YAQL functions see the Standard YAQL Library document.

Usage Examples

- `{{ {"var1":1,"var2":"a"} | yaql('$.var1') }}`
  `returns# 1`
- `{{ "test" | yaql('$.toUpper()') }}`
  `returns# TEST`

Example to filter down to non-false data:
Sample data:

```
{
   "data":{
      "av_cate":"Riskware/NetCat",
      "wf_cate":"",
      "ioc_cate":"",
      "ioc_tags":[

      ],
      "confidence":"High",
      "spam_cates":[

      ],
      "reference_
url":"https://ioc.fortiguard.com/search?query=E8FBEC25DB4F9D95B5E8F41CCA51A4B32BE8674A4DEA7A45B6F7
AEB22DBC38DB&filter=indicator"
   },
   "success":true
}
{{ data | yaql('dict($.items().where(bool($[1])))') }}
returns#
{
    "av_cate": "Riskware/NetCat",
    "confidence": "High",
    "reference_url":
"https://ioc.fortiguard.com/search?query=E8FBEC25DB4F9D95B5E8F41CCA51A4B32BE8674A4DEA7A45B6F7AEB22
DBC38DB&filter=indicator"
}
```

If you have the following example dataset:

```
[{

   "firstname": "Billy",

   "lastname": "gu",

   "email": "billsgu@example.com",

   "pets": []

}, {

   "firstname": "Trevor",

   "lastname": "Palmer",

   "email": "palmer928@example.com",

   "pets": [{

      "name": "Butter",
```

```
      "type": "dog"
   }]
}, {
   "firstname": "Jimmy",
   "lastname": "Bauer",
   "email": "jbles45@example2.com",
   "pets": [{
      "name": "Biscuit",
      "type": "cat"
   }, {
      "name": "Pepper",
      "type": "cat"
   }]
}]
```

You can filter this dataset using YAQL as follows:

- Get Users where firstname starts with B:
  `vars.users | yaql("$.where($.firstname.startsWith('B'))")`
- Get Users where email contains @example.com:
  `vars.users | yaql("$.where('@example.com' in $.email)")`
- Get Users who have pets with type cat:
  `vars.users | yaql("$.where($.pets.where($.type='cat').any())")`

Custom Functions and Filters

FortiSOAR supports following custom functions/filters:

- **`Get_current_date`**: Returns the current date for the file.
- **`Get_current_datetime`**: Returns the current date and time for the file.
- **`currentdateminus`**: Returns a timestamp value of the current date minus the specified days from the current date.
  For example, `{{ currentDateMinus(10) }}` returns a timestamp after deducting 10 days from the current date.
- **`uuid`**: Returns the UUID of the file `{{ uuid() }}`.
- **`arrow`**: Returns a python arrow library.
- **`toJSON`**: Converts a JSON to a string. Useful for storing a JSON in a FortiSOAR textarea field, for example, `Source Data`, so that JSON renders correctly and the content can be presented nicely in the UI.

- **`html2text`**: Converts an HTML string to a text string.
  `{{ html_string | html2text}}`
  For example, `{{'<br>this is html text </br>' | html2text}}`.
  Output will be - this is html text.
- **`json2html`**: Converts JSON data into HTML. The FortiSOAR template is used for HTML and styling of the output.
  `{{ jsondata | json2html(row_fields)}}`
  `row_fields= ['pid', 'sid']`.
  You can optionally specify the `row_fields` attribute. If you do not specify the row_fields, by default, this filter takes all keys as row fields.
  An example without row fields specified: `{{ [{"pid": 123, "sid": "123", "path": "abc.txt"}] | json2html}}`.
  An example with row fields specified: `{{ [{"pid": 123, "sid": "123", "path": "abc.txt"}] | json2htmll([‘pid’, ‘sid’])}}`.
  The HTML output of the above example will be:
  `<table class="cs-data-table"> <tr> <th>pid</th> <th>sid</th> </tr> <tr> <td>123</td> <td>123</td> </tr> </table><button style="display:none" class="cs-datatable-btn btn-link cs-datatable-showmore-btn" type="button" onClick="event.target.previousElementSibling.className += ' cs-data-table-show-more'; event.target.nextElementSibling.style.display = 'block'; event.target.style.display = 'none';">Show more</button><button class="cs-datatable-btn btn-link cs-datatable-showless-btn" type="button" onClick="event.target.previousElementSibling.previousElementSibling.className = 'cs-data-table'; event.target.previousElementSibling.style.display = 'block'; event.target.style.display = 'none';">Show less</button>`
- **`count_occurrence`**: Retrieves the number of times each element appears in the list.
  For example, `{{ [‘apple','red','apple','red','red','pear’] | count_occurrence }}`
  The output of this example is: `{'red': 3, 'apple': 2, 'pear': 1}`
- **`urlencode`**: Encodes the given URL.
  For example, `{{"/api/3/alerts/?name=test" | urlencode}}`
  The output of this example is: `%2Fapi%2F3%2Falerts%2F%3Fname%3Dtest`
- **`urldecode`**: Decodes the given (encoded) URL.
  For example, `{{"%2Fapi%2F3%2Falerts%2F%3Fname%3Dtest" | urldecode}}`
  The output of this example is: `/api/3/alerts/?name=test`
- **`loadRelationships(moduleName, selectFields = [])`**: Used to fetch details of a related (correlation) record. For example, `{{ vars.incidentIRI | loadRelationships('indicators') }}`
  To fetch complete details of the correlation record, use `{{ #recordIRI# | loadRelationships ('#CorrelationModuleFieldName#') }}`
  To fetch specific fields of the correlation record, use `{{ #recordIRI# | loadRelationships ('#CorrelationModuleFieldName#',['#field1#','#field2#']) }}`
- **`picklist`**: Loads the specified picklist item object. For example, `{{"PicklistName" | picklist ("ItemValue") }}`
  The output of this example is an object including the `@id`, `color`, `itemValue`, `listName`, and `orderIndex` of the picklist item. You can extract just a particular key from the object by specifying a second argument to the filter:
  `{{"PicklistName" | picklist("ItemValue", "@id") }}`. This will generate `/api/3/picklists/<uuid>`

You can add custom functions or function references to your custom connector, edit an existing connector to add custom functions, or expose an action of a connector as a function. For more information, see the *Building your own connector* chapter in the "Connectors Guide."

Maintaining Playbooks

Working with Content

The Content Hub is a centralized repository for connectors, widgets, solution packs, and other reference materials. Accessible within the platform and publicly at: https://fortisoar.contenthub.fortinet.com/, it enables users to browse, install, and manage components easily. The Hub syncs hourly with the main repository to ensure listings stay up to date.

To access the Content Hub, a user must be assigned a role that has a minimum of `Read` permission on the `Content Hub` and `Applications` modules.

For details, see the Access and Install Content from the Content Hub chapter in the "User Guide."

Best Practices for Working with Content

- **Test in a Staging environment** : Use the FortiSOAR Continuous Delivery solution pack to set up source control and manage the content development lifecycle (CI/CD) across Development, Staging, and Production environments.
  - Always import and validate new content in a staging environment before deploying it to production.
  - Confirm content compatibility with your current FortiSOAR version.
- **Keep Content up-to-date** : Regularly check for updates to connectors, widgets, and solution packs. Use the **Update Available** filter under the **Manage** tab in the Content Hub to view available updates.

Designing for Flexibility and Scalability

When creating playbooks in FortiSOAR, design them to be flexible and scalable. This is essential when managing complex environments, multi-tenancy, and large datasets.

Usage Tip:

- Use conditional steps in playbooks to handle various scenarios based on data or inputs from the case. This allows the playbook to adapt dynamically to different contexts, which is especially important in multi-tenant environments. For details, see the Playbooks chapter.
  Example:
  Use conditions like `If...Else` or `Switch` blocks based on tenant identifiers to adjust actions according to the organization or customer.
- Break down playbooks into modular components instead of creating a single, large workflow. Modular design simplifies troubleshooting, updates, and scaling.

Leveraging Templates and Reusable Components

Use *templates* and reusable logic to reduce duplication and improve manageability.

Usage Tip:

- Build reference playbooks and reusable actions for use across multiple workflows.
- Create reference-playbooks for specific, repeatable tasks like threat intelligence lookups, IP blocking, or user notifications.
  Example:
  Develop a reference-playbook that enriches alerts using external threat intelligence, and reuse it across various workflows.

Documenting Playbooks and their Usage

Clear documentation ensures playbooks are easier to understand, maintain, and transfer between teams.

Usage Tip:

- Include meaningful comments within playbooks to explain the purpose of each step and key decision points.
  Example:
  When adding a new step, include a comment explaining the logic. This helps ensure that others—and your future self—can understand the reasoning behind specific actions.
- Maintain an external playbook documentation repository to track changes, versions, and known issues associated with each playbook.

Testing and Validating Playbooks

Thorough testing is critical before production deployment. Use FortiSOAR's debugging and simulation tools to verify logic and behavior.

Usage Tip:

- Use test cases or mock data when to validate steps. Simulate various event types (e.g., phishing, malware) to ensure the playbook handles all scenarios.
- Test playbook performance, particularly those interacting with external systems. For playbooks with API calls, ensure they remain responsive under load.

Optimizing Playbook Performance

Well-optimized playbooks ensure faster execution and lower resource usage.

Usage Tip:

- Minimize the number of steps where possible. Group related actions to reduce system overhead.
- Avoid unnecessary API calls; trigger them only when required.
- Optimize queries across integrated systems (e.g., network devices, threat intelligence platforms) to handle large datasets efficiently.
- Use '*Parallel Execution*' for independent steps to execute steps simultaneously and reduce processing time.

Handling Playbook Errors and Failures

Robust error handling prevents workflow interruptions and ensures greater operational resilience.

Usage Tip:

- FortiSOAR allows logging of errors and supports meaningful error messages to help teams understand the root cause.
- Always include error-handling steps in your playbooks, especially when interacting with external systems or APIs.
  Example:
  If a playbook includes an API call to a third-party system, add an error-handling step that logs the failure and retries the request. If the retry limit is exceeded, trigger a notification to administrators.
- Include fallback actions when a step fails—for instance, if sending an email alert fails, the playbook can trigger a secondary notification via another channel.

Security Considerations in Playbook Design

Playbooks often handle sensitive data and systems, so they must be designed with security in mind. It is essential to isolate potentially risky steps and follow secure development practices.

Usage Tip:

- Use FortiSOAR agents for isolation when running custom scripts or interacting with external systems. This ensures sensitive operations are executed in a secure environment. For details, see the FortiSOAR Agent Setup and Configuration chapter in the "Administration Guide."
- Avoid hardcoding credentials or API keys directly into playbooks. Instead, use secret management and environment variables to securely store and retrieve sensitive information.
  Example:
  When making API calls that require authentication, use FortiSOAR's Password Vault Manager to securely store and access API keys instead of embedding them directly into the playbook.

Supporting Multi-Tenancy

Multi-tenancy requires each playbook to adapt based on the tenant or organization it serves. Proper isolation between tenants is essential for maintaining security and compliance.

Usage Tip:

- Use dynamic variables to extract tenant-specific data and drive playbook logic accordingly.
- Isolate actions per tenant to avoid data leakage or unauthorized access.
- Use aliasing to map tenant-specific configurations and actions.
  Example:
  A playbook that detects a security incident for one tenant should process data and take actions only for that tenant. Maintain tenant context throughout the playbook to ensure actions remain tenant-specific.

Monitoring and Continuous Improvement

Ongoing review and monitoring ensure playbooks stay efficient and effective over time.

Usage Tip:

- Configure alerting and logging mechanisms to track playbook execution and flag failures or performance issues to administrators.
- Use FortiSOAR's Health Assessment tools to track the performance and health of playbooks. For details, see topics such as Playbook Execution and Debugging and Monitoring & Optimization chapter in the "Best Practices Guide."
- Periodically assess and refine playbooks using collected performance data to ensure consistent and reliable operation.

Optimizing and Troubleshooting

This chapter provides you information on how to tune various keys and troubleshoot playbook errors.

Optimizing Playbooks

Playbook steps that were looped (using the Loop option in the playbook step) can be run either in a sequentially or in parallel. For more information, see the Loop topic.

You can tune the thread pool size and other settings for parallel execution using the settings that are mentioned in the following table. You can modify default values by editing the `config.ini` file (`sudo vi /opt/cyops-workflow/sealab/sealab/config.ini`) and then restarting the services using the `sudo systemctl restart uwsgi celeryd fsr-workflow celerybeatd` command.

Key name and location	Description	Default value
`THREAD_POOL_WORKER` `/opt/cyops-workflow/sealab/sealab/config.ini`	The thread pool size is used for parallel execution. The `THREAD_POOL_WORKER` variable is used to optimize the parallel execution and enhance performance. You can reduce the default value of the thread pool size from the default value if: 1. The number of cores on your FortiSOAR instance are lesser than the default recommended. 2. The task to be executed in the loop step is synchronous in nature and thread context switching would be an overhead.	8
`SYNC_DELAY_LIMIT` `/opt/cyops-workflow/sealab/sealab/config.ini`	If the delay specified in the playbook step is higher than this threshold, then the loop step will be decoupled from the main playbook and run asynchronously. For example if you set the `SYNC_DELAY_LIMIT` to **60**, it means that a 60 seconds check is added, and after 60 seconds the playbooks should run in parallel. This works in parallel with your playbook soft limit time, `CELERYD_TASK_SOFT_TIME_LIMIT` parameter. The time set in the `CELERYD_TASK_SOFT_TIME_LIMIT` parameter must be greater than the time set in the `SYNC_DELAY_LIMIT` parameter.	60

`CELERYD_TASK_SOFT_TIME_LIMIT /opt/cyops-workflow/sealab/sealab/config.ini`	To change the soft time limit for playbooks. The soft time limit value is set in seconds. **NOTE**: The soft time limit allows the task to catch exceptions and clean up before being terminated; the hard timeout is not catch-able and force terminates the task. If the soft time limit is exceeded and cleanup is stuck then the hard time limit (expected to be higher than the soft time limit) will force terminate the playbook task as a safety net and to avoid forever running of task.	1800
`CELERYD_TASK_TIME_LIMIT /opt/cyops-workflow/sealab/sealab/config.ini`	To change the hard time limit for playbooks. The time limit value is set in seconds. **Note**: This value should always be higher than the `SOFT_TIME_LIMIT`. For more details, see the Celery 4.3.0 documentation >> User guide: task_soft_limit section.	2400
`SQUEEZE_SOFT_TASK_TIMEOUT /opt/cyops-workflow/sealab/sealab/config.ini`	To change the soft time limit for 'pg_squeeze' tasks. The soft time limit value is set in seconds. The soft time limit allows the task to catch exceptions and clean up before being terminated; the hard timeout is not catch-able and force terminates the task. If the soft time limit is exceeded and cleanup is stuck then the hard time limit (expected to be higher than the soft time limit) will force terminate the pg_squeeze task as a safety net and to avoid forever running of task.	3600
`SQUEEZE_TASK_TIMEOUT /opt/cyops-workflow/sealab/sealab/config.ini`	To change the hard time limit for 'pg_squeeze' tasks. When the task exceeds this duration, the system will throw an exception. The time limit value is set in seconds. **Note**: This value should always be higher than the `SQUEEZE_SOFT_TASK_TIMEOUT` and is used to override the `CELERYD_TASK_TIME_LIMIT` for the 'pg_squeeze' task.	3800
`CELERYD_OPTS /etc/celery/celeryd.conf`	To optimize the parallel running of threads in celery so that your overall playbook execution time is reduced. By default, the workflow engine spawns a separate process running a workflow. If the tasks in the workflow are asynchronous and short lived, the thread-based workers can be enabled. For more details, see the Celery 4.3.0 documentation > User guide > Concurrency >> Concurrency with Event section.	`CELERYD_OPTS="-P=eventlet -c=30"`

These optimizations also help in scaling your playbooks by resolving bottleneck that slow down playbook execution and resolving internal timeout issues.

FortiSOAR supports parallel branch execution of playbooks. Parallel branch execution optimizes playbook execution by having the ability to execute two or more independent paths parallelly.

You can enable or disable parallel execution by changing the value (true/false) of the `PARALLEL_PATH` variable in the `[Application]` section in the `config.ini` file (`sudo vi /opt/cyops-workflow/sealab/sealab/config.ini`). By default, a fresh install of version 5.1.0 will have the `PARALLEL_PATH` variable set as `true`.

Optimized Workflow Runtime for Memory and CPU consumption

In release 7.2.0, FortiSOAR has optimized and greatly improved the playbook execution process. Tremendous improvements are observed for playbook execution times as well as OS resource consumption during playbook execution. Some notable improvements are:

- Reduction of Virtual Memory consumption during playbook execution by 50%.
- Reduction of CPU usage during playbook execution by 50%.
- Optimized execution of Jinja expressions.
- Database space utilization is optimized by reducing storing duplicate values used by the playbook framework.

Some test cases were run to compare playbook execution times across 7.0.2 and 7.2.0:

Comparison of Playbook Execution Times between 7.0.2 and 7.2.0			
Workflow Type (creates 1240 records)	**Time taken for execution in FortiSOAR 7.0.2**	**Time taken for execution in FortiSOAR 7.2.0**	**Improvement***
Playbook with create record using the 'For Loop'	1 minute 51 seconds 751 milliseconds	57 seconds 190 milliseconds	1.9x Faster
Playbook with a Create Record step using the 'For Loop with the *Batch* option'	1 minute 44 seconds 715 milliseconds	56 seconds 710 milliseconds	1.8x Faster
Playbook with a Create Record step using the 'For Loop with the *Batch* option and the *Parallel* options'	1 minute 21 seconds 132 millisecond	24 seconds 967 milliseconds	3.2x Faster
Playbook with a Create Record step using the 'For Loop with the *Batch* option and the *Sequential* options'	3 minutes 31 seconds 70 milliseconds	1 minute 47 seconds 143 milliseconds	1.9x Faster
Playbook with a Reference step using the 'For Loop with the *Parallel* option'	7 minutes 40 seconds 386 milliseconds	3 minutes 16 seconds 756 milliseconds	2.3x Faster
Playbook with a Reference step using the 'For	1 minute 6 seconds 567	8 seconds 460 milliseconds	7x Faster

Loop with the *Parallel* option that runs asynchronously' **Note**: In the case of parallel execution, even if the playbook is completed, record creation continues in the background.	milliseconds		
Playbook with a Reference step using the 'For Loop with the *Sequential* option'	10 minutes 43 seconds 133 milliseconds	6 minutes 1 second 350 milliseconds	1.7x Faster
Playbook with a Reference step using the 'For Loop with the *Sequential* option that runs asynchronously'	2 minutes 28 seconds 10 milliseconds	15 seconds 477 millisecond	9.24x Faster
Playbook using the *json2html* filter in the Update Record step	5 minutes 7 seconds 664 milliseconds	1 minute 44 seconds 236 milliseconds	2.9x Faster
***** - *Calculated using the formula: Previous execution time / Current execution time*			

Similarly, the data ingestion process was also tested to compare its execution times across 7.0.2 and 7.2.0. The following steps were performed in the tests:

1. Ingest 17000 records by reading a CSV file.
2. Create corresponding tickets in a ticketing platform using an integration (connector).
3. Playbook loops over a Reference step with each iteration consisting of 5000 records.

Comparison of Playbook Execution Times between 7.0.2 and 7.2.0		
Time taken for execution in FortiSOAR 7.0.2	**Time taken for execution in FortiSOAR 7.2.0**	**Improvement***
36 min 45 seconds 134 milliseconds	14 minutes 40 seconds 812 milliseconds	2.5x Faster
***** - *Calculated using the formula: Previous execution time / Current execution*		

Managing the Day Filter Limit for Global Executed Playbook Logs

Starting with release 7.6.5, the From Date field in the global Executed Playbook Log filter (located in the upper-right corner of the screen) defaults to showing playbook logs from the past 7 days. You can disable this setting to view all playbook execution logs or modify the default number of days displayed (for both historical and recent logs). To enable or disable this setting, or to change the default day limit, follow these steps:

1. Edit the `config.json` file
   `sudo vi /opt/cyops-ui/vendor/config.json`
2. Locate the following section:

```
{
    "...": {},
```

```
    "historicalLogs": {
        "enableLimit": true,
        "lastXDays": 7
    }
}
```

3. Modify the settings as needed:
   - To view all playbook executions, set:
     `"enableLimit": false`
   - To change the default number of days displayed, update the value of:
     `"lastXDays": <desired_number>`
4. Save the `config.json` file.
5. Reload the browser after the changes are saved.

Code Snippet Connector: Usage Tips and Examples

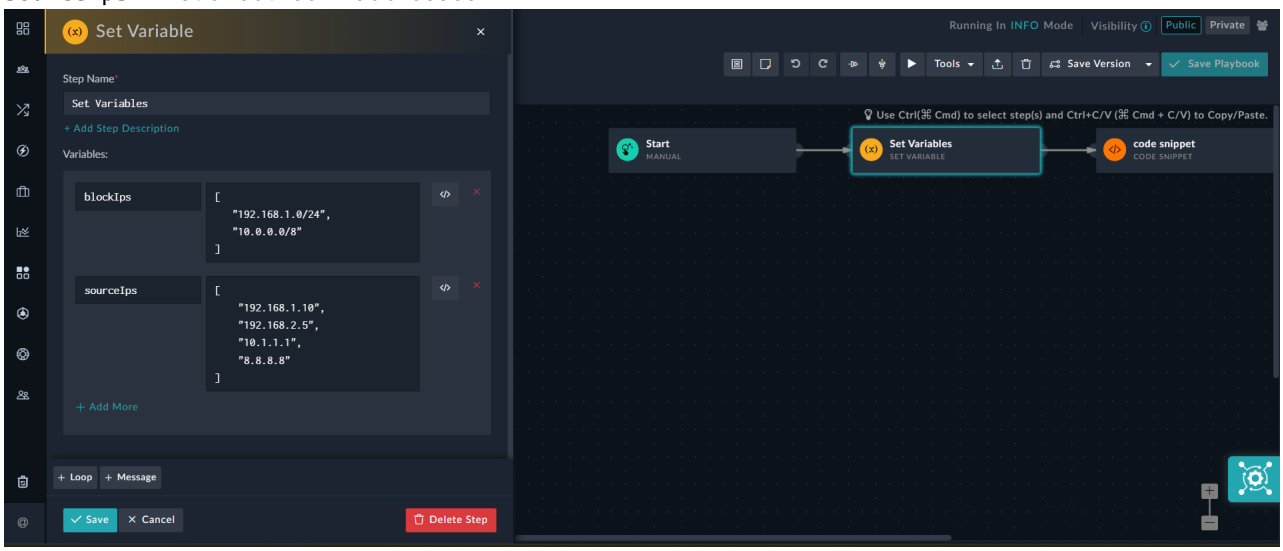
This section provides practical tips and examples for using the Code Snippet connector in playbooks, including how to reference playbook variables and display output for use in subsequent steps:

- **Accessing variables defined in a playbook**: To access variables defined using the Set Variable step use:
  `variable = {{ vars.<variable_name> }}`
- **Displaying output in playbook history**: To make values visible in the Playbook History Output or available for use in subsequent steps, you must explicitly print them in your code.
  `variable = {{ vars.test }}`
  `print(variable)`

Example: Checking whether an IP is contained in a CIDR List

This example demonstrates how to check whether source IP addresses are contained within a list of blocked CIDR ranges using a Python function.

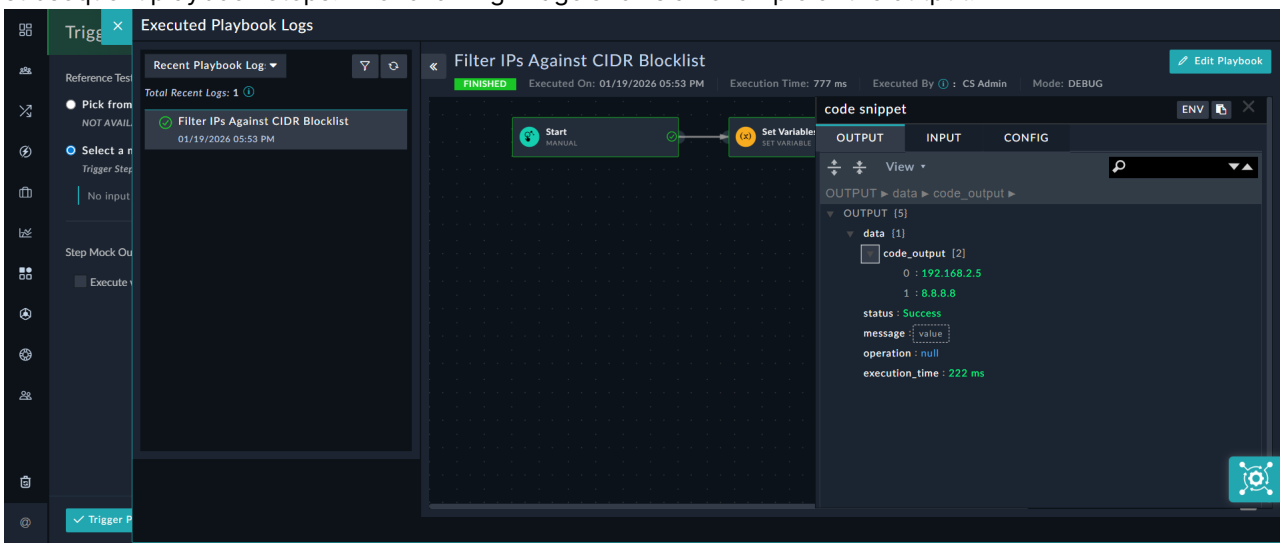
1. Use the **Set Variable** step to define the following variables (type: list):
   `blockIps` — list of blocked CIDR ranges
   `sourceIps` — list of source IP addresses



2. Add the **Code Snippet Connector** step to the playbook.
3. Select the **Execute Python Code** action.
4. In the **Python Function** field, enter the Python code enter the following Python code to filter source IP addresses that are not included in a list of blocked CIDR ranges:

```
from ipaddress import ip_network, ip_address

block_cidrs = {{vars.blockIps}}
source_ips  = {{vars.sourceIps}}

block_networks = [ip_network(cidr) for cidr in block_cidrs]

filtered_ips = []
for ip_str in source_ips:
    ip_obj = ip_address(ip_str)
    blocked = False
    for net in block_networks:
        if ip_obj in net:
            blocked = True
            break
    if not blocked:
        filtered_ips.append(ip_str)

print(filtered_ips)
```

   - The variables `blockIps` and `sourceIps`, which are defined in a previous *Set Variable* step, are accessed using:
     `block_cidrs = {{ vars.blockIps }}`
     `source_ips = {{ vars.sourceIps }}`
   - The list `filtered_ips` contains only the IP addresses that are not part of the blocked CIDR ranges.
5. Save the step and the playbook.
   The filtered IP addresses are displayed in the Playbook History Output, and they can also be referenced by subsequent playbook steps. The following image shows an example of the output:



Common Errors and Fixes

High CPU/Memory Usage

The High CPU usage and/or memory usage by 'python' process issue could be caused by to some errors in workflows, like updating the same record in a post-update playbook; or a flood of events from a SIEM that causes massive workflow queuing on the FortiSOAR side.

Resolution

If the large queue build-up is due to a playbook error, or similar events' flood that need not be created as alerts or incidents into FortiSOAR, you can purge the workflow queue.

Run the following command on your FortiSOAR system to see the number of workflows queued:
`sudo rabbitmqctl list_queues -p fsr-cluster`

To purge the queue run the following command:

```
sudo systemctl stop celeryd
sudo rabbitmqctl purge_queue celery -p fsr-cluster
sudo systemctl start celeryd
```

Purging is an irreversible action, which should be exercised with caution in a production environment.

'Access Denied' Errors

After you have upgraded your system to release 7.2.0 or later from a release prior to 7.2.0, and you execute playbooks that download files, you will observe that such playbooks might fail with the 'Access Denied' error. This is because in 7.2.0 `fsr-integrations` is used as the workflow service user to create files during playbook execution. As the releases prior to 7.2.0 used `nginx` as the user, files created by `ngnix` will not be deleted by new user (`fsr-integrations`).

Resolution

If there is are playbooks that create files using the Code Snippet step or any such step, where the playbook is not directly creating files, then you have to ensure that those files are deleted as part of playbook flow. If this is not done, then old files that were created before the upgrade are not deleted and such playbooks fail to delete the older files because of the difference in user permissions.

Jinja Parsing Error: Integer Too Long

Jinja cannot handle integers that have more than 16 characters issue is caused due to the JavaScript max integer safe size, i.e., 2^{53} - 1, i.e., 9007199254740991.

Resolution

You do not need to do anything since the difference is seen only while rendering the data in the browser, but not in the Database persistence.

Playbook failure due to missing Jinja `json_query` filter

This issue occurs when a playbook fails with the following error:
`CS-WF-5: Inappropriate use of Jinja key: 'no filter named 'json_query''`

The error indicates that the required Jinja `json_query` filter is unavailable, typically due to an issue with the Ansible installation.

Resolution

1. Ensure that you have access to the FortiSOAR package repository: https://repo.fortisoar.fortinet.com
2. SSH into the FortiSOAR system and log in as the 'root' user.
3. Uninstall the existing Ansible packages:
   ```
   sudo -u nginx /opt/cyops-workflow/.env/bin/pip uninstall ansible
   sudo -u nginx /opt/cyops-workflow/.env/bin/pip uninstall ansible-base
   ```
4. Install Ansible from the FortiSOAR repository:
   ```
   sudo -u nginx /opt/cyops-workflow/.env/bin/pip install ansible --index-url
   https://repo.fortisoar.fortinet.com/prod/connectors/deps/simple/
   ```
5. Restart the celeryd service:
   ```
   sudo systemctl restart celeryd
   ```
6. Re-run the affected playbook.

Picklist Item Errors

The `Picklist item:<name of picklist filter>` error is caused if the picklist filter in the playbook has a value that does not exactly match with the existing actual value (case sensitive), then such playbooks will fail.

Resolution

Ensure that you have specified the value of the playbook filters exactly as its existing value. From release 7.2.0 onwards, an exact case match is required for playbook filters to work.
For example, if you have specified `{"Severity" | "MEDIUM", "@id")}}` in a playbook, then this playbook will fail if the actual value of `Severity` is `Medium`.

No Permission Errors

Resolution

When the `Playbook` does not have appropriate permissions, then playbooks fail. `Playbook` is the default appliance in FortiSOAR that gets included in a new team.

If you cannot access records, such as alerts, then you must ensure that you are part of the team or part of a sibling or a child team that can access the records, and you must have appropriate permissions on that module whose records you require to access or update. Only users with `CRUD` access to the `Appliances` module can

update the `Playbook` assignment. For more information on teams and roles, see the Security and Access Management chapter in the "Administration Guide."

Ingestion Failures

Playbooks could fail due to many reasons after ingestion is triggered, for example, if a required field is `null` in the target module record, or there are problems with the Playbook Appliance keys.

Resolution

Investigate the reason for failure using the **Playbook Execution History** tab (earlier known as **Running Playbooks**) in the `Playbook Administration` page. Review the step in which the failure is being generated and the result of the step, which should contain an appropriate error message with details. Once you have identified the error, and if you cannot troubleshoot the error, contact the FortiSOAR support team for further assistance using the Fortinet Customer Service & Support web portal at https://support.fortinet.com/.

Issues with Purging Executed Playbook Logs

If you are facing issues while purging of executed playbook logs such as, the purge activity is taking a long time or the purging activity seems to be halted, then you could check if the `Soft time limit (600s) exceeded for workflow.task.clean_workflow_task[<taskid>]` error is present in the `/var/log/cyops/cyops-workflow/celeryd.log` file. The `Soft time limit` error might occur if the amount of playbook logs to be purged is very large.

Resolution

Increase the value set for the `LOG_PURGE_CHUNK_SIZE` parameter, which is present in the `[application]` section, of the `/opt/cyops-workflow/sealab/sealab/config.ini` file.

By default, the `LOG_PURGE_CHUNK_SIZE` parameter is set to `1000`.

Too Many Connections Error

Playbooks can fail with the `Too many connections to database` error when using the "parallel" option for a loop step in Playbooks

Resolution

To resolve this issue, reduce the number of parallel threads. To reduce the number of parallel threads, you have to change the value of the `THREAD_POOL_WORKER` variable. The `THREAD_POOL_WORKER` variable is present in the `/opt/cyops-workflow/sealab/sealab/config.ini` file, and by default the value of this variable is set to **8**.

"Picklist item not found" Error

Your playbooks fails with an error such as:
`Picklist Item: /api/3/picklists/a1bac09b-1441-45aa-ad1b-c88744e48e72 not found URL: https://localhost/api/3/alerts`

Resolution

This issue is caused when you have specified an incorrect value for a picklist item. To resolve this issue, check and correct the values of the picklist item in the current step or the previous step of the playbook.

Manual Input Email - Incorrect Link Error

FortiSOAR uses the dynamic variable '`Server_fqhn`' to construct the server url for all links in emails. The default value of this variable is the hostname of the machine set at the time the VM Configuration wizard is run. If your deployment is in a NATed or cloud environment, the public address of the instance will not match the hostname, and the manual input email might contain a wrong server address.

Resolution

To rectify this issue, you must update the '`Server_fqhn`' dynamic variable and provide the DNS resolvable FQHN of your FortiSOAR instance.

Incorrect Hostname in System Playbooks

The Incorrect Hostname being displayed in links contained in emails sent by System Playbooks occurs when you are using a system playbook to send an email. For example, when an alert is escalated to an incident, and an Incident Lead is assigned, then the system playbook sends an email to the Incident Lead specified. The email that is sent to the Incident Lead contains the link to the incident using the default hostname.

Resolution

To ensure that the correct hostname is displayed in the email, you must update the appropriate hostname as per your FortiSOAR instance, in the Playbook Designer as follows:

1. Open the Playbook Designer.
2. Click **Tools** > **Global Variables** to display a list of global variables.
3. Click the `Edit` icon in the `Server_fqhn` global variables, and in the `Field Value` field add the appropriate hostname value.
4. Click **Submit**.
   The system playbook will now send emails containing the updated hostname link.

In the system playbook (or any playbook) that is sending an email, ensure that you have used the `Server_fqhn` global variable in the `Send Email` step.

Database Connection Issues

Access to PostgreSQL without a password has been restricted. This causes a failure in the case of any playbook step that uses the 'Database' connector to access a database directly, without a valid password.

Resolution

Update the playbook step to use a read-only database user to access the database.

Frequently Asked Questions

Q: Is it possible to use the result of a previous loop item in the next iteration for sequential loop execution?

A: When running a step in a loop, you might need to access result of the previous item execution in the next iteration. For example, when creating playbook steps using LLMs, you might need to use the previous set of steps generated as input for generating the next step. To do this, add `save_iteration_result` as an environment variable and set its value to 'true'. The result of the previous iteration can then be accessed using `iteration_result_<step_name_with_underscore>`. This method is applicable for both sequential 'for each' loops and 'do until' loops.

Q: How to adjust the limit the number for exporting records in the CSV format from the modules' grid view?

A: The maximum number of records that can be exported in the CSV format from the modules' grid view is currently set to '2000'. To change this value, go to the playbook designer and click **Tools** > **Global Variables**. Search for the `csv_export_max_records` variable, and update its value to specify the maximum number of records that can be exported in CSV format from the modules' grid view.

Q: Is there a way to force variables set in a reference playbook to carry over into the parent playbook? I rather not put a group of steps I need in the parent if I can avoid it, as I am using the child playbook as an action itself, so would it duplicate the functions?

A: In general, variables set in child playbooks do not carry over to the parent playbook. The one exception is that the **Reference a Playbook** step will return (in `vars.steps.<step_name>.keyname`) the return value of the last executed step in the child playbook. For instance, if the last step in the child playbook is **Find Record**, then the **Reference a Playbook** step will populate `vars.steps.<step_name>.data` with the records that have been found using the **Find Record** step.

If u want to define the playbooks result as a combination of results of previous steps or sub-steps, you can use the `Set Variable` step at the end of the playbook and define variables that would contain data that you require to be returned.

Q: How do I convert Epoch time returned by a SIEM to a datetime format?

A: If you have a playbook, which has a connector step that connects to a SIEM, such as ArcSight or QRadar, and the SIEM returns the result in Epoch time (milliseconds), then you can convert Epoch time to the datetime format using the following command:

```
# arrow.get(1531937147932/1000).to('Required Timezone').strftime("%Y-%m-%d %H:%M:%S %Z%z")
```

or

```
# arrow.get(1531937147932/1000).to('Required').format('YYYY-MM-DD HH:mm:ss ZZ')
```

For example,

```
# arrow.get(1531937147932/1000).to('EST').format('YYYY-MM-DD HH:mm:ss ZZ')
```

Will return the following output:

```
2018-07-18 14:05:47 EDT-0400
```

For more examples on dates and times used in Python, see http://arrow.readthedocs.io/en/latest/.

Q: How do I change the timeout limit for playbooks?

A: To change the time limit or soft time limit for playbooks you must edit the `CELERYD_TASK_TIME_LIMIT` and `CELERYD_TASK_SOFT_TIME_LIMIT` parameters in the `config.ini` file (`sudo vi /opt/cyops-workflow/sealab/sealab/config.ini`). By default, these parameters are set in seconds, as follows:

```
CELERYD_TASK_TIME_LIMIT = 2400
CELERYD_TASK_SOFT_TIME_LIMIT = 1800
```

Once you have made the change you must restart all the FortiSOAR services using the following command:
`# sudo csadm services --restart`

Q: Filters in running playbooks are not working after post-upgrade in case of pre-upgrade log records

A: You can apply filters on running playbooks using the **Executed Playbook Logs**. These filters will apply to log records that are created post-upgrade and will not apply to log records that were created pre-upgrade.

For log records that were created before the upgrade, use the playbook detail API:
`GET: https://<FortiSOAR_HOSTNAME/IP>/api/wf/api/workflows/<playbook id>/?format=json`

To get the playbook id, use the playbook list API:
`GET: https://<FortiSOAR_HOSTNAME/IP>/api/wf/api/workflows/?depth=2&limit=30&ordering=-modified`

Q: How are dynamic variables used in condition steps?

A: Decision steps use dynamic variables with logical equalities of the form:

```
{{ 8 == 8 }}
```

This statement will return either the string 'True,' or 'False' which will automatically be converted into a real boolean value.

Decision steps advanced interface does not require the use of curly braces like `{{ }}`.

Q: How to retain a variable as a string post auto conversion?

A: The playbook engine's auto conversion behavior tries to determine each variable's type based on its value and assign the appropriate type to the variable each time it is referenced. In the absence of this behavior, all variables would be interpreted as 'strings'. Now, however, if you apply a FortiSOAR Jinja filter to data with the intention of getting a result as a 'string' but, the auto conversion causes the result to be of a different type, you need to enclose the Jinja function in quotes.
For example,
`data = [1,2,3]`
Jinja function `{{vars.data| join(',') }}` returns the same integer list `[1,2,3]`
However, if you want the result as a string, enclose the Jinja function in quotes:
`"{{vars.data| join(',') }}"`
This returns the result as `"1,2,3"`, instead of [1,2,3].

www.fortinet.com

Copyright© 2026 Fortinet, Inc. All rights reserved. Fortinet®, FortiGate®, FortiCare® and FortiGuard®, and certain other marks are registered trademarks of Fortinet, Inc., and other Fortinet names herein may also be registered and/or common law trademarks of Fortinet. All other product or company names may be trademarks of their respective owners. Performance and other metrics contained herein were attained in internal lab tests under ideal conditions, and actual performance and other results may vary. Network variables, different network environments and other conditions may affect performance results. Nothing herein represents any binding commitment by Fortinet, and Fortinet disclaims all warranties, whether express or implied, except to the extent Fortinet enters a binding written contract, signed by Fortinet's Chief Legal Officer, with a purchaser that expressly warrants that the identified product will perform according to certain expressly-identified performance metrics and, in such event, only the specific performance metrics expressly identified in such binding written contract shall be binding on Fortinet. For absolute clarity, any such warranty will be limited to performance in the same ideal conditions as in Fortinet's internal lab tests. Fortinet disclaims in full any covenants, representations, and guarantees pursuant hereto, whether express or implied. Fortinet reserves the right to change, modify, transfer, or otherwise revise this publication without notice, and the most current version of the publication shall be applicable.