FORTINET

Widget Development

FortiSOAR 7.6.5

December, 2025
FortiSOAR 7.6.5 Widget Development
00-400-000000-20221031

TABLE OF CONTENTS

Change Log 5

Overview 6

Request for the creation of a Git repository 6

Development process 6

Prerequisites to widget development 6

Process of widget development 7

Widget dependencies 10

Services 10

$scope 10

$rootScope 11

Config 12

$http 15

$resource 15

$q 16

$uibModalInstance 16

Toaster 17

$timeout 17

$filter 18

_(Underscore Library) 20

$window 20

$state 20

WizardHandler 20

LocalStorageService 21

PromiseQueue 21

Constants 21

appModulesService 23

Common Utils 23

connectorService 24

currentPermissionsService 25

Entity 25

exportService 26

Field 26

FormEntityService 27

licenseService 27

modelMetadatasService 27

Modules 28

picklistsService 28

playbookService 29

PromiseQueue 29

Query 30

queryCollectionService 30

PagedCollection 31

settingsService 31

tokenService 32

usersService 32

ViewTemplateService 33
widgetTemplateService 33
Directives 34
Field Directive 34
Conditional Filters 37
data-cs-messages 39
Grid Directive 40
Chart Directive 42
Focus Directive 44
Card Directive 45
Pagination Directive 45
Datetime Grid Directive 46
Tags Directive 46
Markdown Editor Directive 47
Spinner Directive 48
Unique Directive 48
Filters 49
Default Angular Filters 49
Decode URI Filter 49
Get Module Type Filter 50
Valid IRI Filter 50
Truncate Text Filter 50
Angular JS events and the FortiSOAR Web Socket service 50
Angular JS Events 50
FortiSOAR Web Socket Service 51
More information 52
Appendix A: Tutorials 52
Sample widget to group records in a module by severity 52
Creating a custom non-modal widget with an interactive background 59
Creating a widget with multilingual support 62
Appendix B: Frequently asked questions (FAQs) 66
Can an external css/js file plus CDN assets be added while creating a widgets? 66
How do I make a widget compatible with a supported theme? 68
How do I display a widget on specific pages within FortiSOAR? 68
How can I access the module record context for a widget written for the View Panel? 69
What is the process to submit a widget to Content Hub? 69
Can I write a service specific to a widget? 69
How to find out the current context of the widget? 70

Change Log

Date	Change Description
2025-12-17	Initial release of 7.6.5

Overview

Widgets are simple, easy-to-use software applications designed to perform specific tasks. Serving as independent applications, widgets can easily be embedded into websites and examples include event countdowns, visitor counters, clocks, etc.

FortiSOAR offers many out-of-the-box (OOB) widgets, but customizations and new use cases can be needed to meet user expectations. This document outlines the widget development process, from creating a repository to submitting the widget on the Content Hub.

Request for the creation of a Git repository

A Git repository is required to be created for the widget. To create a repository, users must send an email to <DL Email Address> with the following format:

Widget Name	Git Repository Name	Branch Name*	PMDB Link
*** Branch Name**: For the first release of the widget, use release/1.0.0 as the branch name. Subsequent updates require discussion with the PM/Stakeholders to determine the new branch name, based on the quantum of changes made in the updated version. For example, release/2.0.0 for a major release or release/1.0.1 for a minor release.			

Example request for creating a git repository for an 'Incident Timeline' widget:

Widget Name	Git Repository Name	Branch Name*	PMDB Link
Incident Timeline	widget-incident-timeline	release/1.0.0	<PMDB Link>

Development process

This section provides information about the process to be followed for widget development, including the prerequisites and process of widget development. Additionally, it contains links to additional information that users can use to develop their custom widgets.

Prerequisites to widget development

Before developing custom widgets, it's important to have some understanding of the following:

- AngularJS: https://docs.angularjs.org/guide
- HTML: https://html.spec.whatwg.org/

- CSS: https://www.tutorialspoint.com/css
- JavaScript: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

Basic knowledge of Bootstrap: https://getbootstrap.com/ is also beneficial.

Process of widget development

Widgets can be created using the 'Widget Building Wizard' in Content Hub, as detailed in the Create and Use Widgets topic in the "User Guide" in the FortiSOAR Product Documentation. Values entered during initial configuration are used to generate the `info.json` and other default files such as:

- `view.html`
- `edit.html`
- `view.controller.js`
- `edit.controller.js`
- `widgetUtility.service.js` (from release 7.5.0 onwards)
- `locales` folder, containing files for various locales (languages), such as `en.json`, `ko.json`, etc. (from release 7.5.0 onwards)

The directory structure of a widget is as follows:

widgetname folder
```
--+ info.json
--+ view.html
--+ edit.html
--+ viewController.js
--+ editController.js
--+ images
---+ imagefiles
--+ WidgetAssets
---+ css
----+ <widgetName>.css
---+ html
----+ <sampleTemplate>.html
---+ js
----+ widgetUtility.service.js
----+ <sample>.directive.js
----+ <sample>.service.js
----+ widgetUtility.service.js
---+ locales
----+ en.json
----+ ja.json
----+ ko.json
----+ zh_cn.json
```

For more information on the widget's directory structure, Create and Use Widgets topic in the "User Guide."

Additionally, the following copyright should be included in the widget's html, css, and js files:

```
MIT License
Copyright (c) <copyright year> Fortinet Inc
```

For example,

```
MIT License
Copyright (c) 2024 Fortinet Inc
```

The `info.json` file includes the following fields:

- name: Name used to access the widget.
- title: Title displayed for the widget.
- subtitle: Short description for the widget.
- version: Version of the widget in the x.y.z format.
- published_date: Current timestamp using https://www.unixtimestamp.com/?unixTimestampInput=%7B%7B%7Bs%7D%7D%7D
- metadata:
  - certified: Whether the widget is certified by FortiSOAR.
  - publisher: Organization name to be added as the publisher of the widget.
  - compatibility: Comma-separated list of FortiSOAR versions with which the widget is compatible.
  - pages
  - snapshots: Path of the screenshots that are part of the widget.
  - help_online: Contains the link to the widget documentation in the HTML format.
  - description: Information about the widget, including its feature list.

An example of the `info.json` of the 'User Tile Widget':

An example of the `info.json` of the 'User Tile Widget':

```
{
  "name": "userAssignments",
  "title": "User Tile",
  "subTitle": "Shows required information related to the user.",
  "version": "2.1.1",
  "published_date": "1694006825",
  "metadata": {
    "certified": "Yes",
    "publisher": "Fortinet",
    "compatibility": ["7.0.2", "7.2.0"],
    "pages": ["Dashboard", "Reports", "View Panel", "Listing", "Add Form"],
    "snapshots": [
      "https://repo.fortisoar.fortinet.com/fsr-widgets/userAssignments-2.1.1/images/user-
assignments-settings.png",

      "https://repo.fortisoar.fortinet.com/fsr-widgets/userAssignments-2.1.1/images/user-
assignments-view.png"
    ],
    "help_online": "https://github.com/fortinet-fortisoar/widget-user-
assignments/blob/release/2.1.1/docs/README.md",
    "description": " > User Asssignments Feature List\n\n* Allows ability to select multiple
modules and apply filters to build context around the user.\n* Shows User Avatar \n* Has support
to filter records in Settings \nExample: User Avatar along with data such as Assigned alerts and
incidents."
  }
}
```

After creating the basic structure of a widget, you can use various available widget dependencies, such as Services on page 10, Directives on page 34, and Filters on page 49, to edit the widget to meet your specific requirements.

Additionally, see Angular JS events and the FortiSOAR Web Socket service on page 50 topic to get information about services for event-based communication between controllers and FortiSOAR Web Socket, and the Appendix B: Frequently asked questions (FAQs) on page 66 section to get answers to common questions.

For a step-by-step guide to developing widgets, including steps on how to localize widgets, etc. see the Appendix A: Tutorials on page 52 chapter.

Once you have created your widget, you can submit the same using the process listed in https://github.com/fortinet-fortisoar/how-tos.

Widget dependencies

This chapter provides information about the various elements that are available to develop custom widgets:

- Services on page 10
- Directives on page 34
- Filters on page 49
- Angular JS events and the FortiSOAR Web Socket service on page 50

Services

This topic provides information about the various services that are available for developing widgets. Additionally, FortiSOAR offers the Services API Documentation as part of the Content Hub, providing comprehensive information on the available FortiSOAR services.

`$scope`

Overview

In AngularJS, `Scope` is the glue between the application controller and the view. During the template linking phase, the directives set up `$watch` expressions on the scope. These `$watch` expressions enable the directives to receive notifications of property changes, allowing the directive to render the updated value in the DOM.

Both controllers and directives have reference to the scope but not to each other. This arrangement isolates the controller from the directive and the DOM. This is crucial because it makes the controllers view-agnostic, significantly enhancing the application's testing capabilities.

Reference: Angular JS/Scopes

Example-

View:

```
<input type="text" ng-model="message">
```

Controller:

```
app.controller('myController', function ($scope)
       {
               $scope.message = "Hello, AngularJS!";
       }
);
```

$rootScope

Overview

In AngularJS, rootScope is the parent scope object for an AngularJS application, and there is always one unique $rootScope for an application. The data and methods of the $rootScope object are available to all the controllers. All scope objects are child objects of the $rootScope object.

Reference: Angular JS/rootScope

Example-

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
</head>
<body ng-controller="Controller1">
    <p>Controller 1: {{ dataFromController1 }} </p>
    <button ng-click="broadcastEvent()">Broadcast Event from Controller 1</button>
    <div ng-controller="Controller2">
        <p>Controller 2: {{ dataFromController2 }} </p>
        <button ng-click="emitEvent()">Emit Event from Controller 2</button>
    </div>
    <script>
        var app = angular.module('myApp', []);

        app.controller('Controller1', function ($scope, $rootScope) {
            $scope.dataFromController1 = "Hello from Controller 1";

            $scope.broadcastEvent = function () {
                $rootScope.$broadcast('customEvent', { message: 'Broadcasted from Controller 1'
});
            };
        });

        app.controller('Controller2', function ($scope, $rootScope) {
            $scope.dataFromController2 = "Hello from Controller 2";

            $scope.emitEvent = function () {
                $rootScope.$emit('customEvent', { message: 'Emitted from Controller 2' });
            };

            // Listen for the broadcasted event
            $scope.$on('customEvent', function (event, data) {
                console.log(data.message);
            });
        });
    </script>
</body>
</html>
```

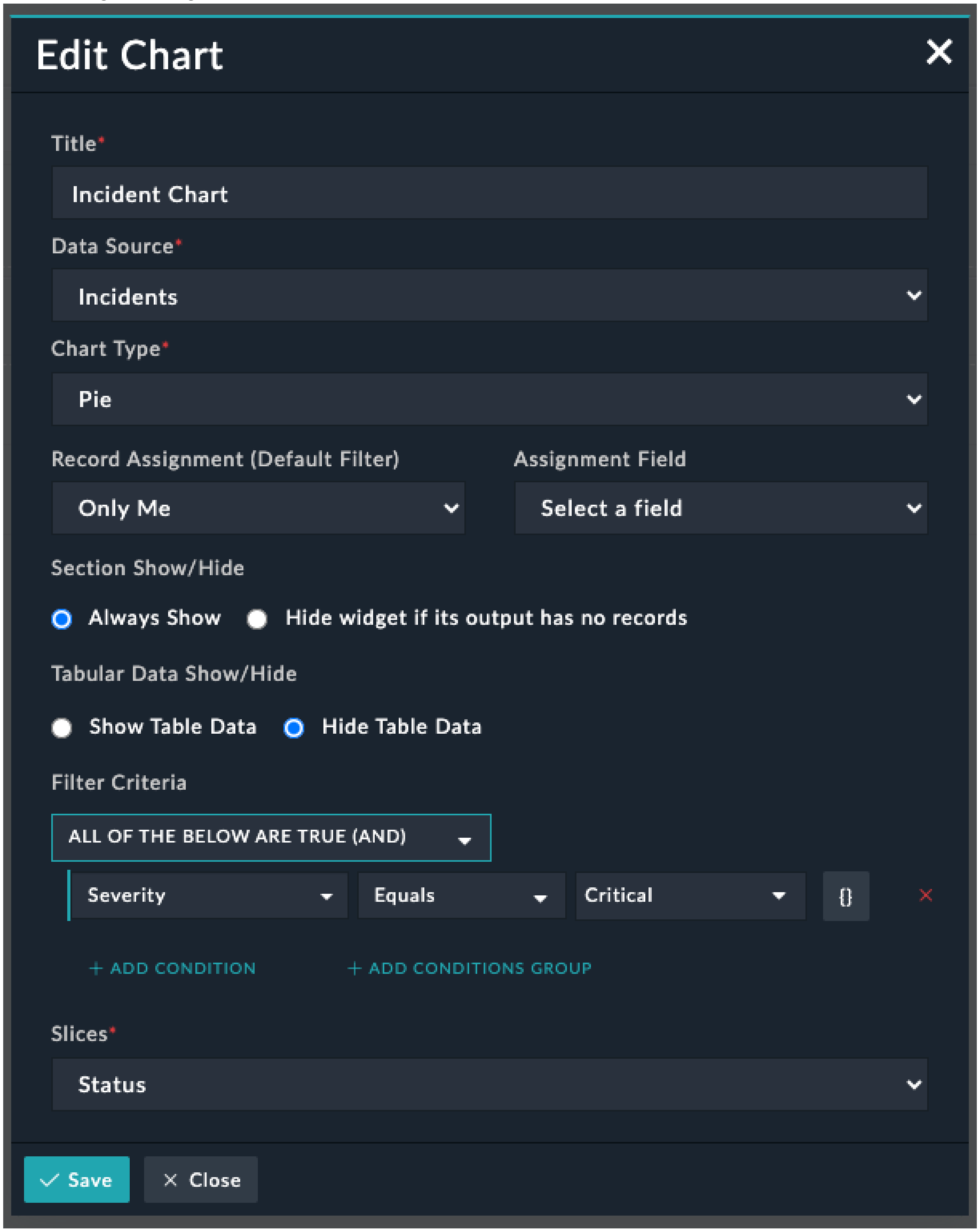
`Config`

Overview

A "config" object in the context of widget configuration serves as a structured set of parameters and settings required to define and render a specific widget within a web application. This object encapsulates details, such as the widget type, input parameters, and optional filters necessary to tailor the widget's appearance and behavior.

Example-

Chart Widget Configuration:



Edit Chart

Title*
Incident Chart

Data Source*
Incidents

Chart Type*
Pie

Record Assignment (Default Filter)
Only Me

Assignment Field
Select a field

Section Show/Hide
[✓ Always Show] Hide widget if its output has no records

Tabular Data Show/Hide
Show Table Data [✓ Hide Table Data]

Filter Criteria
ALL OF THE BELOW ARE TRUE (AND)

Severity	Equals	Critical

Slices*
Status

Config Details:

```
{
      "type": "chart",
      "config": {
        "wid": "283e52eb-8a88-4e89-974b-12ee6a39d3d2",
        "widgetAlwaysDisplay": true,
        "showTabularData": false,
        "query": {
          "sort": [
            {
              "field": "status.orderIndex",
              "direction": "ASC"
            }
          ],
          "limit": 2147483647,
          "logic": "AND",
          "filters": [
            {
              "field": "severity",
              "operator": "eq",
              "value": "/api/3/picklists/7efa2220-39bb-44e4-961f-ac368776e3b0",
              "_value": {
                "display": "Critical",
                "itemValue": "Critical",
                "@id": "/api/3/picklists/7efa2220-39bb-44e4-961f-ac368776e3b0"
              },
              "type": "object"
            }
          ],
          "aggregates": [
            {
              "operator": "countdistinct",
              "field": "*",
              "alias": "total"
            },
            {
              "operator": "groupby",
              "alias": "status",
              "field": "status.itemValue"
            },
            {
              "operator": "groupby",
              "alias": "color",
              "field": "status.color"
            },
            {
              "operator": "groupby",
              "alias": "orderIndex",
              "field": "status.orderIndex"
            }
          ]
        },
        "mapping": {
            "fieldName": "status"
          },
          "assignedToSetting": "onlyMe",
          "aggregate": true,
          "title": "Incident Chart",
          "resource": "incidents",
          "chart": "pie"
        }
}
```

$http

Overview

In AngularJS, the `$http` service is a core component that facilitates communication with remote HTTP servers. It provides methods for making HTTP requests and handling the responses.

Reference: Angular JS/services/$http

Example-

```
$http.get(url).then(function(response) {
       deferred.resolve(response.data);
}, function() {
       deferred.reject('could not load template');
       }
);
```

$resource

Overview

In AngularJS, the `$resource` service is an extension of the core `$http` service, which makes working with RESTful APIs simpler. It is designed to provide a higher-level, resource-based abstraction, making it easier to interact with RESTful web services.

- Similar to `$http`, `$resource` is injected into controllers, services, or other AngularJS components where interaction with RESTful APIs is needed.
- The `$resource` service is used to create a resource object by providing the URL template and optional default parameters.

Reference: Angular JS/services/$resource

Example-

`$resource` simplifies API interactions by providing predefined methods and actions corresponding to common HTTP methods, GET, POST, PUT, and DELETE:

```
app.controller('MyController', ['$resource', function($resource) {
var MyResource = $resource('/api/data/:id', { id: '@id' });
var myResourceInstance = new MyResource();
```

```
        // GET request
        myResourceInstance = MyResource.get({ id: 1 });

        // POST request
        myResourceInstance.$save();

        // PUT request
        myResourceInstance.$update();

        // DELETE request
        myResourceInstance.$delete();

}]);
```

Similar to `$http`, `$resource` returns promises for handling asynchronous operations, allowing developers to use the `.then()` method.

```
MyResource.get({ id: 1 }).$promise.then(function(response) {
    // Handle successful response
}, function(error) {
    // Handle error response
});
```

$q

Overview

In AngularJS, the `$q` service is a promise library that allows for the management of asynchronous operations. It is used to create and manage promises, handle their resolution or rejection, and perform operations when multiple promises need to be coordinated.

Reference: Angular JS/services/$q

Example-

```
var promise = $q.defer().promise;
promise.then(
    function (result) {
        // Promise resolved
    },
    function (error) {
        // Promise rejected
    }
);
```

$uibModalInstance

Overview

In AngularJS, the `$uibModalInstance` service provided by the Angular UI Bootstrap library represents an instance of a modal window. It is used to interact with and control the state of a modal dialog, which is a common UI component for displaying content or capturing user input in a separate overlay.

Reference: Angular UI Bootstrap/modal

Toaster

Overview

In AngularJS, a "`toaster`" is a UI component used to display notifications or alerts in a user-friendly way. Although the AngularJS core does not include a specific "toaster" module, developers often use third-party libraries to implement toast notifications.

Example-

Success Message -

```
toaster.success({
       body: 'Configuration updated successfully.'
});
```

Error Message -

```
toaster.error({
       body: 'Configuration Failed to save.'
});
```

Info Message -

```
toaster.info({
       body: 'Live sync is now active.'
});
```

$timeout

Overview

In AngularJS, the `$timeout` service is a wrapper around the native '`setTimeout`' function in JavaScript. It allows you to to execute a function or evaluate an expression after a specified delay. Here are some important details about the `$timeout` service:

- **Service Injection**: Similar to other AngularJS services, you need to inject the '`$timeout`' service into controllers, services, or other AngularJS components where you need the asynchronous timing functionality.

  ```
  angular.module('myApp').controller('MyController', ['$timeout', function($timeout) {
      // Controller logic with $timeout usage
  }]);
  ```

- **Usage**: '`$timeout`' is commonly used to introduce delays in code execution, especially when you want to wait for a specific event or update the UI after a certain time.

```
$timeout(function() {
       // Code to be executed after a delay
       console.log('Delayed execution');
   }, 1000); // 1000 milliseconds (1 second) delay
```

- **Promise-Based API**: '`$timeout`' returns a promise that is resolved when the delay has passed and the provided function or expression has been executed.

```
var delayPromise = $timeout(function () {
       // Code to be executed after a delay
       console.log('Delayed execution');
   }, 1000);

   delayPromise.then(function() {
       console.log('Delay completed');
   });
```

- **Canceling the Timeout**: '`$timeout`' returns a promise that can be used to cancel the timeout before it occurs using the '`cancel`' method.

```
var delayPromise = $timeout(function() {
    // Code to be executed after a delay
    console.log('Delayed execution');
       }, 1000);

       // Cancel the timeout before it occurs
       $timeout.cancel(delayPromise);
```

Reference: Angular JS/services/$timeout

$filter

Overview

In AngularJS, the `$filter` service provides a set of built-in filters that can be used to filter arrays or objects. These filters allow you to display data in the desired format on the user interface. Additionally, you can create custom filters. Here are some important details about the `$filter` service:

- **Service Injection**: Similar to other AngularJS services, you need to inject the '`$filter`' service into controllers, services, or other AngularJS where data filtering is needed.

```
angular.module('myApp').controller('MyController', ['$filter', function($filter)
   {
       // Controller logic with $filter usage
   }
]);
```

- **Usage**: '`$filter`' is used to apply filters to data in templates using the '`{{ expression | filter:arguments }}`' syntax. You can also use filters programmatically in controllers and services. For

example, the 'uppercase' filter can convert a string in the 'data' variable to uppercase as follows:
`<$div>{{ data | uppercase }}</div>`

- **Built-in Filters**: AngularJS provides a set of built-in filters for common filtering and formatting tasks. Some examples include:
  - `uppercase`: Converts a string to uppercase.
    `<div>{{ name | uppercase }}</div>`
  - `lowercase`: Converts a string to lowercase.
  - currency: Formats a number as currency.
    `<div>{{ amount | currency }}</div>`
  - `date`: Formats a date.
    `<div>{{ currentDate | date:'yyyy-MM-dd' }}</div>`
  - `number`: Formats a number.
- **Chaining Filters**: You can chain filters together to perform multiple transformations on the data. For example, to convert text to uppercase and then limit the result set to 10, use the following:
  `<div>{{ text | uppercase | limitTo:10 }}</div>`
- **Custom Filters**: You can define custom filters to perform specific formatting or filtering tasks. Custom filters are created using the `filter` method of the `$filter` service.

```
angular.module('myApp').filter('customFilter', function() {
    return function(input) {
        // Custom filtering logic
        return filteredOutput;
    };
});

<div>{{ data | customFilter }}</div>
```

- **Using Filters Programmatically**: You can apply filters programmatically in controllers or services using the `$filter` service. For example, the `uppercase` filter can be applied to the text 'Hello' using `$filter ('uppercase')`:

```
angular.module('myApp').controller('MyController', ['$filter', function($filter) {
       var uppercaseFilter = $filter('uppercase');
       $scope.uppercasedText = uppercaseFilter('Hello');
    }]);
```

- **Filtering Arrays**: You can use the `filter` method of the `$filter` service to filter arrays based on certain criteria. For example, to display only items with the category 'Electronics' from a given item list, use the following:
  `<div ng-repeat="item in items | filter: { category: 'Electronics' }">{{ item.name }}</div>`
- **Filtering with Functions**: You can use the `filter` method of the `$filter` service to accept functionsa s arguments. This allows for more complex filtering logic to be applied:

```
<div ng-repeat="item in items | filter: filterFunction">{{ item.name }}</div>

   $scope.filterFunction = function(item) {
       // Custom filtering logic
       return item.price > 100;
   };
```

Reference: Angular JS/filter/filter

_(Underscore Library)

Overview

`Underscore.js` is a utility library that provides functional programming helpers without extending any built-in objects. It includes functions for working with arrays, objects, collections, and functions.

Reference: Underscore.js

Example-

```
var _ = require('underscore');

var numbers = [1, 2, 3, 4, 5];
var sum = _.reduce(numbers, function (memo, num) {
    return memo + num;
}, 0);

console.log(sum); // Output: 15
```

$window

Overview

In AngularJS, the `$window` service is a wrapper for the global `window` object in JavaScript. It allows interaction with the browser's window object within an AngularJS application.

Reference: Angular JS/services/$window

$state

Overview

In AngularJS, the `$state` refers to the UI-Router's `$state` service. UI-Router is a flexible and powerful alternative to AngularJS's built-in routing mechanism. It enables developers to define states, views, and transitions within their AngularJS applications. The `$state` service is used for state management and navigation.

WizardHandler

Overview

In AngularJS, a wizard or multi-step form is a common UI pattern that breaks down a complex form into a series of steps or sections to enhance user-friendliness.

Reference: Angular Wizard

LocalStorageService

Overview

`LocalStorageService` is a custom service or library that provides methods or functions for interacting with the browser's local storage. Local storage is a key-value storage mechanism available in web browsers, commonly used for persisting small amounts of data on the client-side. Here is a general outline of such a service:

- **Service Injection**: The `LocalStorageService` service needs to be injected into AngularJS components where local storage operations are required.

```
angular.module('myApp').controller('MyController', ['LocalStorageService', function
(LocalStorageService) {
  // Controller logic with LocalStorageService usage
 }]);
```

- **Methods for Local Storage Operations**: Such services typically offer methods for common local storage operations, such as getting, setting, and removing items.

```
// Example methods in a LocalStorageService
LocalStorageService.setItem('key', 'value');
var storedValue = LocalStorageService.getItem('key');
LocalStorageService.removeItem('key');
```

PromiseQueue

Overview

In a generic sense, a 'Promise Queue' is a mechanism for managing a series of asynchronous tasks, where each task returns a promise. This ensures that tasks are executed in a specific order, with the next task only starting once the previous one has been completed.

Reference: PromiseQueue

Constants

Overview

The list of constants used in an application can vary based on the application's architecture, design patterns, and requirements. To use API constants in a component or controller, inject the 'API constant' and then use them in your application.

API Constants

```
('API',
       {
       BASE: 'api/3/',
       API_3_BASE: '/api/3/',
       TEMPLATE: 'api/3/template/',
       WORKFLOW: 'api/wf/',
       WORKFLOW_HEALTH: 'api/wf/workflow/healthcheck/job/',
       INTEGRATIONS: 'api/integration/',
       SEALAB: 'wf/',
       ETL: 'gateway/etl/',
       AUDIT: 'api/gateway/audit/',
       AUTH: 'api/auth/',
       PUBLIC: 'api/public/',
       QUERY: 'api/query/',
       QUERIES: 'api/3/queries',
       REPORTS: 'gateway/report/',
       DAS: 'auth/',
       POSTMAN: 'api/postman/',
       SAML: 'api/saml/',
       SEARCH: 'api/search/',
       ARCHIVAL: 'api/archival/',
       PUBLISH: 'api/publish',
       MANUAL_TRIGGER: 'api/triggers/1/notrigger/',
       WORKFLOW_STEPS: 'workflow_steps/',
       WORKFLOW_GROUPS: 'workflow_groups/',
       WORKFLOW_BLOCKS: 'workflow_blocks/',
       WORKFLOWS: 'workflows/',
       WORKFLOW_ACTION: 'api/workflows/actions',
       REVERT: 'api/publish/revert',
       PUBLISH_ERROR: 'api/publish/error',
       ACTION_TRIGGER: 'api/triggers/1/action/',
       CURRENT_AVATAR: 'avatars/current',
       CURRENT_ACTOR: 'actors/current',
       AUTHENTICATION: 'authentication',
       ROLES_TEAM_READ_ONLY: 'api/userteam',
       USER_PREF_PREFIX: 'user/view/',
       REMOTE_ACTION_EXECUTION:'api/integration/remote-action-execution/',
       API_HMAC_TRIGGER_URL: 'api/triggers/1/',
       IMPORT: 'api/import/',
       EXPORT: 'api/export/',
       SYSLOG_CONFIG: 'api/gateway/config/syslog',
       API: 'api',
       WEBSOCKET: 'websocket/cyops-websocket',
       SYSTEM_MODULES: 'api/system/fixtures',
       RULE: 'api/rule/',
       DELETE_WITH_QUERY: 'api/3/delete-with-query/',
       EXPORT_TEMPLATES: 'api/3/export_templates/',
       SOLUTION_PACKS: 'api/3/solutionpacks/'
       }
)
```

Example-
URL - `API.SAML + 'metadata';`

Playbook Step Types

```
('PLAYBOOK_STEP_TYPES',
       {
       API_TRIGGER: 'cybersponse.api_call',
       ACTION_TRIGGER: 'cybersponse.action',
       ABSTRACT_TRIGGER: 'cybersponse.abstract_trigger',
       DECISION: 'Decision',
       MANUAL_DECISION: 'ManualDecision',
       MANUAL_INPUT: 'ManualInput',
       APPROVAL_MANUAL_INPUT: 'ApprovalManualInput',
       SET_VARIABLE:'SetVariable',
       INSERT_DATA:'InsertData',
       UPDATE_DATA:'UpdateRecord',
       REFERENCE_BLOCK: 'ReferenceBlock',
       TRIGGER_REFERENCE_BLOCK: 'action.reference.block',
       POST_DELETE_TRIGGER: 'cybersponse.post_delete',
       PRE_DELETE_TRIGGER: 'cybersponse.pre_delete',
       POST_CREATE_TRIGGER: 'cybersponse.post_create',
       PRE_CREATE_TRIGGER: 'cybersponse.pre_create',
       POST_UPDATE_TRIGGER: 'cybersponse.post_update',
       PRE_UPDATE_TRIGGER: 'cybersponse.pre_update',
       MANUAL_DECISION_STEP_TYPE: '/api/3/workflow_step_types/dc61b68b-4967-4e82-b4ed-a1315aa81998',
       MANUAL_INPUT_STEP_TYPE: '/api/3/workflow_step_types/fc04082a-d7dc-4299-96fb-6837b1baa0fe'
       }
)
```

appModulesService

Overview

The `appModulesService` loads application modules and provides state information based on the given module names.

Reference: appModulesService

Function

- `getListState` - Returns the list view state for a given module name.

Example-
`let stateName = appModulesService.getListState(module.type);`

Common Utils

Overview

The `CommonUtils` API provides commonly used functions that can be utilized across different parts of your application.

Reference: CommonUtils

Functions

- `copyToClipboard` - Used to copy content to the clipboard.
- `generateUUID` - Used to create and return UUIDs.
- `isBase64Image` - Used to check whether the provided value is in the Base64 image format.
- `isJinja` - Used to check whether the provided value is in the jinja format.
- `isNumber` - Used to check whether the provided value is a number.
- `isObject` - Used to check whether the provided value is in the object format.
- `isUUID` - Used to check whether the provided value is a valid UUID.
- `isUndefined` - Used to check whether the provided value is undefined or null.
- `isValidURL` - Used to check whether the provided value is a valid URL.
- `parseJSON` - Used to convert the provided input data into the JSON format.
- `searchInJSON` - Used to search whether the provided value exists in the specified JSON object.

Example-

```
let uuid = CommonUtils.generateUUID();
let isJinja = CommonUtils.isJinja(scope.value);
let isNumber = CommonUtils.isNumber(scope.value);
let isUUID = CommonUtils.isUUID(scope.value);
let isUndefined = CommonUtils.isUndefined(scope.value);
```

connectorService

Overview

The `connectorService` API provides operations for managing connectors and agents, facilitating seamless integration between different software systems.

Reference: connectorService

Functions

- `deleteConnector` - Deletes a specified connector.
- `executeConnectorAction` - Explicitly executes a specified connector action.
- `getConnector` - Returns the health status of a specified connector.
- `getDevelopedConnector` - Returns details of a custom created connector. These connectors are in the 'Draft' state and have not been published.
- `updateConnectorConfig` - Creates, updates, or deletes a connector configuration.

Example-

```
connectorService.deleteConnector(conn.id).then(function() {
       toaster.success({
               body: 'Connector removed successfully.'
       });
modalInstance.close();
});

connectorService.getConnector(connector.name, connector.version).then(function (connectorData) {
```

```
                defer.resolve(connectorData);
        }, function (error) {
                defer.resolve(error);
});
```

`currentPermissionsService`

Overview

The `currentPermissionsService` API retrieves permissions for modules within the application.

Reference: currentPermissionsService

Functions

- `availablePermission` - Retrieves permissions for specific actions such as, 'read/write/update/delete' permissions for a provided module.
  `$scope.rulesCanView = currentPermissionsService.availablePermission('rules', 'read');`
- `availablePermissions` - Retrieves permissions for specific actions such as, 'read/write/update/delete' permissions for a provided list of modules.
- `availableFieldPermission` - Retrieves permissions for specific actions such as, 'read/write/update/delete' permissions for a specific field of a provided module.
- `get` - Retrieves all module permissions for the currently logged-in user.
- `getPermission` - Retrieves all permissions of a specific module.
- `getPermissions` - Retrieves all permissions for multiple modules.
  `var permission = currentPermissionsService.getPermission(module.type)`
- `isAdmin` - Retrieves the status of the currently logged-in user's 'Security Update' permission.
  `$scope.isAdmin = currentPermissionsService.isAdmin();`

`Entity`

Overview

The `Entity` Service API provides functionalities for managing modules, handling metadata and fields, loading and populating data based on permissions, module names, and IDs. It also takes care of adding and removing relationships and transforming form data.

Reference: Entity

Functions

- `applyDefaultValues` - Sets default values for all fields.
- `delete` - Deletes an entity
- `evaluate` - Evaluates query filters.
- `evaluateAllFields` - Evaluates visibility and required filters for all entity fields.
- `get` - Returns properties of all the fields.
- `getFormFields` - Returns an object of the form fields including system fields of the entity.
  `scope.fields = scope.entity.getFormFields();`
- `getFormFieldsArrayWithoutSystem` - Returns an array of form fields excluding system fields of the entity.

- `getRelationship` - Retrieves relationship data for a provided field.
- `loadFields` - Loads fields of a provided module.

  ```
  entity.loadFields().then(function() {
         entityDefer.resolve();
  }, function(error) {
         entityDefer.reject(error);
  });
  ```

- `save` - Saves the entity.

exportService

Overview

The `exportService` API provides functionalities for copying, exporting, and downloading playbooks.

Reference: exportService

Functions

- `changePlaybookAndStepsUuid` - Replaces the UUID of an existing playbook with the UUID of an exported playbook.
- `copyEntities` - Copies 'entities' of type 'entityName' (workflows) and returns the result. 'workflows' need their UUIDs to be updated and their associated steps and routes to be copied.
- `downloadJsonFile` - Forces the download of a JSON file with given data object and filename.
- `exportGridRecords` - Exports a selected playbook collection from a provided grid.
  `scope.exportGridRecords(true, 'Export selected rows as CSV', true);`
- `getMacrosFromPlaybook` - Returns the macros from a provided playbook.
  `var macros = exportService.getMacrosFromPlaybook(workflow, globalVariables);`
- `getReferencePlaybookForExport` - Loads the reference playbooks associated with the playbook being exported.
  `scope.fields = scope.entity.getFormFields();`
- `loadCollectionNames` - Loads the names of all playbook collections.
- `loadCollectionPlaybooks` - Fetches all playbooks from a provided collection.
- `loadRowsForExport` - Loads exported versions of given rows from 'crudhub'.
- `saveNewMacrosFromPlaybook` - Saves the macros in the database.

Field

Overview

The `Field` Service API provides functionalities for representing the model of a field.

Reference: Field

Functions

- `evaluateRequired` - Evaluates the required constraints of a given field.
  `scope.field.evaluateRequired(entity);`

- `evaluateVisible` - Evaluates the visible constraints of a given field.
- `getFormValue` - Converts the stored value of a given field into its raw value, which can be of various types.
  `var fieldValue = field.getFormValue();`

FormEntityService

Overview

The `FormEntityService` API provides functionalities for handling entity-related operations, including setting, getting, and submitting fields in a form.

Reference: FormEntityService

Functions

- `get` - Retrieves a provided entity.
- `set` - Saves a provided entity.
  `FormEntityService.set(entity);`

licenseService

Overview

The `licenseService` API provides functionalities for retrieving and updating license details.

Reference: licenseService

Functions

- `getBrandingDetails` - Retrieves the branding details of the instance.

```
licenseService.getLicenseDetails().then(function(license) {
       $scope.licenseDetails = license;
       if(license.details.is_distributed){
               columns.push('remoteExecutableFlag');
       }
});
```

- `set` - Retrieves the license details of the instance.

modelMetadatasService

Overview

The `modelMetadatasService` API provides functionalities for loading and retrieving the model metadata information.

Reference: modelMetadatasService

Functions

- `getIriByType` - Retrieves the model metadata IRI for a given module type.
- `getMetadataByModuleType` - Retrieves the model metadata type for a given module type.
- `getModuleList` - Checks local storage for model metadata. If the module list is not in the storage, it falls back to checking whether another promise to get them has already been set. If one has not been set, then an API request is made to retrieve the module list and then store them in local storage.

```
modelMetadatasService.getModuleList().then(function(modules) {
       scope.modules = modules;
});
```

- `getModuleNameByType` - Retrieves the model name in the singular or plural form from the local storage for a given module type.

```
scope.modules = modelMetadatasService.getModuleNameByType(triggerStep.arguments.resources,
true);
```

- `getTenantModuleList` - Returns the module metadata for a specific tenant based on its UUID.
- `getTenantStaggingModule` - Returns the staging module metadata for a specific tenant based on its UUID and module ID.
- `loadAllModules` - Loads all modules including system modules.
- `publishTenant` - Initiates the 'Publish' process for a specific tenant based on its UUID.
- `loadRowsForExport` - Loads exported versions of the given rows from 'crudhub'.
- `saveNewMacrosFromPlaybook` - Saves macros in the database.

Modules

Overview

The `Modules` API Service API is a wrapper for **`ng.$resource`**. This API extends `ng.$resource` with additional 'update' functionality, and the optional acceptance of parameters, such as module, id, and fieldname.

Reference: Modules

Functions

- `save` - Updates or saves a given module.

```
promise = Modules.save({
       module: module,
       $relationships: true
}, queryToSave).$promise;
```

picklistsService

Overview

The `picklistsService` API facilitates actions for retrieving and manipulating picklists.

Reference: picklistsService

Functions

- `getPicklistByIri` - Returns the top-level picklist item and its child picklists, or the child picklist itself, based on the given IRI.

  ```
  picklistsService.getPicklistByIri(value).then(function(data) {
         interpolateObject[key] = data.itemValue;
  });
  ```

- `getPicklistByItemValue` - Retrieves the picklist for a given field, based on the given itemValue.
- `loadAllPicklists` - Returns all picklist values.

  ```
  picklistsService.loadAllPicklists().then(function (data) {
         scope.picklistNames = data;
  });
  ```

- `loadPicklistsByParams` - Returns the top-level picklist item and its child picklists, or the child picklist itself, based on the given picklist name.

playbookService

Overview

The `playbookService` API provides operations related to playbooks, such as retrieving playbook details, creating playbooks, and accessing playbook execution log details.

Reference: playbookService

Functions

- `getPlaybookExecutionCount` - Returns the total playbook execution count.

  ```
  playbookService.getPlaybookExecutionCount($scope.params.query).then(function(result) {
         Console.log(data);
  });
  ```

- `getRunningPlaybookDetails` - Returns details of a specific running playbook.
- `getRunningPlaybooks` - Retrieves a list of running playbooks from the playbook execution history.

  ```
  playbookService.getRunningPlaybooks(query).then(function(result) {
         scope.childRecords.reference = result['hydra:member'];
  });
  ```

- `getStepRunningDetails` - Returns the of a specific step in a specific playbook.

PromiseQueue

Overview

The `PromiseQueue` Service API provides functionalities for storing, retrieving, and clearing promises, which helps optimize resource usage when making API calls in quick succession.

Reference: PromiseQueue

Functions

- `clear` - Reinitializes the queue.
- `get` - Retrieves a promise from a given queue.

```
var promise = PromiseQueue.get('picklists');
```

- `set` - Sets a promise into a given queue for later retrieval.

```
PromiseQueue.set('picklists', promise);
```

`Query`

Overview

The `Query` Service API provides an object for API query parameters and filters, exposing methods for retrieving and manipulating data.

Reference: Query

Functions

- `getFlatQuery` - Returns a properly evaluated query.

```
var flatQuery = self.query.getFlatQuery();
```

- `getQuery` - Forms a query object with all required parameters. Query object parameters include `sort`, `limit`, `logic`, `search`, `showDeleted`, `filters`, `$relationships`, `aggregates`, and `__selectFields`.
- `getQueryModifiers` - Modifies query modifiers, such as 'limit' to '`$limit`', 'page' to '`$page`'.
- `updateFilter` - Updates a specific filter object.
- `updateFilters` - Updates a specific filter array.

`queryCollectionService`

Overview

The `queryCollectionService` API provides functionalities for loading, saving, manipulating, and deleting query collections.

Reference: queryCollectionService

Functions

- `load` - Loads the queries for a specific module and stores a promise for all future requests.

```
queryCollectionService.load(scope.entity.name).then(function(collection) {
        scope.queries = collection;
});
```

- `loadQueryFilterValues` - Updates the values of a provided query's filters with either a picklist object or a display value.

```
var promise = queryCollectionService.loadQueryFilterValues(scope.selectedQuery);
```

- `loadResource` - Loads a specific object using its IRI, and applies the display value once the object is loaded.

PagedCollection

Overview

The `PagedCollection` Service API creates objects that work against the paged '`crudhub`' API. It serves as the base class for other `PagedCollection` child classes.

Reference: PagedCollection

Functions

- `buildSortQuery` - Builds a query to sort results.
- `convertToKeyPairs` - Converts 'hydra:member' records to key pairs.
- `extendFilter` - Extends a filter condition, i.e., merges multiple filter conditions and then applies the merged filter.
  For example, one filter is name = alert and another is severity = high, `extendFilter` merges both these filters into a single filter and then applies the same to the grid.
- `gotoPage` - Navigates to a specific page within the data.
- `load` - Loads all records with customQuery, customColumns, and relationships.
- `loadByPost` - Loads the grid data using the 'POST' call.
- `loadDefaultColumns` - Loads the default columns of a given grid.

```
pagedCollection.loadDefaultColumns().finally(function() {
       promise = pagedCollection.loadGridRecord(_config, true);
       checkPromise(promise, pagedCollection);
});
```

- `loadGridRecord` - Loads the records of a given grid.

```
pagedCollection.loadGridRecord().then(function() {
       scope.gridApi.grid.options.columnDefs = scope.pagedCollection.columns;
}, dataError);
```

- `pageFirst` - Navigates to the first page of the data.
- `pageLast` - Navigates to the last page of the data.
- `pageNext` - Navigates to the next page within the data.
- `pagePrevious` - Navigates to the previous page within the data.
- `setPage` - Sets records according to the specified pagination.
- `sortColumnsByFieldName` - Sorts columns by a given 'field name'.

settingsService

Overview

The `settingsService` API provides functionalities for getting and setting user and system settings.

Reference: settingsService

Functions

- `get` - Gets value of a setting for the provided key.

  ```
  $scope.countryId = settingsService.get('user/phone/countryId');
  ```

- `getSystem` - Checks the local storage for system settings. If the system settings are not in the storage, it falls back to checking whether another promise to get them has already been set. If one has not been set, then an API request is made to retrieve the system settings and store them in local storage.

  ```
  settingsService.getSystem().then(function (response) {
          scope.systemSettings = response;
  });
  ```

tokenService

Overview

The `tokenService` API manages idle and session timers, offering methods for manipulating and retrieving information from JWT web tokens.

Reference: tokenService

Functions

- `get` - Gets the authentication token.

  ```
  var token = tokenService.get();
  ```

- `set` - Sets the encrypted token and user information in local storage. It also create timers to validate tokens for refreshing and adding listeners.

  ```
  tokenService.set(response.token);
  ```

usersService

Overview

The `usersService` is an auxiliary service for user-related actions. It includes methods for interacting with the current user and updating DAS (Data Access System) users.

Reference: usersService

Functions

- `getAvatar` - Gets the avatar img src with base64 encoded image data for the given IRI. If the avatar IRI is empty, it will return the default avatar URI.
- `getCurrentAvatar` - Gets the avatar of the current user from local storage if it exists; otherwise, loads it using the `getAvatar` function and saves it in local storage.
- `getCurrentUser` - Gets the current user that was loaded using the `loadCurrentUser` function.

```
var user = usersService.getCurrentUser();
```

- `getUserByIri` - Gets the user details based on the given IRI.

```
usersService.getUserByIri(CommonUtils.getIriApiPath(userIri)).then(function(result){
       if(result.data['@type']==='Person'){
       scope.executionDetail.userDisplayName = result.data.firstname + ' ' + result.data.lastname;
       }
});
```

- `loadCurrentUser` - Loads the current user object from 'crudhub' and stores it in local storage, or retrieves it from the local storage cache if already loaded.

ViewTemplateService

Overview

The `ViewTemplateService` provides functionalities for loading and saving view templates.

Reference: ViewTemplateService

Functions

- `changeStructure` - Updates the structure of a row/column-based view template widget. Used when the layout or number of columns is changed.
- `get` - Checks local storage for the requested page and returns the page. If the page is not found in the storage, it falls back to checking whether another promise to get them has already been set. If one has not been set, then an API request is made to retrieve the requested page and stores it in local storage.

```
var appTemplatePromise = ViewTemplateService.get('app').then(angular.noop,$q.reject).finally
(function () {
       console.log('App View Template');
});
```

- `getConditionalVisibilityFilteredData` - Fetches records for an entity based on a provided visibility filter. Used to handle widget and tab visibility settings on the Dashboard, Reports, and View Panels.
- `populateConditionalFields` - Populates an array with fields used in the conditional filter. Used to handle widget and tab visibility settings on the View Panels.

widgetTemplateService

Overview

The `widgetTemplateService` API provides functionalities for loading widget definitions.

Reference: widgetTemplateService

Functions

- `generateWidgetDefinition` - Creates widget data in a generic format that can be used to render the widget in different layouts, such as popup, half widget, or full widget.

```
var widgetDefinition = widgetTemplateService.generateWidgetDefinition(widget);
```

Directives

This topic provides information about the various custom directives that are available for developing widgets.

Field Directive

Overview

The "`cs-field`" directive dynamically renders fields based on a specified form type. It enforces that the rendered fields are required and binds the value to the `ng-model` attribute.

Selector to render a field - **`data-cs-field`**

Usage

```
<div data-cs-field="field" data-mode="'add'" data-ng-model=" value" data-size="'small'" data-
change-method="changeMethod" data-enable-jinja="true" data-enable-expression="enableExpression"
data-allow-add-tag="true"
       data-blur-method="changeMethod" data-ignore-editable="true" data-disabled="disabled" data-use-
placeholder="true" data-fields-mapping="fieldsMapping" data-form-name="'fieldForm'">
</div>
```

Arguments

Parameters	Definition	Data Type

Field
(Required)
	Field definition for rendering a field.
	
	`scope.field = new Field({`
	`   'name':'password',`
	`   'formType': 'password',`
	`   'title': 'Password',`
	`   'writeable': true,`
	`   'validation': {`
	`   'required': true`
	`  }`
	`});`
	
	FormType
	• password
	• text
	• checkbox
	• integer
	• checkbox.select
		Field

	• decimal • datetime • Datetime.advance • phone • email • DynamicList • multiselect • richtext • json • textarea • picklists • multiselectpicklist • lookup • ipv4 • ipv6 • domain • url • tags	
ng-mode (Required)	Used to store/populate the value of a field after it is rendered or changed.	Any value based on field type

FieldOptions
(Optional)
	Used in ViewPanel SVT to set fields such as linky, readonly, highlightMode etc.
	For example,
	`fieldOptions = {linky: boolean, readOnly: boolean, highlightMode: boolean}`
	For multiselect fields, it can also pass options such as,
	`{entity: {module: string, id: string }}`
		object

UsePlaceholder (Optional)	Used to enable use of a placeholder in a field.	boolean
Size (Optional)	Size of the field to be rendered. Options can be large, small.	string
ChangeMethod (Optional)	Method to be called after the field value is changed.	function
BlurMethod (Optional)	Method to be called after the blur event of a field.	function
FocusMethod (Optional)	Method to be called after the focus event of a field.	function
Disabled (Optional)	Used to enable or disable a field.	boolean
EnableJinja (Optional)	Used to enable Jinja for a field.	boolean

EnableExpression (Optional)	Used to enable expressions for a field.	boolean
Mode (Optional)	Used to check whether the field is in the 'edit' mode.	boolean
fieldsMapping (Optional)	Used to update the `fieldType` based on the field mapping.	object

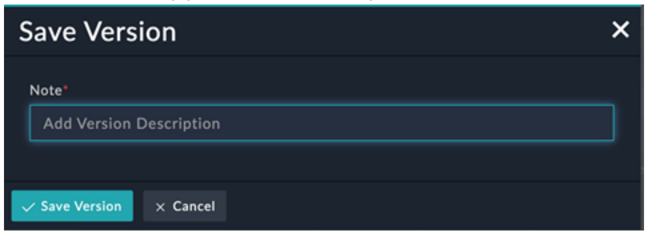
Examples

Create an input text field using the field directive:

```
<div data-cs-field="field" data-ng-model="playbookNote" data-use-placeholder="'Add Version
Description'" data-form-name="'playbookVersionForm'"></div>

$scope.field = new Field({
       'name': 'note',
       'formType': 'text',
       'title': 'Note',
       'writeable': true,
       'validation': {
       'required': true
   }
})
```

The above snippet creates a input text field named "Note":

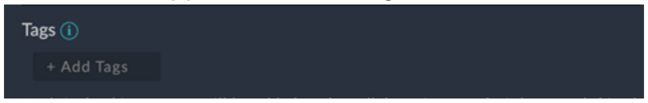


Create a tag field using field directive:

```
<div data-cs-field="tagsField" data-ng-model="tag" data-use-placeholder="'Add tags'" data-form-
name="'playbookVersionForm'"></div>

$scope.tagsField = new Field({
       'name': 'tags',
       'formType': 'tags',
       'title': 'Tags',
       'writeable': true,
       'validation': {
       'required': true
   }
})
```

The above snippet creates a "tag" field:



Conditional Filters

Overview

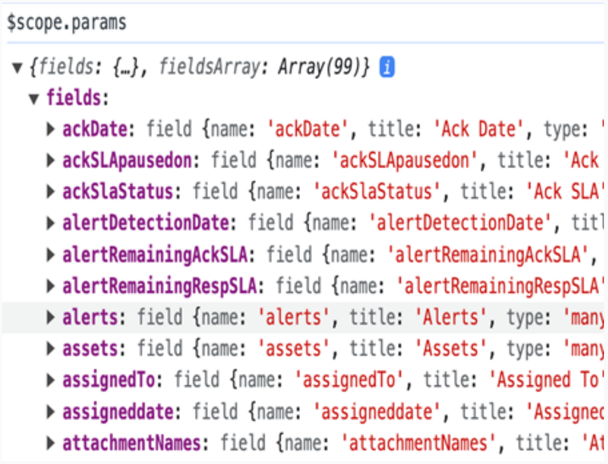
The "`cs-conditional`" directive dynamically renders conditional fields based on a specified form type. It enforces that the rendered fields are required and binds the value to the `ng-model` attribute.

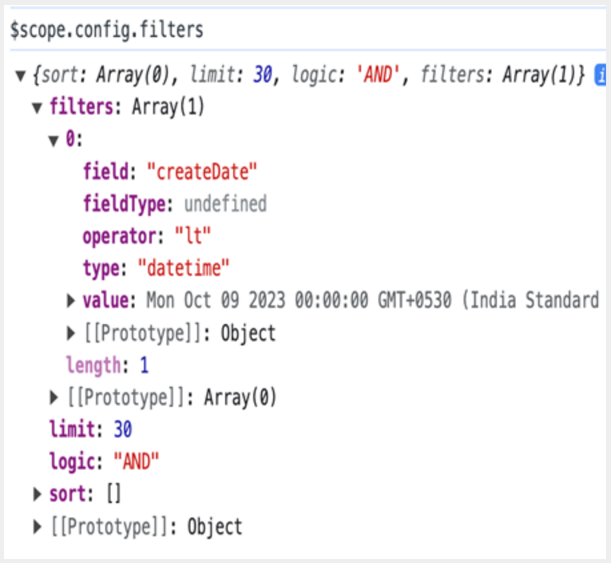
Selector to add conditional filters - **`data-cs-conditional`**

Usage

```
<div data-cs-conditional data-enable-nested-filter="enableNestedFilter"
data-fields="params.fields" data-ng-model="config.filters" data-form-name="'editWidgetForm'" data-
parent-form="editWidgetForm"
data-reset-field="params.fields"></div>
```

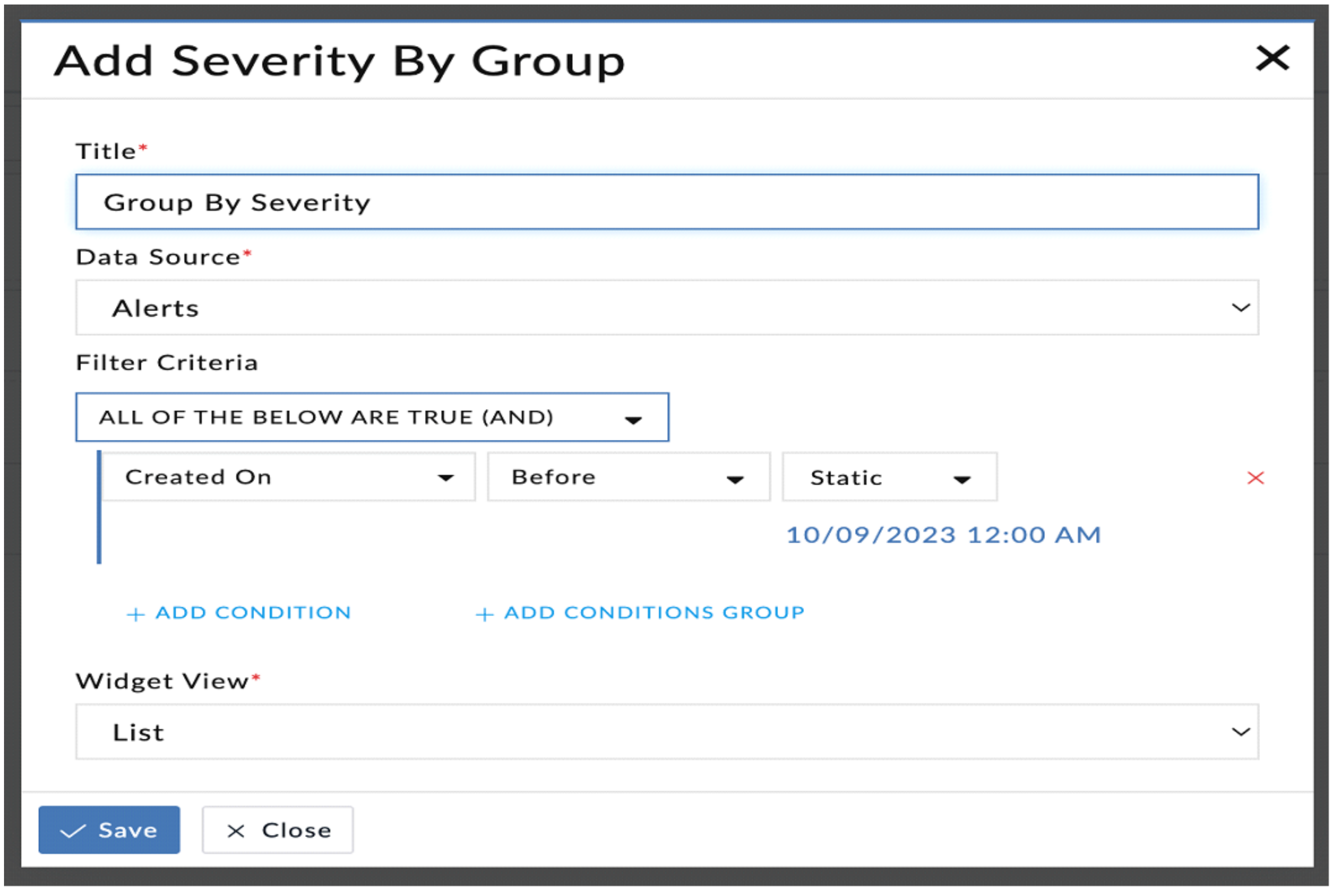
Arguments

Parameters	Definition	Data Type
data-fields (Required)	Populates the fields to be applied in the filter drop-down. Fields parameters are mandatory.  `$scope.params` `▼{fields: {…}, fieldsArray: Array(99)}` `▼fields:` `▶ackDate: field {name: 'ackDate', title: 'Ack Date', type: '` `▶ackSLApausedon: field {name: 'ackSLApausedon', title: 'Ack` `▶ackSlaStatus: field {name: 'ackSlaStatus', title: 'Ack SLA'` `▶alertDetectionDate: field {name: 'alertDetectionDate', titl` `▶alertRemainingAckSLA: field {name: 'alertRemainingAckSLA',` `▶alertRemainingRespSLA: field {name: 'alertRemainingRespSLA'` `▶alerts: field {name: 'alerts', title: 'Alerts', type: 'many` `▶assets: field {name: 'assets', title: 'Assets', type: 'many` `▶assignedTo: field {name: 'assignedTo', title: 'Assigned To'` `▶assigneddate: field {name: 'assigneddate', title: 'Assignec` `▶attachmentNames: field {name: 'attachmentNames', title: 'At`	Array of an object `[Array<Object>]`
data-ng-model (Required)	Used to assign the selected filters in the filter criteria.	Array of an object `[Array<Object>]`

		
data-enable-nested-filter (Optional)	Used to allow nested fields in filters.	boolean
Template (Optional)	Used only if a custom template is defined. This parameter is used to render all conditional fields. By default, the built-in template is used, so nothing needs to be specified for this parameter.	HTML template
data-form-name (Optional)	Used to create the current form name in HTML.	string
data-reset-name (Optional)	Used to reset a query if any field passed in this parameter is updated. Fields are similar to data-fields.	Array of an object `[Array<Object>]`
data-hide-related-fields (Optional)	Used to hide related fields.	boolean
data-mode (Optional)	Used to specify the context of the condition. For example, if it is being used as a query filter or as a trigger condition in workflows.	string
EnableJinja (Optional)	Used to enable Jinja for a filter field.	boolean
EnableExpression (Optional)	Used to enable expressions for a filter field.	boolean

```
$scope.config.filters
▼{sort: Array(0), limit: 30, logic: 'AND', filters: Array(1)}
  ▼filters: Array(1)
    ▼0:
       field: "createDate"
       fieldType: undefined
       operator: "lt"
       type: "datetime"
      ▶value: Mon Oct 09 2023 00:00:00 GMT+0530 (India Standard
      ▶[[Prototype]]: Object
     length: 1
    ▶[[Prototype]]: Array(0)
   limit: 30
   logic: "AND"
  ▶sort: []
  ▶[[Prototype]]: Object
```

Example:


Add Severity By Group

Title*

Group By Severity

Data Source*

Alerts

Filter Criteria

ALL OF THE BELOW ARE TRUE (AND)

Created On | Before | Static

10/09/2023 12:00 AM

Widget View*

List

Reference Widget: widget-funnel-chart

data-cs-messages

Overview

The "`data-cs-messages`" directive displays error messages for specific form fields. It requires the form name and form field as input parameters and works in conjunction with AngularJS's built-in form validation.

Selector to render messages - **`data-cs-messages`**

Usage

```
<div data-cs-messages="auth.login_username"></div>
```

Arguments

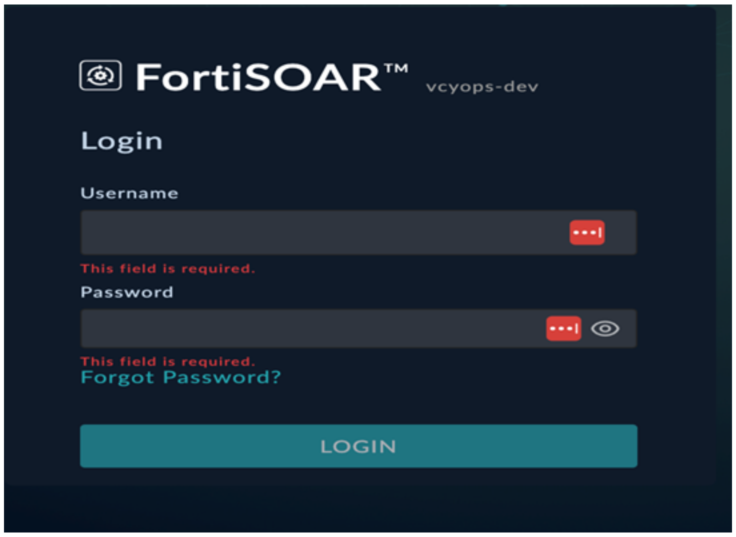
Parameters	Defintion	Data Type
data-cs-messages (Required)	Used to displays error message of a form field. It contains the form name and form field.	string

Example

Create an form field with input parameters that display information that they are required:

```
<div class="form-group margin-top-20">
       <label for="login_username" class="form-label font-size-12">Username</label>
       <input class="form-control" type="text" id="username" name="login_username" data-ng-
model="credentials.loginid" data-cs-focus autocomplete="off" required>
       <div data-cs-messages="auth.login_username"></div>
</div>
```

The above snippet creates a form field with required input parameters that display the "`This field is required`" message.



Grid Directive

Overview

The "`cs-grid`" directive renders data in a grid format. It relies on the "`gridOptions`" property to define the configuration of the grid, and uses the "`pagedCollection`" property to display data in the grid.

Selector to render a grid on pages - **`data-cs-grid`**

Usage

```
<div data-cs-grid data-grid-options="gridOptions" column-defs="columnDefs" data-paged-
collection="pagedCollection"></div>
```

Arguments

Parameters	Definition	Data Type
GridOptions (Required)	Used to add options to the grid as required. `var defaultGridOptions = {`	`{Object}`

```
                csOptions: {
                  allowAdd: true,
                  addText: 'Add',
                  allowLink: false,
                  allowUnlink: false,
                  allowDelete: true,
                  allowClone: true,
                  allowSync: false,
                  allowGlobalFilter: true,
                  allowModuleFilter: false,
                  customRecordTypeFilter:false,
                  allowDateFilter: false,
                  allowUserFilter: false,
                  allowGridFilter: false,
                  allowCardView: false,
                  auditLogView: false,
                  viewType: '',
                  allowActions: false,
                  allowPlaybookActions: false,
                  bulkButtons: [],
                  showPagination: true,
                  searchPlaceholder: 'Search',
                  searchEnable: true,
                  cloneRelationshipsByDefault: false,
                  clone: clone,
                  contextMenu:
        contextMenuService.getConfig,
                  isRelationship: false,
                  searchMinLength: 0,
                  enableSelectMenu: true,
                  enableSavedFilters: true,
                  wideSearchBar: false,
                  unlinkButtonText: 'Remove Link',
                  isFullScreenMode: false
                },
                rowTemplate:
        'app/components/grid/clickableRow.html',
                paginationPageSizes: [5, 10, 30, 50,
        100, 250],
                paginationPageSize: 30,
                onRegisterApi: scope._setGridApi,
                enableGrouping: false,
                enableFiltering: true,
                useExternalFiltering: true,
                useExternalSorting: true,
                enableGridMenu: true,
                enableColumnMenus: false,
```

	`ing: true,`
	`enableColumnMoving: true,`
	`enableExpandable: false,`
	`exporterMenuCsv: false,`
	`exporterMenuPdf: false,`
	`expandableRowHeaderWidth: 0,`
	`expandableRowHeaderTitle: null,`	

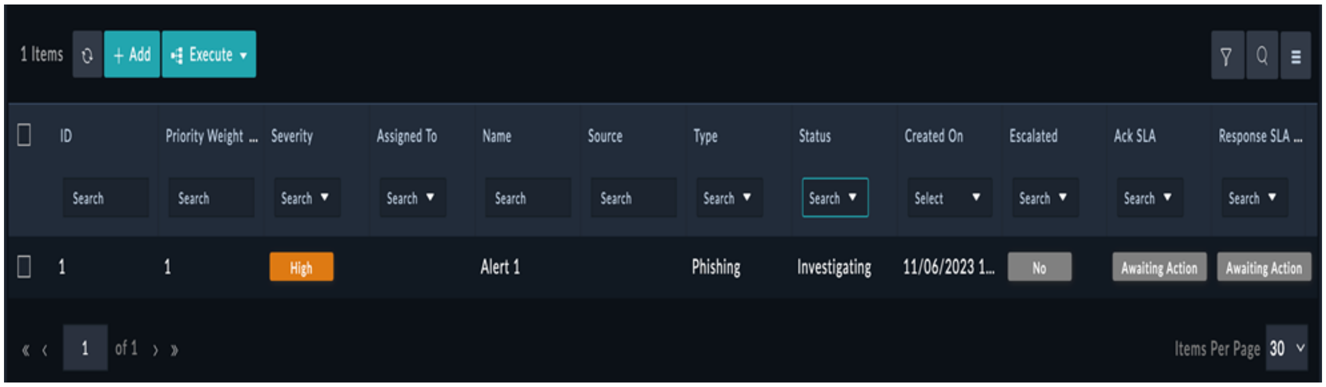
ColumnDefs (Optional)	Used to render the specified column definitions in the JSON format.	`{Object}`
PagedCollection (Required)	The page collection service used to fetch the data that needs to be displayed on the grid.	`Array<Object>`

Example

To display a 'grid' of an 'alert':

```
<div class="col-md-12 collection-grid-container margin-top-lg fade-in-animation">
       <div data-cs-grid data-grid-options="gridOptions" data-paged-collection="pagedCollection"
class="grid-widget-container"></div>
</div>
```

The above snippet is used to display an alert grid:



	ID	Priority Weight ...	Severity	Assigned To	Name	Source	Type	Status	Created On	Escalated	Ack SLA	Response SLA ...
	Search	Search	Search	Search	Search	Search	Search	Search	Select	Search	Search	Search
☐	1	1	High		Alert 1		Phishing	Investigating	11/06/2023 1...	No	Awaiting Action	Awaiting Action

Chart Directive

Overview

The "`cs-chart`" directive renders charts based on selected configuration details. Users can specify chart configuration options and data to render the chart.

Selector to create a chart with selected config details - **`data-cs-chart`**

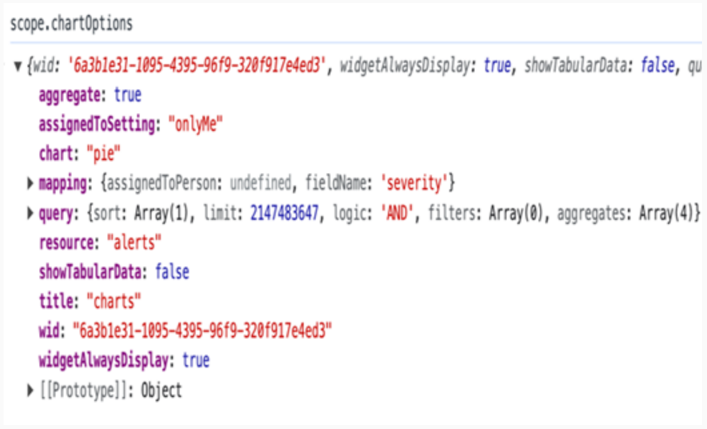
Usage

```
<div data-cs-chart="config"></div>

Config = {
```

```
aggregate: true,
assignedToSetting: "onlyMe",
chart: "pie",
mapping: {
  assignedToPerson: 'assignedTo',
  fieldName: 'severity'
},
query: {
  sort: Array(1),
  limit: 2147483647,
  logic: 'AND',
  filters: Array(1),
  aggregates: Array(4)
},
resource: "alerts",
showTabularData: false,
title: "Open Alerts By Severity",
wid: "f9eb224f-b6bb-4480-a5df-f43dc0ab6d5d",
widgetAlwaysDisplay: true
}
```

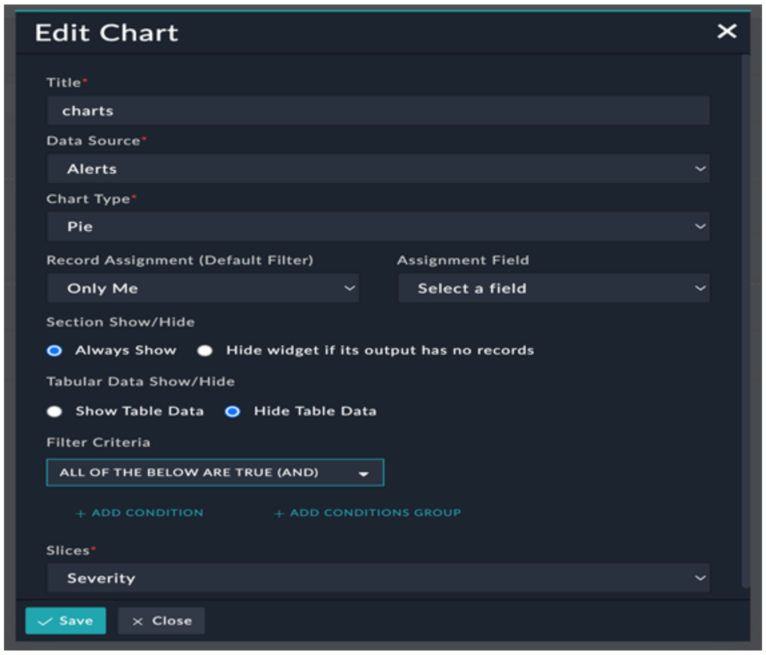
Arguments

Parameters	Definition	Data Type
data-cs-chart	Adds chart configuration details to a widget. 	`[Array<Object>]`

```
scope.chartOptions
▼ {wid: '6a3b1e31-1095-4395-96f9-320f917e4ed3', widgetAlwaysDisplay: true, showTabularData: false, qu
    aggregate: true
    assignedToSetting: "onlyMe"
    chart: "pie"
  ▶ mapping: {assignedToPerson: undefined, fieldName: 'severity'}
  ▶ query: {sort: Array(1), limit: 2147483647, logic: 'AND', filters: Array(0), aggregates: Array(4)}
    resource: "alerts"
    showTabularData: false
    title: "charts"
    wid: "6a3b1e31-1095-4395-96f9-320f917e4ed3"
    widgetAlwaysDisplay: true
  ▶ [[Prototype]]: Object
```

Example

An example of a rendered chart:


Edit Chart

Title*: charts

Data Source*: Alerts

Chart Type*: Pie

Record Assignment (Default Filter): Only Me

Assignment Field: Select a field

Section Show/Hide: [✓ Always Show] Hide widget if its output has no records

Tabular Data Show/Hide: Show Table Data [✓ Hide Table Data]

Filter Criteria: ALL OF THE BELOW ARE TRUE (AND)

Slices*: Severity

Focus Directive

Overview

The "`data-cs-focus`" directive sets focus on an HTML DOM element when applied to that element. This enhances the user experience by automatically focusing on specific form fields or elements.

Selector to focus on an HTML DOM element - **`data-cs-focus`**

Usage

```
<input type="text" id="filename" name="filename" class="form-control" data-ng-model="title" data-
cs-focus required>
```

Arguments

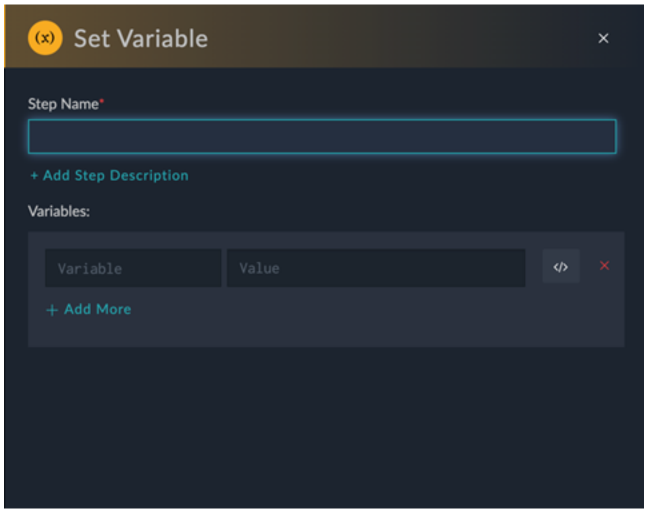
Parameters	Definition	Data Type
data-cs-focus	An attribute directive for adding focus to an HTML element.	

Example

To create a focus on the HTML element, i.e. stepName:

```
<input id="stepName" type="text" class="input-sm form-control" name="stepName" data-ng-
model="stepSelected.name" data-cs-focus data-ng-required="stepSelected">
```

The above snippet is used to display a dialog in which the 'Step Name' field is in focus:



Card Directive

Overview

The "`data-cs-card`"directive displays data as a card in the widget.

Selector to create cards used to display data - **`data-cs-card`**

Usage

```
<div data-cs-card data-ng-model="record" data-size="config.size" data-mapping="config.mapping"
data-actions="actions"></div>
```

Arguments

Parameters	Definition	Data Type
data-ng-model (Required)	Used to display data in a card format.	`Array<Object>`
Size (Required)	Used to define the size of the card widget.	string
data-mapping (Required)	Used to define the list of fields to be displayed in the card widget.	`Array<Object>`

Pagination Directive

Overview

The "`data-cs-pagination`" directive adds pagination to grids or pages.

Selector to add pagination on grids or pages - **`data-cs-pagination`**

Usage

```
<div class="search-pagination"
       data-ng-hide="pagedCollection.filters.q.length === 0 || pagedCollection.filters.index.length ===
0"
       data-cs-pagination data-ng-model="pagedCollection">
</div>
```

Arguments

Parameters	Definition	Data Type
data-ng-model (Required)	Used to display data on the page where pagination is to be applied.	`Array<Object>`

Datetime Grid Directive

Overview

The "`data-cs-datetime-grid`" directive adds datetime functionality to a grid.

Selector to add datetime to a grid - **`data-cs-datetime-grid`**

Usage

```
<div data-cs-datetime-grid="::field"></div>
```

Tags Directive

Overview

The "`data-cs-tags`" directive adds tags to a specified module. Tags are useful for categorizing or labeling items within an application.

Selector to add tags to a specified module - **`data-cs-tags`**

Usage

```
<div class="cs-tags-container padding-left-sm padding-right-sm">
       <div data-cs-tags="tagsField" cs-allow-add-tag="false" data-change-method="tagsChanged" data-ng-
model="item.value.tags"
       data-placeholder="tagsPlaceholder" class="position-relative width-100pt">
       </div>
</div>
```

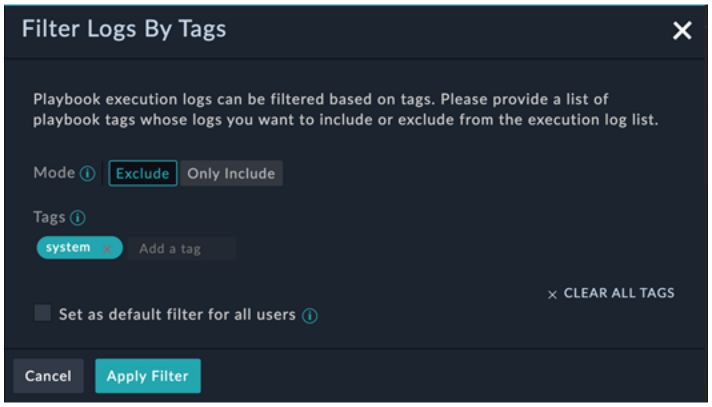
Example

To add tags:

```
<div class="cs-tags-container">
       <div data-cs-tags="tagsField" cs-allow-add-tag="false" data-change-method="tagsChanged" data-ng-
model="input.recordTags"
       data-placeholder="tagsPlaceholder" class="position-relative pull-left">
```

```
        </div>
</div>
```

Example of the 'Filter Logs by Tags' dialog using the 'tags' directive :



Filter Logs By Tags

Playbook execution logs can be filtered based on tags. Please provide a list of playbook tags whose logs you want to include or exclude from the execution log list.

Mode: Exclude | Only Include

Tags: system, Add a tag

CLEAR ALL TAGS

Set as default filter for all users

Cancel | Apply Filter

Markdown Editor Directive

Overview

The "`data-cs-markdown-editor`" directive embeds a markdown editor in the DOM, enabling users to create and edit markdown content in a user-friendly manner.

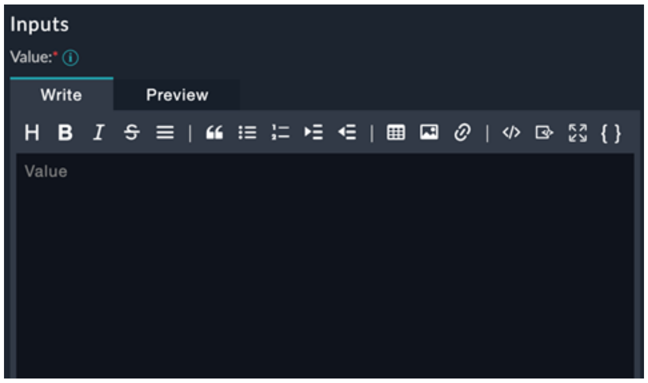
Selector to add tags to embed markdown editor in DOM - **`data-cs-markdown-editor`**

Usage

```
<div data-cs-markdown-editor data-mode="'view'" data-ng-model="task.description"
data-form-name="'task-' + $index"></div>
```

Example

Example of the Value field that using the markdown editor directive to allow users to add or edit content using markdown:



Spinner Directive

Overview

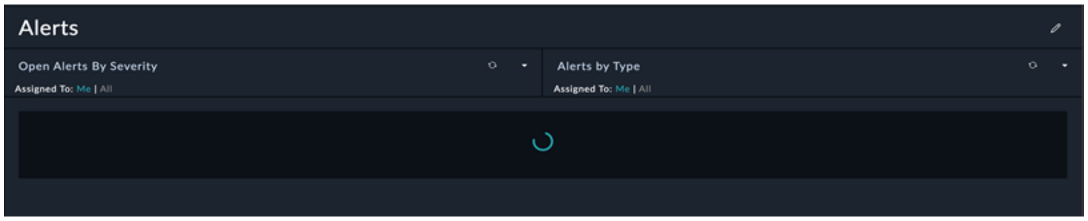
The "`data-cs-spinner`" directive displays a spinner while data is being processed or fetched. This enhances the user experience by indicating ongoing operations.

Selector to add a spinner- **`data-cs-spinner`**

Usage

```
<cs-spinner data-ng-show="processing"></cs-spinner>
```

Example



Unique Directive

Overview

The "`data-cs-unique`" directive checks if a given value is unique within a specified context or dataset, allowing for uniqueness validation on input fields or elements.

Selector to check if the given value is unique or not- **`data-cs-unique`**

Usage

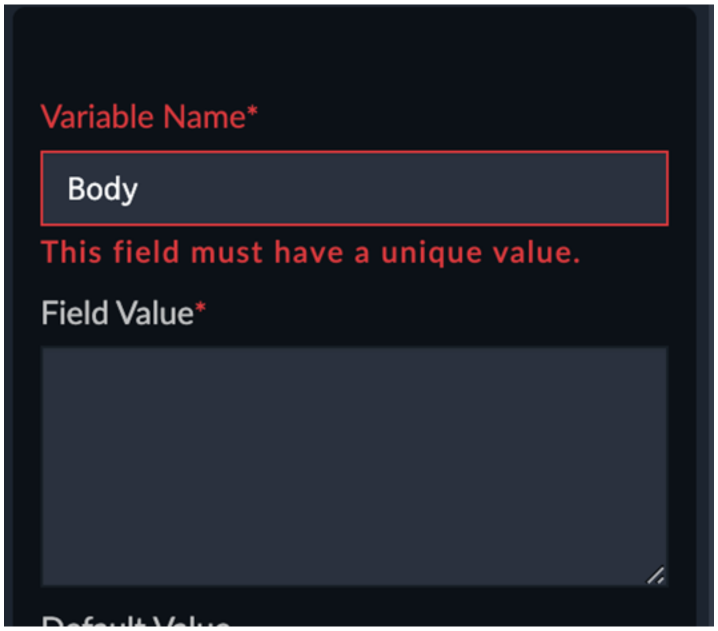
```
<input id="variableName" type="text" class="input-sm form-control" name="variableName" data-ng-
model="dynamicVariable.name"
data-ng-pattern="varRegex" data-cs-unique="dynamicVariablesNameList" data-cs-focus required>
```

Arguments

Parameters	Definition	Data Type
data-ng-model (Required)	Used to display data in the widget.	string
data-cs-unique (Required)	Used to specify the list of data in which uniqueness needs to be checked.	`Array<Object>`

Example

Example of the Variable Name field using the unique directive, indicating that the value specified in this field must be unique:



Filters

This topic provides information about the various filters that you can use while developing widgets.

Default Angular Filters

Overview

Angular filters are utilized to format data before it is displayed on the UI. They can transform data of different types, such as strings, numbers, and dates. Angular offers a variety of built-in filters, and you can also create custom filters.

To use a filter, you must inject it into your controller. Then, you can apply it in an expression or directive by using the pipe character (|).

Reference: filter

Decode URI Filter

Overview

The Decode URI filter retrieves the UUID from the IRI:
`$filter('getEndPathName')(iri)`
IRI: `/api/3/alerts/80addd07-8636-42b0-bb87-e1817f01c48`
Result: `80addd07-8636-42b0-bb87-e1817f01c48`

Get Module Type Filter

Overview

The Get Module Type filter retrieves the module type from the IRI:
`$filter('getModuleTypeOfIri')(iri)`
IRI: `/api/3/alerts/80addd07-8636-42b0-bb87-e1817f01c48`
Result: `alerts`

Valid IRI Filter

Overview

The Valid IRI filter checks the validity of the IRI:
`$filter('isValidIRI')(iri)`
IRI: `/api/3/alerts/80addd07-8636-42b0-bb87-e1817f01c48`
Result: `true`

Truncate Text Filter

Overview

The Truncate Text filter truncates the text when its length exceeds 55 characters:
`{{'This is sample test' | truncateText}}`
Result: `This is sample...`

Angular JS events and the FortiSOAR Web Socket service

Angular JS Events

AngularJS provides the `$on`, `$emit`, and `$broadcast` services for event-based communication between controllers.

- `$emit` dispatches an event name upwards through the scope hierarchy and notifies the registered `$rootScope.Scope` listeners:
  `$scope.$emit('eventName', { message: msg });`
  For example to trigger the `fieldChange` event when there is a change in a field use:
  `$scope.$emit('fieldChange', $scope.field);`
- `$broadcast` dispatches an event name downward to all the child scopes in the hierarchy (child scopes and their children) and notifies the registered `$rootScope.Scope` listeners:
  `$scope.$broadcast('eventName', { message: msg });`
  For example to broadcast the `updateConfigurationFields` event when there is an update in a

configuration field use:
`$scope.$broadcast('updateConfigurationFields', $scope.field);`

- `$on` listens to events of a given type. It catches events dispatched by `$broadcast` and `$emit`.
  For example to activate a listener for the `fieldChange` event to listen to an emitted event and perform necessary actions, use:
  `scope.$on('fieldChange', function(field) {});`
  For example to activate a listener for the `updateConfigurationFields` event to listen to a broadcasted event and perform necessary actions, use:
  `self.$scope.$on('updateConfigurationFields', self.saveFields);`

FortiSOAR Web Socket Service

The FortiSOAR Web Socket Service allows users to create a channel between the UI and the database to fetch and update real-time data.

The following function is commonly used to listen to the `websocket:reconnect` event that is broadcasted when a socket is open for connection:

```
scope.$on('websocket:reconnect', function () {
       initWebsocket();
       }
);
```

Once the socket is open, users can subscribe to the socket in the widget using the `websocketService.subscribe()` function.

The following sample `initWebsocket()` method gets called when a widget is initialized and to subscribe to the websocket:reconnect:

```
var subscription;
 function initWebsocket(){
       websocketService.subscribe('runningworkflow',function(data){
               }).then(function(data){
                       subscription = data;
               });
 }
```

Once the widget lifecycle is closed, users must unsubscribe from the web socket using the `websocketService.unsubscribe()` function:

```
$scope.$on('$destroy', function() {
       if(subscription){
               // Unsubscribe
       websocketService.unsubscribe(subscription);
       }
});
```

More information

This section includes:

- Appendix A: Tutorials on page 52
- Appendix B: Frequently asked questions (FAQs) on page 66

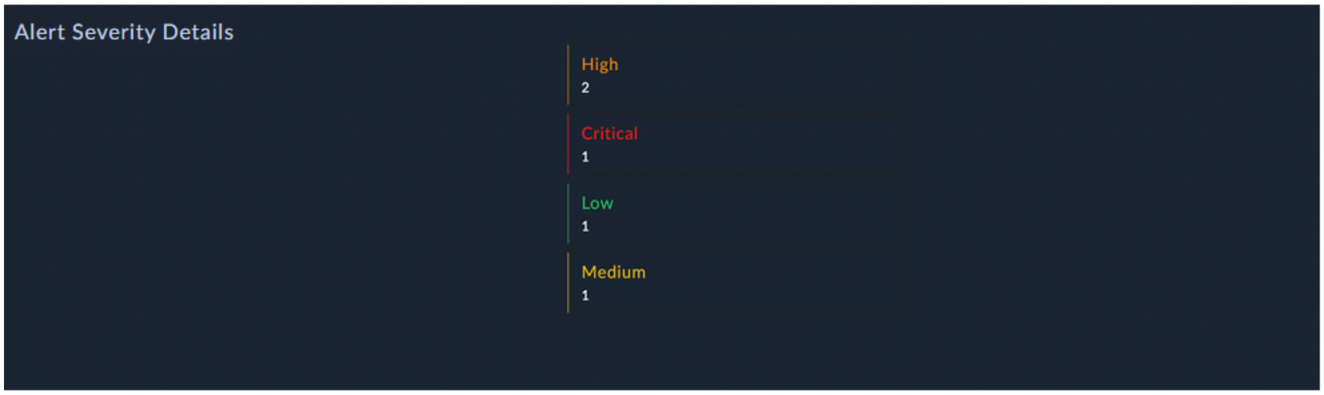
Appendix A: Tutorials

This topic provides a step-by-step guide to developing widgets, including steps on how to localize widgets, etc.

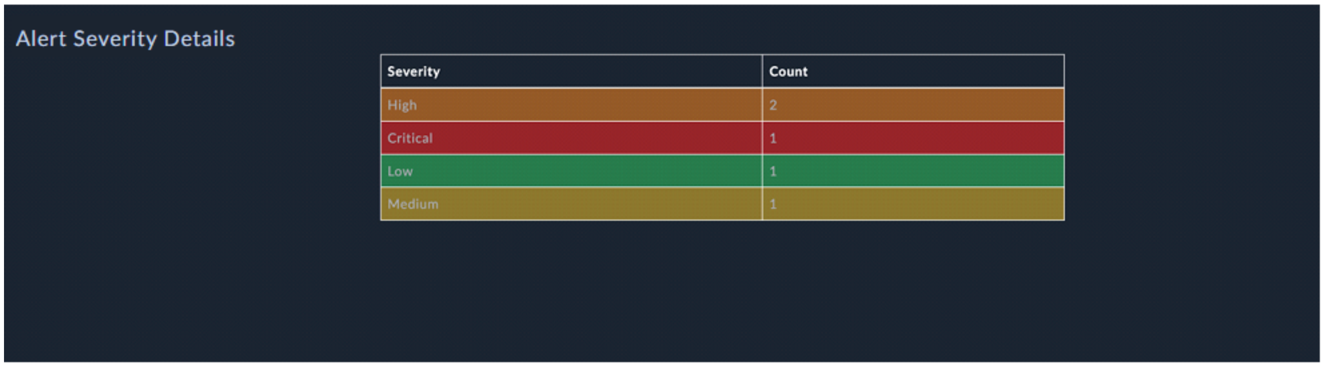
Sample widget to group records in a module by severity

The requirement of the sample widget is to first group a module's records for example, the 'Alerts' module, according to severity and then display the total number of records in each severity in various views or formats such as the List, Table, or Stacked views. Following is a sample screenshot of how this widget will be displayed in various views, i.e., the List, Tabular, and Stacked Bar Views:

List View-

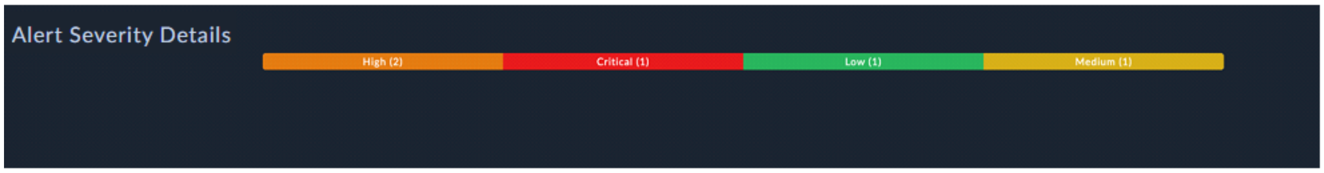


Tabular View-


Alert Severity Details

Severity	Count
High	2
Critical	1
Low	1
Medium	1

Stacked Bar View-



To create this widget, log into FortiSOAR and click on **Content Hub**. Next, click the **Create** tab and select **New Widget** from the **Create** drop-down list. This opens the 'Widget Building Wizard'. Enter the appropriate details for the widget such as, the name and title of the widget (Severity by Group), the pages on which you want to display the widget (Dashboard), and click **Create Widget**. For details on the 'Widget Building Wizard', see the Create and Use Widgets topic in the "User Guide" in the FortiSOAR Product Documentation.

The values entered during this initial configuration are used to generate the widget's `info.json` file. Additionally, the following default files are added with the `info.json` files:

- `view.html`
- `edit.html`
- `view.controller.js`
- `edit.controller.js`

- `widgetUtility.service.js` (from release 7.5.0 onwards). For more information, see the Creating a widget with multilingual support on page 62 topic.
- `locales` folder, containing files for various locales (languages), such as `en.json`, `ko.json`, etc. (from release 7.5.0 onwards). For more information, see the Creating a widget with multilingual support on page 62 topic.

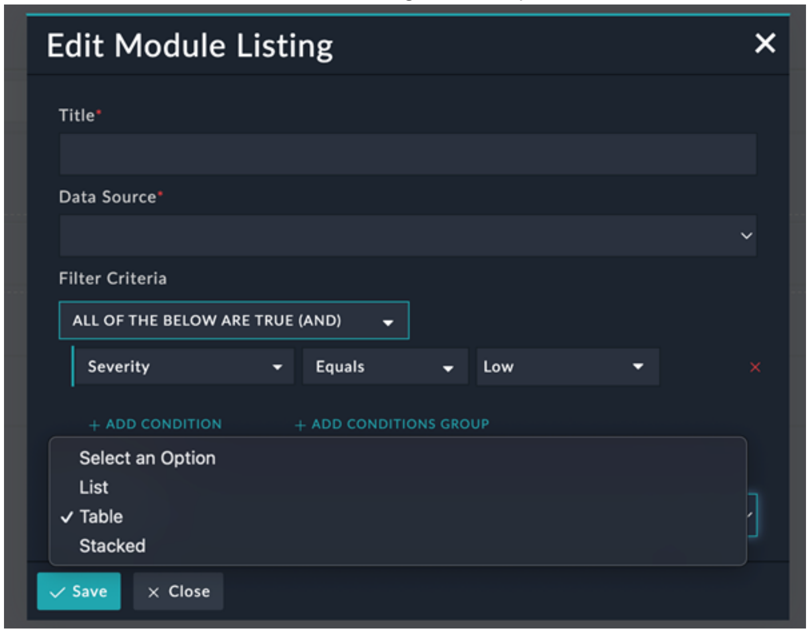
The contents of an info.json file are explained in the Overview on page 6 chapter.

Sample `info.json` file for the 'Severity by Group' widget:

```
{
    "name": "severityByGroup",
    "title": "Severity By Group",
    "subTitle": "Module's Severity By Group",
    "version": "1.0.0",
    "publishedDate": 1696317032,
    "releaseNotes": "unavailable",
    "metadata": {
        "description": "Test Description",
        "pages": [
            "Dashboard"
        ],
        "certified": "No",
        "publisher": "",
        "compatibility": [
            "7.4.3", "7.4.2"
        ]
    },
    "development": true
}
```

Widget Development Steps

The first step in widget development is to determine the required inputs, which are the inputs that need to be provided by the user to display the widget. These inputs are set in the `edit.html` file, which contains HTML tags and DOM elements. The `edit.html` file can contain elements such as, input boxes, drop-downs, checkboxes, radio buttons, and icons. Following is a sample screenshot of the editable view of the widget form:



To develop this form, you need to update the default files, i.e., the `edit.html`, `view.html`, `edit.controller`, and `view.controller` as described in the following sections.

Editing the `edit.html` file

Title: To create an input field for users to enter the widget title, use the following code snippet:

```
<div class="form-group" data-ng-class="{ 'has-error': editWidgetForm.title.$invalid &&
editWidgetForm.title.$touched }">
    <label for="title" class="control-label">Title<span class="text-danger">*</span></label>
    <input id="title" name="title" type="text" class="form-control" data-ng-model="config.title"
required>
    <div data-cs-messages="editWidgetForm.title">
    </div>
</div>
```

The value entered by users is assigned to `config.title`, which is used in the view.html.

Data Source list: To create a Data Source list for the user to select the module whose data the widget needs to displayed, use the following code snippet:

```
<div class="form-group"
    data-ng-class="{ 'has-error': editWidgetForm.resource.$invalid &&
editWidgetForm.resource.$touched }"
    data-ng-if="modules">
    <label for="resource" class="control-label">Data Source<span class="text-
danger">*</span></label>
    <select name="resource" id="resource" class="form-control"
        data-ng-options="module.type as module.name for module in modules" data-ng-
model="config.resource"
        required data-ng-change="loadAttributes()">
        <option value="">Select an Option</option>
    </select>
    <div data-cs-messages="editWidgetForm.resource"></div>
</div>
```

To load the all the modules from the application to the widget and populate the module-specific fields in the **Filter** section, once users select a module, use functions added to the edit.controller file.

Filter Criteria: Once users select a module from the **Data Source** field, module-specific fields are displayed in the 'Filter Criteria':

```
<div class="form-group" data-ng-if="config.resource">
    <label for="resource" class="control-label">Filter Criteria</label>
    <div data-cs-conditional data-mode="'queryFilters'" data-enable-nested-
filter="enableNestedFilter"
        data-fields="params.fields" data-ng-model="config.filters" data-form-
name="'editWidgetForm'"
        data-parent-form="editWidgetForm" data-reset-field="params.fields"></div>
</div>
```

Widget View: To create a list of 'Views' (List, Table, or Stacked) for the user to select how the data should be rendered, use the following code snippet:

```
<div class="form-group"
    data-ng-class="{ 'has-error': editWidgetForm.widgetView.$invalid &&
editWidgetForm.widgetView.$touched }">
    <label for="widgetView" class="control-label">Widget View<span class="text-
danger">*</span></label>
    <select name="widgetView" id="widgetView" class="form-control"
        data-ng-options="view as view for view in widgetView" data-ng-model="config.widgetView"
required>
        <option value="">Select an Option</option>
    </select>
    <div data-cs-messages="editWidgetForm.widgetView"></div>
</div>
```

Editing the `edit.controller` file

Populating the **Data Source** field requires the list of modules. To load all the modules from the application to the widget, use the `loadModules` function in the `edit.controller` file:

```
function loadModules() {
       appModulesService.load(true).then(function (modules) {
               $scope.modules = modules;
       });
}
```

Once users select a module from the Data Source field, module-specific fields need to be populated in the **Filter** section:

```
function loadAttributes() {
    var entity = new Entity($scope.config.resource);
    entity.loadFields().then(function () {
        $scope.params.fields = entity.getFormFields();
        angular.extend($scope.params.fields, entity.getRelationshipFields());
        $scope.params.fieldsArray = entity.getFormFieldsArray();
    });
}
```

To populate the **Widget View** field, use the following code snippet:

```
$scope.widgetView = ['List', 'Table', 'Stacked'];
```

Editing the view.controller file

To fetch data as per the configuration provided by the user in the widget form, use the following code snippet:

```
function severityByGroup100DevCtrl($scope, $http, Query, API, $q, config) {
    $scope.config = config;

    function fetchData() {
        getResourceAggregate().then(function (result) {
            if (result && result['hydra:member'] && result['hydra:member'].length > 0) {
                $scope.alertSources = result['hydra:member'].filter(function (item) {
                    return item.severity !== null;
                });
                if ($scope.config.widgetView === 'Stacked') {
                    var totalCount = $scope.alertSources.reduce(function (a, b) {
                        return { total: a.total + b.total };
                    });
                    $scope.alertSources.forEach(element => {
                        var _width = (element.total) / totalCount.total * 100;
                        element.progressBarWidth = _width;
                    });
                }
            }
        });
    }

    function getResourceAggregate() {
        var defer = $q.defer();
        var queryObject = {
            sort: [{
                field: 'total',
                direction: 'DESC'
            }],
            aggregates: [
                {
                    'operator': 'countdistinct',
                    'field': '*',
                    'alias': 'total'
                },
                {
                    'operator': 'groupby',
                    'alias': 'severity',
                    'field': 'severity.itemValue'
                },
                {
                    'operator': 'groupby',
                    'alias': 'color',
                    'field': 'severity.color'
                }
            ],
            filters: [$scope.config.filters]
        };
        var _queryObj = new Query(queryObject);
        $http.post(API.QUERY + $scope.config.resource, _queryObj.getQuery(true)).then(function
(response) {
            defer.resolve(response.data);
        }, function (error) {
            defer.reject(error);
        });

        return defer.promise;
    }

    fetchData();
}
```

The 'Query' object contains the query parameters along with the config filters to fetch data.

Editing the `view.html` file

The `View.html` file contains HTML tags and DOM elements according to the widget's UX. It consists of divs, images, svgs, or Angular libraries such as charts.

Title: To render the input field where the user can enter the title of the widget, use the following code snippet:

```
<div class="padding-right-0 padding-left-0 widget-dashboard-title-width"
    data-ng-class="(page === 'dashboard' || page === 'reporting') ? 'widget-dashboard-title-width'
: 'widget-title-width'">
    <h5 class="margin-top-0 margin-bottom-0 text-overflow ng-binding">{{ config.title }}</h5>
</div>
```

`config.title` renders the value as the title on the widget.

Widget View: To render data in the widget as per the user's selection, i.e., in the List, Table, or Stacked views, use the following code snippet:

```
<div data-ng-hide="collapsed">
    <div class="margin-left-15" ng-if="config.widgetView === 'List'">
        <div class="container width-30">
            <div class="list-group">
                <div class="list-group-item margin-bottom-md" style="border-left: 1px solid
{{element.color}};"
                    data-ng-repeat="element in alertSources">
                    <h4 class="list-group-item-heading" style="color: {{element.color}};">
```

```
{{element.severity}} </h4>
                    <p class="list-group-item-text">{{element.total}}</p>
                </div>
            </div>
        </div>
    </div>
    <div class="margin-left-15" ng-if="config.widgetView === 'Table'">
        <div class="container width-50">
            <table class="table">
                <thead>
                    <tr>
                        <th>Severity</th>
                        <th>Count</th>
                    </tr>
                </thead>
                <tbody>
                    <tr data-ng-repeat="element in alertSources" data-ng-style="{'background-
color':element.color}">
                        <td>{{element.severity}}</td>
                        <td>{{element.total}}</td>
                    </tr>
                </tbody>
            </table>
        </div>
    </div>
    <div class="margin-left-15" ng-if="config.widgetView === 'Stacked'">
        <div class="container">
            <div class="progress">
                <div data-ng-repeat="element in alertSources" class="progress-bar"
role="progressbar"
                    style="width:{{element.progressBarWidth}}%;background-color:
{{element.color}};">
                    {{element.severity}} ({{element.total}})
                </div>
            </div>
        </div>
    </div>
</div>
```

If you want to move the service logic to a separate file, you can create a `<name>.service.js` file in the widget 'asset' directory, and inject the service in the controller files to utilize its methods.

Creating a custom non-modal widget with an interactive background

You can create a non-modal widget that is accessible across the FortiSOAR application.

Widgets that are launched using a button and do not need to be configured to work should be made non-modal.

To create a non-modal widget, you must add the following to the widget's `info.json` file, save the changes, and publish the widget:

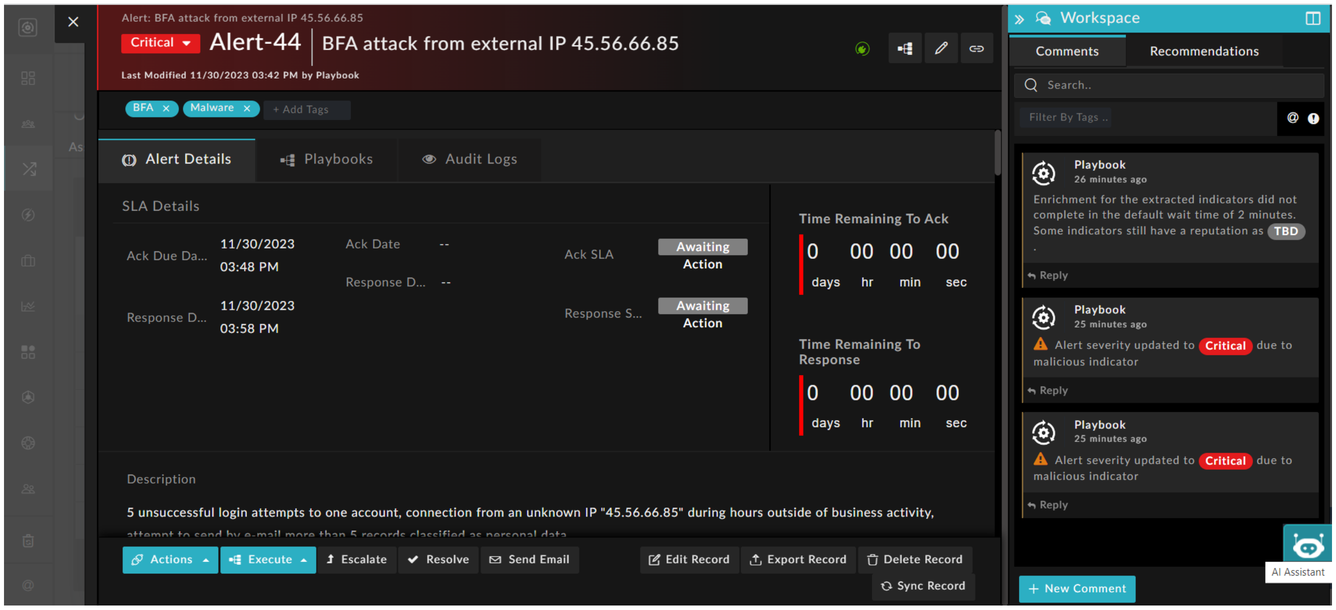
```
"contexts": [
        "drawer"
]
```

Widgets can be enabled for multiple specific contexts, such as 'drawer' and 'playbook designer':

```
"contexts": [
        "drawer",
        "pb_designer"
]
```

In this case, the widget's logo is accessible from anywhere in FortiSOAR when using the '`drawer`' context and as an icon in the playbook designer. The 'drawer' and the icon in the 'playbook designer' can both be used to launch the widget.

When you install a widget with a 'drawer' context, it will show up in a drawer with the display name ('AI Assitant' in the following image) and logo that match the details you provided when the widget was developed. A default widget icon is shown if the logo is not specified.



Alert: BFA attack from external IP 45.56.66.85

Alert-44 BFA attack from external IP 45.56.66.85

Last Modified 11/30/2023 03:42 PM by Playbook

SLA Details

Ack Due Da... 11/30/2023 03:48 PM
Ack Date --
Response D... --
Ack SLA Awaiting Action
Response D... 11/30/2023 03:58 PM
Response S... Awaiting Action

Time Remaining To Ack: 0 days 00 hr 00 min 00 sec

Time Remaining To Response: 0 days 00 hr 00 min 00 sec

Description

5 unsuccessful login attempts to one account, connection from an unknown IP "45.56.66.85" during hours outside of business activity, attempt to send by e-mail more than 5 records classified as personal data

Playbook, 26 minutes ago: Enrichment for the extracted indicators did not complete in the default wait time of 2 minutes. Some indicators still have a reputation as TBD .

Playbook, 25 minutes ago: Alert severity updated to Critical due to malicious indicator

Playbook, 25 minutes ago: Alert severity updated to Critical due to malicious indicator

When users click the logo, the widget is launched as per the configuration provided when the widget was created. Therefore, it is imperative to make non-modal just those widgets that do not require configuration to function.

You can also customize how the widget is launched and viewed by adding properties in the `view` section of the `info.json`. For example, you can choose to launch a widget with an interactive background. Having an interactive background enables users to optimally perform their tasks in their current context.
The various properties that can be added to the `view` section are:

- To assign a name to a widget, use the `"displayName"` property. For example, if you want to name your widget 'FortiAI', add `"displayName" : "FortiAI"`.
  If you do not provide a `"displayName"` for the widget, it will be displayed with only its logo.

- To have a fully functional background when a widget is opened, add `"popup": custom`. If you do not specify the `popup` property or set to anything other than 'custom', the widget will remain modal, meaning it will have a non-interactive background.
- To make the widget draggable, add `"draggable": true`

  Only widgets that have an interactive background, i.e., those that have `"popup": custom` set, should be made draggable. If '`popup`' is not added in the 'view' section or if it is set to anything except 'custom', then the previous version of the FortiSOAR widget is launched with a blurred background. Since FortiSOAR widgets were not draggable in previous iterations, this property will be ignored.
  Also note that the '`popup`' and '`draggable`' properties are ignored if a widget is not launched in the '`drawer`'.

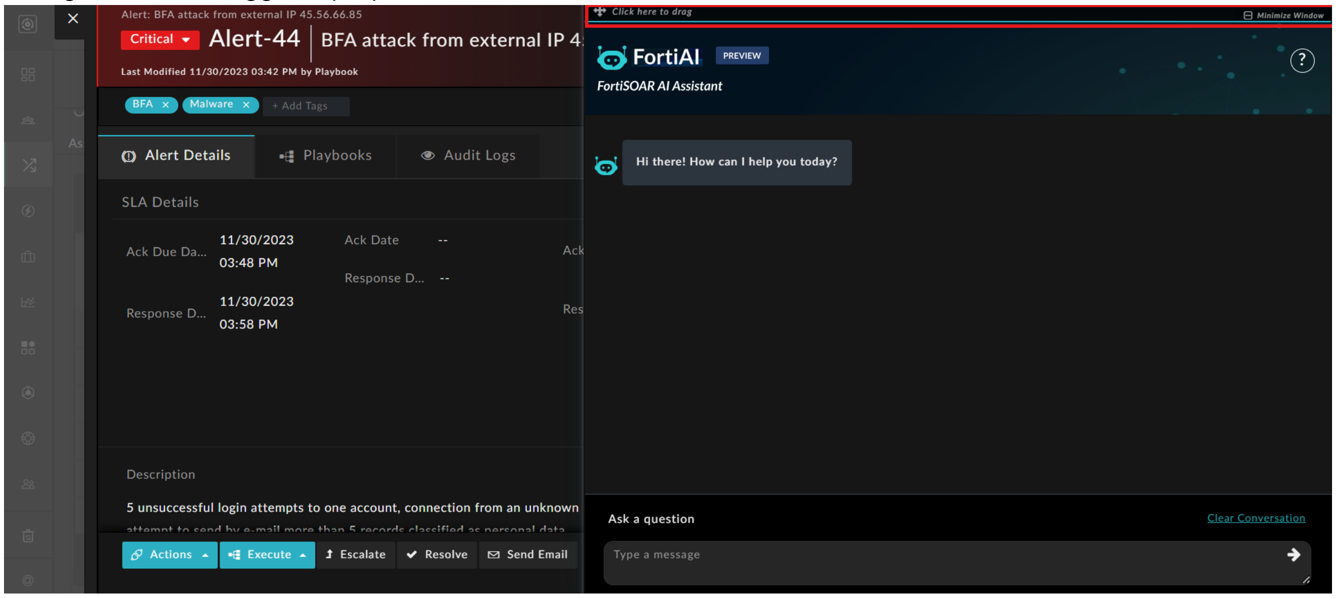
- To enable the widget to be visible as a 'drawer' on specific pages in FortiSOAR, use the `enableFor` property:
  ```
  "enableFor": [
     "main.playbookDetail",
     "main.modules.list",
     "viewPanel.modulesDetail",
     "main.dashboard"
  ]
  ```
  This will enable the widget in a drawer in the playbook detail page, dashboards page, and the listings and details pages for all the modules in FortiSOAR.

Once you add the required properties to the `view` section of the `info.json` file, save the file and publish the widget.

Following is an example of adding properties in the `info.json` file to enable a widget named "FortiAI" in a 'drawer' on the playbook designer and detail page of the modules, with an interactive background and draggable:

```
         "contexts": [
                "drawer"
        ],
        "view": {
                "popup": "custom",
                "draggable": true,
                "displayName": "FortiAI",
                "enableFor": [
                        "viewPanel.modulesDetail",
                        "main.playbookDetail"
                ]
        },
```

Following is an example of how a widget is displayed in the detail view of an alert record with its interactive background and draggable properties enabled:


Alert: BFA attack from external IP 45.56.66.85

Critical Alert-44 BFA attack from external IP 4

Last Modified 11/30/2023 03:42 PM by Playbook

BFA Malware + Add Tags

Alert Details Playbooks Audit Logs

SLA Details

Ack Due Da... 11/30/2023 03:48 PM Ack Date -- Response D... --

Response D... 11/30/2023 03:58 PM

Description

5 unsuccessful login attempts to one account, connection from an unknown attempt to send by e-mail more than 5 records classified as personal data

Click here to drag

FortiAI PREVIEW FortiSOAR AI Assistant

Hi there! How can I help you today?

Ask a question Clear Conversation

Type a message

Creating a widget with multilingual support

In FortiSOAR release 7.5.0, support has been added to support multiple languages. Administrators and users can now choose the language in which they want their FortiSOAR instance to be displayed by changing the language settings in the '**General**' tab of the 'System Configuration' page and 'User Profile' page. For more information, see the System Configuration and Management chapter in the "Administration Guide" and the User Profile topic in the *Manage User Access and Profiles* chapter of the "User Guide."
NOTE: Preview of a widget will not work on function on releases before 7.4.1 if your widget is localized.

To enable multilingual support, starting from release 7.5.0 the `widgetUtility.service.js` file is automatically included in the `widgetAssets/js` folder, and the JSON files used for language translation are added to the `widgetAssets/locales` folder, when creating a widget. For the directory structure of a widget, see the Development process chapter.

The current release supports the following languages:

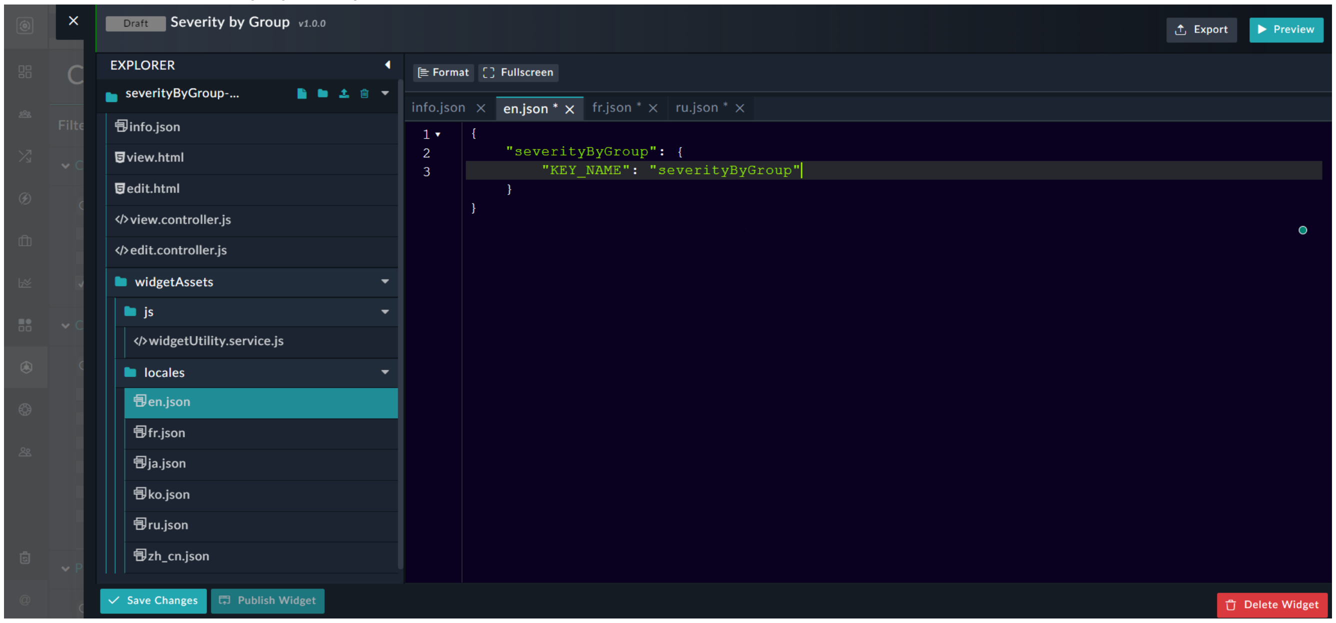
Language	Name of the locale file
English	en.json
Japanese	ja.json
Korean	ko.json
Simplified Chinese	zh_ch.json
French (added in FortiSOAR release 7.6.4)	fr_fr.json
Traditional Chinese (added in FortiSOAR release 7.6.4)	zh_tw.json

Each locale file follows a key-value structure, with the translation key and the corresponding value representing the translated text that will be used in the widget.

For example, let's consider the 'Severity by Group' widget. The widget's name (API key) is "severityByGroup" , so the `en.json` file contains this data in the following format:

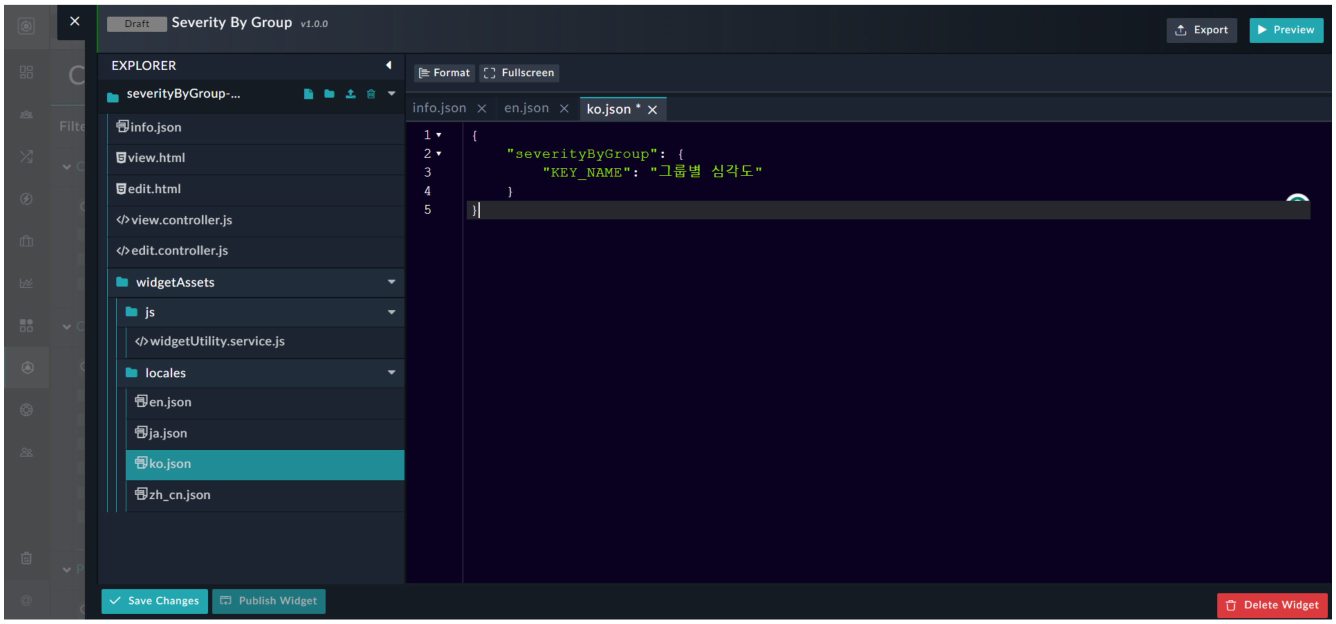
```
{
    "severityByGroup": {
        "KEY_NAME": ""
    }
}
```

You can add "severityByGroup" as its `KEY NAME`:



```
{
    "severityByGroup": {
        "KEY_NAME": "severityByGroup"
    }
}
```

Similarly, you can edit the `ko.json` file to add the value of the "severityByGroup" API key in Korean:



```
{
    "severityByGroup": {
        "KEY_NAME": "그룹별 심각도"
    }
}
```

You can edit all the JSON files in a similar manner to translate all the keys of the widget into the required language.

The `widgetUtility.service.js` is responsible for handling translations in widgets and includes three main functions: `checkTranslationMode`, `translate`, and `getWidgetNameVersion`.

The `checkTranslationMode` function checks if the translation service is included, and if not, injects the translation service into the widget.

```
function checkTranslationMode(widgetName) {
  widgetNameVersion = widgetName;
  try {
    translationService = $injector.get('translationService');
  } catch (error) {
    console.log('"translationService" doesn\'t exists');
  }
  var defer = $q.defer();
  translationServiceExists = typeof translationService !== 'undefined';
  if (!translationServiceExists) {
    var WIDGET_BASE_PATH;
    try {
      WIDGET_BASE_PATH = $injector.get('WIDGET_BASE_PATH');
    } catch (e) {
      WIDGET_BASE_PATH = {
        INSTALLED: 'widgets/installed/'
      };
    }
    $http.get(WIDGET_BASE_PATH.INSTALLED + widgetNameVersion +
'/widgetAssets/locales/en.json').then(function(enTranslation) {
      translationData = enTranslation.data;
      defer.resolve();
    }, function(error) {
      console.log('English translation for widget doesn\'t exists');
      defer.reject(error);
    });
  } else {
    defer.resolve();
  }
  return defer.promise;
}
```

The `translate` function returns the translated value of the key passed as a parameter.

```
function translate(KEY, params) {
  if (translationServiceExists) {
       return translationService.instantTranslate(KEY, params);
  } else {
    var translationValue = angular.copy(translationData);
    var keys = KEY.split('.');

    for (var i = 0; i < keys.length; i++) {
      if (translationValue.hasOwnProperty(keys[i])) {
        translationValue = translationValue[keys[i]];
      } else {
        translationValue = '';
        break;
      }
    }
    if (params) {
      return $interpolate(translationValue)(params);
    }
    return translationValue;
  }
}
```

The `getWidgetNameVersion` function provides the name and version of the widget to the `edit.controller.js`:

```
function getWidgetNameVersion(widget, widgetBasePath) {
   let widgetNameVersion;
   if (widget) {
     widgetNameVersion = widget.name + '-' + widget.version;
   } else if (widgetBasePath) {
     let pathData = widgetBasePath.split('/');
     widgetNameVersion = pathData[pathData.length - 1];
   } else {
     toaster.warning({
       body:'Preview is unavailable for widgets that support localization.'
     });
   }
   return widgetNameVersion;
 }
```

In the `edit.controller.js` file, edit the `_handleTranslations` function to evaluate the translation keys. This function checks the translation mode from the `widgetUtility` service and renders the translated key, `{"severityByGroup": {"KEY_NAME": ""}}` in our example, on FortiSOAR:

```
function _handleTranslations() {
  let widgetNameVersion = widgetUtilityService.getWidgetNameVersion($scope.$resolve.widget,
$scope.$resolve.widgetBasePath);

  if (widgetNameVersion) {

    widgetUtilityService.checkTranslationMode(widgetNameVersion).then(function () {
      $scope.viewWidgetVars = {
        // Create your translating static string variables here
              module_key: widgetUtilityService.translate("severityByGroup.KEY_NAME")
      };
    });

  } else {
    $timeout(function() {
      $scope.cancel();
    });
  }
}
```

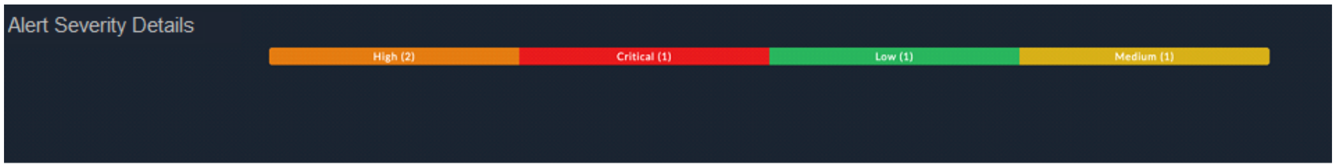
In the `view.controller.js` file, edit the `_handleTranslations` function to evaluate the translation keys.

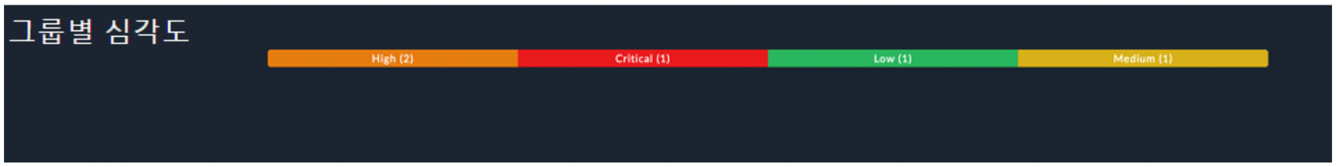
```
function _handleTranslations() {
  widgetUtilityService.checkTranslationMode($scope.$parent.model.type).then(function () {
    $scope.viewWidgetVars = {
      // Create your translating static string variables here
       module_key: widgetUtilityService.translate("severityByGroup.KEY_NAME")
    };
  });
}
```

In the `edit.html` and `view.html` files, define how you want to display the translated content:

```
<div class="modal-body">
    <div>
          Module Name: {{ viewWidgetVars.module_key }}
  </div>
</div>
```

The `_handleTranslations()` function checks the translation mode from the `widgetUtility` service and renders the "`severityByGroup.KEY_NAME`" on FortiSOAR once you save and publish the widget:





Appendix B: Frequently asked questions (FAQs)

This topic contains various FAQs to help users quickly get answers to some commonly-asked questions.

Can an external css/js file plus CDN assets be added while creating a widgets?

Certainly! You can add external JavaScript, CSS, and HTML files and CDN assets when creating a widget. To do this, organize the external files (JavaScript, CSS, and HTML) into a custom folder structure in your widget, as follows:

1. Add a custom folder structure in your widget folder. For example:

```
/app
  /widgets
       /myWidget
               /js
                       - widgetScript.js
               /css
                       - widgetStyle.css
               /html
                       - widgetTemplate.html
  /views
       - view.html
```

2. Add the .js files of the external library to the `/js` folder in the widget folder and then reference the external files in the `view.html` file of your widget:

```
<!-- Include Widget JavaScript file -->
<script src="widgets/installed/mywidget/js/widgetScript.js"></script>

<!-- Include Widget CSS file -->
<link rel="stylesheet" href="widgets/installed/myWidget/css/widgetStyle.css">

<!-- Include Widget HTML template -->
<div ng-include="'widgets/installed/myWidget/html/widgetTemplate.html'"></div>
```

NOTE: Ensure that you specify the correct path based on your project structure.

Alternatively, if you do not want to add the external JS files, you can load the external JS file using CDN injection (`promise`), when a third-party CDN is required. This code waits for the libraries to load before proceeding with the rest of the widget code:

```
function loadJsAsync(jsPath) {
    var deferred = $q.defer();
    var script = document.createElement("script");
    script.type = "text/javascript";
    script.src = jsPath;
    script.onload = function () { deferred.resolve(); };
    script.onerror = function () { deferred.reject("Failed to load script: " + jsPath); };
    document.getElementsByTagName("head")[0].appendChild(script);
    return deferred.promise;
}
const scriptsToLoad = [
         'https://unpkg.com/@hpcc-js/wasm@0.3.11/dist/index.min.js',
         'https://unpkg.com/d3-graphviz@3.0.5/build/d3-graphviz.js'
     ];

     await $q.all(scriptsToLoad.map(loadJsAsync));
```

Additionally, you can use the timeout function to ensure that the file from CDN will be loaded before rendering the widget functionality:

```
$timeout(function () {
    loadJs('https://cdnjs.cloudflare.com/ajax/libs/d3-sankey/0.12.3/d3-sankey.min.js');
    fetchData();
  }, 1000);
```

In this case, the execution of the widget function is delayed by 1000 ms. The return value of calling `$timeout` is a '`promise`' that is resolved when the delay has passed and the `timeout` function, if provided, is executed.

Reference Widgets: widget-custom-picklist-message and widget-task-management.

How do I make a widget compatible with a supported theme?

To make a widget compatible with a supported theme, use `$rootScope.theme` to access the current theme in your AngularJS application. Once `$rootScope.theme` returns the current theme, you can check its value to determine if it is "light," "dark," or "space."

NOTE: When making API calls, refer to the "space" theme as "steel".

```
var theme = $rootScope.theme;
       if (theme.id === "light") {
               $scope.cardTilesThemeColor.cardBackgroundColor = "#eeeeee";
       } else if (theme.id === "steel") {
               $scope.cardTilesThemeColor.cardBackgroundColor = "#29323e";
       } else {
               $scope.cardTilesThemeColor.cardBackgroundColor = "#262626";
       }
```

Reference Widgets: widget-record-card-tiles, widget-record-summary-card

How do I display a widget on specific pages within FortiSOAR?

The `info.json` file in a widget, especially in the context of certain web applications or platforms, often serves as a configuration file that provides metadata about the widget. It contains information about the widget's name, description, version and other configuration details.

The `pages` section in the `info.json` file specifies the pages or sections where the widget should be visible. The "`pages`" section in your widget's `info.json` is a required field containing an array of pages where the widget should be displayed. For example, if you want your widget to be displayed in Dashboards and Reports, you should include them in the "`pages`" section of your widget's `info.json` file:

```
{
       "metadata": {
               "pages": [
                       "Dashboard",
                       "Reports"
               ],
               "certified": "No",
               "publisher": "Community",
               "compatibility": [
                       "7.2.0",
                       "7.0.2"
               ]
       },
       "name": "roiCalculator",
       "title": "ROI Calculator",
       "subTitle": "Uses playbook tags to display an accurate automation ROI",
       "version": "1.0.0"
}
```

Reference Widgets: widget-roi-calculator, widget-record-summary-card

How can I access the module record context for a widget written for the View Panel?

To access the module record context from the View Panel, use `$state.params.module` to access the module name and `$state.params.id` to access the record ID. It is common practice to use 'UI-Router' for routing in AngularJS applications.

```
Modules.get({
               module: $state.params.module,
               id: $state.params.id
 })
```

Reference Widgets: widget-custom-picklist-message and widget-sla-countdown-timer.

What is the process to submit a widget to Content Hub?

The process for submitting your widget is listed at: https://github.com/fortinet-fortisoar/how-tos.

Can I write a service specific to a widget?

Certainly! You can write a service specific to a widget. To do this, add the service file into a custom folder structure and then including this file in the `view.html` in your widget:

1. Add a custom folder structure in your widget folder. For example:

```
/app
 /widgets
     /myWidget
         /js
             - widgetScript.service.js
```

```
/views
   - view.html
```

2. Add the script file of the external library to the `/js` folder in the widget folder and then reference this script file in the `view.html` file of your widget:

```
<!-- Include Widget JavaScript service file -->
<script src="widgets/installed/myWidget/js/widgetScript.service.js"></script>
```

NOTE: Ensure that you specify the correct path based on your project structure.

Reference Widgets: widget-custom-picklist-message and widget-task-management.

How to find out the current context of the widget?

The current context of a widget provides you information such as whether the widget is on Dashboard or View Panel (Record details page). To find out if the current context of widget is for example, the 'View Panel', use the following code snippet:

```
isViewPanelPage = $state.current && $state.current.name.indexOf('viewPanel') !== -1;
```

www.fortinet.com