IBM Security QRadar Application Framework
April 2018

Developer Quick Start Guide
Note

Before you use this information and the product that it supports, read the information in “Notices” on page 147.
Contents

1 QRadar apps .................................................. 1

2 QRadar app development overview ........................... 3

3 GUI application framework fundamentals ................... 7

4 App file structure ............................................ 11
   Application manifest structure ................................ 12
   Source dependencies .......................................... 13
   Installing Node.js as a source dependency .................. 14

5 Manifest object types ......................................... 17
   Areas type ..................................................... 17
   REST method type .......................................... 17
   Dashboard items type ........................................ 18
   Configuration pages type .................................... 19
   GUI Action type ............................................. 20
   Page scripts type ........................................... 22
   Metadata providers type .................................... 23
   Resource bundles type ...................................... 24
   Developer options type ..................................... 24
   Resources type .............................................. 25
   Fragments type .............................................. 26
   Custom columns type ...................................... 27

6 The Hello World sample app ................................ 29
   New tab example ............................................ 31

7 QRadar App Editor ............................................. 35
   What’s new in the QRadar App Editor ....................... 35
   Installing the QRadar App Editor ......................... 35
   Starting the QRadar App Editor ......................... 36
   Editing apps in the editor ................................... 38

8 Software development kit overview .......................... 41
   Optimize app memory usage .................................. 41

9 Installing the SDK .............................................. 45

10 Use Python 2.7 in your app ................................... 47

11 Creating your development environment .................... 49
   Developing apps in Eclipse .................................. 50
   Installing Python 2.7.9 on OSX ............................ 51
   Packaging and deploying your app ......................... 51
   Running your application locally ............................ 52

12 Creating an extension from your app ....................... 53
   Adding multiple apps in an extension ..................... 53
   IBM QRadar content extensions ............................ 54
   Extensions management ..................................... 56
## 13 Sample apps
- Dashboard item example ............................................. 57
- Page script / toolbar button example ............................... 57
- Page script samples ..................................................... 60
- Passing context-specific information to a page script .......... 64
- Context-specific metadata provider example ...................... 66
- Add right-click functionality ......................................... 70
- Custom fragments example .......................................... 72
- Custom column example .............................................. 74

## 14 Support functions .................................................. 77

## 15 QRadar Python helper library functions .......................... 79

## 16 Jinja2 templates .................................................... 81
- Integrate JavaScript libraries into your template .................. 81

## 17 App Framework JavaScript library ................................. 83

## 18 Communicating with QRadar hosts from Python ............... 85

## 19 GUI Application Framework REST API endpoints .............. 87

## 20 App logs .................................................................. 89
- Adding logging to your app ............................................ 89
- Viewing your app logs ................................................. 90

## 21 Stopping, restarting, and uninstalling an app .................... 91

## 22 Accessing your app's command line ............................... 93

## 23 App upgrades .......................................................... 95

## 24 Available user role capabilities ..................................... 97

## 25 App names, GUI action groups, and page IDs ................... 99

## 26 Application globalization ............................................ 103
- Globalization of QRadar elements .................................. 103
- Globalization of application-specific content ...................... 107

## 27 Custom fragments injection points ................................. 113

## 28 Custom column injection points .................................... 119

## 29 Custom actions for CRE responses ................................. 121
- Defining custom actions .............................................. 122
- Testing your custom action .......................................... 124
- Adding a custom action script to an event rule .................. 124
- Custom action REST API endpoints ................................. 125

## 30 Custom AQL functions .............................................. 127
- Custom AQL function fields ......................................... 129
- Custom AQL function utilities ...................................... 133

---

iv IBM Security QRadar Application Framework: Developer Quick Start Guide
31 Resources .............................................. 145

Notices ..................................................... 147
  Trademarks ........................................... 148
  Terms and conditions for product documentation .................. 148
  IBM Online Privacy Statement ................................ 149
1 QRadar apps

Use IBM® QRadar® apps to extend and enhance your current QRadar deployment with new data and ready-to-use use cases.

A QRadar app is a means to augment and enrich your current QRadar system with new data and functionality. You can download and install other shared apps that are created by IBM, its Business Partners, and other QRadar customers.

You can create your own apps from QRadar by using the QRadar GUI Application Framework Software Development Kit (SDK). You can then package the app and reuse it in other QRadar deployments. You can share your app on the [IBM X-Force Exchange](https://exchange.xforce.ibmcloud.com/) portal.

Apps provide new tabs, API methods, dashboard items, pop-up menus, toolbar buttons, configuration pages, and more within the QRadar user interface. The functionality is entirely defined by Python Flask framework apps that serves the app endpoints from a secure container.

**Download public apps**

All apps and security product enhancements are hosted on the [IBM X-Force Exchange](https://exchange.xforce.ibmcloud.com/) portal.

You can see a list of available apps on the [IBM Security App Exchange](https://exchange.xforce.ibmcloud.com/hub). Filter apps by selecting the **Application** check box.

Every download from the X-Force App Exchange is known as an extension. An extension can consist of an app or security product enhancement (content extension) that is packaged as an archive (.zip) file, which you can deploy on QRadar by using the **Extensions Management** tool on the **Admin** tab.
2 QRadar app development overview

Use the IBM QRadar GUI Application Framework to develop new application modules that integrate with QRadar and provide new capabilities.

Applications or apps are small plug-in modules to the GUI Application Framework. Apps serve endpoints from within a secure container to inject the content directly into the QRadar web interface.

Each app has its own dedicated memory allocation and a defined amount of CPU resources that are allocated to it.

The main web language that is used to author an application is Python, and the Flask framework is integrated and available for use by the application.

How an application runs and interacts with QRadar

QRadar applications run inside an isolated Python Flask environment that is independent of the QRadar user interface.

The application can also use static images, scripts, and HTML pages.

All interaction with the application is proxied through the QRadar user interface. No direct access to network ports or web services is usually permitted.
Apps that require internet access

If the app that you develop requires internet access, you must implement proxy support in your app. Apps can't use the proxy support that is built into QRadar.

Types of app

The QRadar GUI Application Framework supports the following app types that are described in the following table.

Table 1. Types of app

<table>
<thead>
<tr>
<th>App Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Areas (or visualizations)</td>
<td>New screen that is presented in a new tab.</td>
</tr>
</tbody>
</table>
Table 1. Types of app (continued)

<table>
<thead>
<tr>
<th>App Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right-click menus</td>
<td>New right click menu options available with the QRadar GUI.</td>
</tr>
<tr>
<td>Toolbar buttons</td>
<td>New toolbar buttons, with the enabling code that runs from the confines of the app.</td>
</tr>
<tr>
<td>Dashboard/Dashboard widgets</td>
<td>New dashboard widgets, with the HTML served from a particular app.</td>
</tr>
<tr>
<td>Administrative screens</td>
<td>New Admin tab, configuration, and setup screens.</td>
</tr>
<tr>
<td>Hover Over metadata</td>
<td>Injection of hover over metadata into existing hover over areas.</td>
</tr>
<tr>
<td>JavaScript page scripts</td>
<td>Injected browser JavaScript functionality specific to an existing QRadar GUI screen area.</td>
</tr>
<tr>
<td>Resource Bundles</td>
<td>Partial support of Java style key value pair properties files to provide some level of globalization support.</td>
</tr>
<tr>
<td>Custom fragments</td>
<td>Inject custom HTML fragments into the QRadar UI.</td>
</tr>
<tr>
<td>Custom columns</td>
<td>Add columns with custom content to tables in the QRadar</td>
</tr>
</tbody>
</table>

The app type content is dynamically injected back into the GUI display.

Apps are packaged as compressed archives (.zip), within the extension archive. You can install and uninstall apps by using RESTful endpoints. More RESTful endpoints exist to control the lifecycle of an app within QRadar.

Note: As a best practice, store your app configuration and data in /store because data in this directory is protected during app upgrades.

For more information about QRadar application framework REST API endpoints, see 19, “GUI Application Framework REST API endpoints,” on page 87.
3 GUI application framework fundamentals

QRadar GUI application framework apps are stand-alone web applications that run on the Flask micro-framework, and are served from the Flask web server.

Installation overview

Every app runs in its own unique Flask server. Each Flask server, in turn, runs within a secure Linux container. Docker is the implementation stack for the secure containment of the Flask app codebase.

Each app is installed by using the RESTful API endpoints. The installation endpoint handles these tasks:
- Validates the manifest of the app.
- Automatically creates a Docker image (asynchronous) with the app code that is bundled within it.
- Registers the app (asynchronous) with QRadar to enable web traffic proxy and the HTTP request/response lifecycle from QRadar to the app.
- Automatically runs a Docker container from the Docker image (asynchronous), which is bound to a data-only secondary container that is used for persistent storage.

QRadar RESTful API endpoints

The key interface between lifecycle management of an app, during both its creation and running phases, is the QRadar GUI App Framework REST API endpoints.

The following table describes the QRadar RESTful API endpoints.

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /gui_app_framework/application_creation_task</td>
<td>Application ID</td>
<td>Retrieves a list of status details for all asynchronous requests to create apps.</td>
</tr>
<tr>
<td>GET /gui_app_framework/application_creation_task/{application_id}</td>
<td>Application ID</td>
<td>Retrieves a list of status details of an asynchronous request to create apps.</td>
</tr>
<tr>
<td>POST /gui_app_framework/application_creation_task</td>
<td>Application (.zip) bundle file</td>
<td>Creates an app within the application framework, and registers it with QRadar. The app is created asynchronously. A reference to the <code>application_id</code> is returned and must be used in subsequent API calls to determine the status of the app installation.</td>
</tr>
<tr>
<td>POST /gui_app_framework/application_creation_task/{application_id}</td>
<td>Application ID, cancel status</td>
<td>Updates a new app installation within the application framework. The <code>application_id</code> and a <code>status</code> parameters are required.</td>
</tr>
<tr>
<td>GET /gui_app_framework/applications</td>
<td></td>
<td>Retrieves a list of apps that are installed on the QRadar console, and their manifest JSON structures and status.</td>
</tr>
<tr>
<td>GET /gui_app_framework/applications/{application_id}</td>
<td>Application ID</td>
<td>Retrieves a specific app that is installed on the console and its manifest JSON structure and status.</td>
</tr>
</tbody>
</table>
Table 2. GUI Application Framework REST API endpoints (continued)

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /gui_app_framework/applications/{application_id}</td>
<td>Application ID, start/stop status</td>
<td>Updates an app. Starts or stops an app by setting status to RUNNING or STOPPED respectively.</td>
</tr>
<tr>
<td>PUT /gui_app_framework/applications/{application_id}</td>
<td>Application ID</td>
<td>Upgrade an application.</td>
</tr>
<tr>
<td>DELETE /gui_app_framework/applications/{application_id}</td>
<td>Application ID</td>
<td>Deletes an application.</td>
</tr>
</tbody>
</table>

**Python**

The main web language that is used to author an app is Python, and the Flask framework is integrated and available for use by the app.

For more information, go to the [Python](https://www.python.org/doc/) website.

**Flask**

Flask is a micro web application framework that is written in Python.

Flask is the web server from which the app-coded endpoints are served. You use Python functions to deliver use cases. You can use route annotations for each Python method in the Flask application. After the Flask web server starts, HTTP/HTTPS-bound requests are serviced by Flask for that route, and the Python functions are run.

Each Flask server that is run from within the Docker container uses port 5000. Outwardly from the container, Docker maps that internal port 5000 to the next free port from the 49152-65535 ephemeral range. During the registration phase, this outward mapped port is stored by QRadar so that web requests for an app, through QRadar are proxied to the correct container.

The following code is a sample Python route:

```python
@app.route('/

def hello_world():
    return 'Hello World!'
```

In a standalone Flask web server, a web request through a browser to http://localhost:5000 returns: Hello World!

The following table outlines the specific version of Flask, and its dependencies:

<table>
<thead>
<tr>
<th>Packages</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flask</td>
<td>0.10.1</td>
<td>Microframework, or micro web application framework</td>
</tr>
<tr>
<td>itsdangerous</td>
<td>0.24</td>
<td>Utility package for signing and encrypting data</td>
</tr>
<tr>
<td>jinja2</td>
<td>2.7.3</td>
<td>Template engine for python</td>
</tr>
<tr>
<td>markupsafe</td>
<td>0.23</td>
<td>Unicode escape library used alongside Jinja2</td>
</tr>
<tr>
<td>Werkzeug</td>
<td>0.96</td>
<td>WSGI (Web Server Gateway Interface) utility library for Python</td>
</tr>
</tbody>
</table>

For more information, see the [Flask](http://flask.pocoo.org/) website.
Jinja2

Jinja2 is a Python library that enables you to create templates for various output formats from a core template text file. HTML is the format that is used for QRadar apps. Jinja2 has a rich API, and large array of syntactic directives (statements, expressions, variables, tags) that you use to dynamically inject content into the template file.

Flask's in-built `render_template()` method is the easiest way to inject data from a Python method, served by the route, into a Jinja2 HTML template, as shown in the following example.

```python
@app.route('/')
def hello_world():
    return render_template('hello.html', title='QRadar')
```

The template `hello.html` contains the following code:

```html
<!doctype html>
<title>Hello from Flask</title>
<h1>Hello {{ title }}</h1>
```

The following HTML output is produced:

```html
<!doctype html>
<title>Hello from Flask</title>
<h1>Hello QRadar!</h1>
```

For more information, see the [Jinja2 website](http://jinja.pocoo.org/docs/dev/).

**HTTP request response lifecycle**

When an app is successfully installed, requests to the app are proxied only by using an established connection to QRadar. The app cannot be directly accessed by using direct URL requests or any other method.

Apps can establish a secure authenticated and authorized session to QRadar. Any authorization tokens that are created to verify that the integrity of session can be reused. The app obtains all the capabilities, security, and authenticity facets of QRadar. The app can use the user session state to get access to all of QRadar RESTful API endpoints to pull or push data to or from the QRadar system.

**Containerized apps and the network**

With the GUI Application Framework, traffic flows from container to container, from container to host on its public IP address (not localhost), and from containers to the outside world.

When each app is passed as an archive (.zip file) of source code to the QRadar endpoints, QRadar builds the initial image specific to your app codebase. Each image runs as an individual container. As the container is run or started, QRadar maps the internal flask server port (5000) to an external ephemeral port. This external ephemeral port is registered to QRadar so that proxied requests to your app code are routed to the correct container.
4 App file structure

An IBM QRadar app that you create is distributed within a compressed file.

The Hello World sample app that is created when you set up your development environment is a basic template that you can use for your application. However, the application file structure can be more complex.

The following list outlines the layout of files and sub directories that you can add to the root directory of your app. It also outlines the required nomenclature for app files and sub directories:

<App Root Folder>
  This is main directory for application files
  app/views.py
  The main entry point into the web app.
  app/templates
  An optional subdirectory that contains any Python Flask of Jinja templates that are required by the app.
  app/static
  An optional subdirectory that contains CSS, JavaScript, globalization, and other resource files.
    • css
    • js
    • resources
      The application_<LANG>.properties file is a Globalization resource bundle for the specified language code. Text strings for globalization are stored as key/value pairs in Java format properties files. If you configured text strings for globalization, they appear in QRadar when the user sets their preferences for the relevant locale.
  manifest.json
  The application manifest description file.
  src_deps
  An optional directory that contains source dependencies.
    • pip
      Optional subdirectory that contains any extra Python libraries that the app requires.
    • rpms
      Optional subdirectory that contains any RPM dependencies that the app requires. RPMs must be CentOS 6.7 x86_64 compatible.
    • init
      Optional subdirectory that contains any dependencies that the app requires that are not RPMs or Python libraries.
## Application manifest structure

The manifest is a JSON file that describes to IBM QRadar the capabilities that the app provides.

The following table describes the fields that you can include in the `manifest.json` file.

### Table 3. Application manifest fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>Yes</td>
<td>String</td>
<td>The user-readable name of the app. If the app is globalized, this field can optionally point at a resource bundle key.</td>
</tr>
<tr>
<td><strong>description</strong></td>
<td>Yes</td>
<td>String</td>
<td>The user-readable description of the app. If the application is globalized, this field can optionally point at a resource bundle.</td>
</tr>
<tr>
<td><strong>version</strong></td>
<td>Yes</td>
<td>String</td>
<td>A version string for the app. You can use any format that you want here.</td>
</tr>
<tr>
<td><strong>uuid</strong></td>
<td>Yes</td>
<td>String</td>
<td>An RFC 4122-compliant universally unique identifier for the application. The <code>create</code> command uses the Python UUID package to generate a random 128-bit number for the <code>uuid</code> value. If you do not use the SDK to create the app manifest file, you must manually enter a unique value in the <code>uuid</code> field.</td>
</tr>
<tr>
<td><strong>debugging</strong></td>
<td>No</td>
<td>Boolean</td>
<td>Set to true to turn on logging for your app. If not specified, this field defaults to false.</td>
</tr>
<tr>
<td><strong>areas</strong></td>
<td>No</td>
<td>Array of Area Type</td>
<td>One or more Area objects describe new complete pages of the application. In QRadar, Area objects are represented as tabs.</td>
</tr>
<tr>
<td><strong>rest_methods</strong></td>
<td>No</td>
<td>Array of REST Method Type</td>
<td>One or more REST Method objects describe REST methods that the app exposes. REST Method objects are required parameters for Dashboard Items and Metadata Providers, and are optional for Actions.</td>
</tr>
<tr>
<td><strong>dashboard_items</strong></td>
<td>No</td>
<td>Array of Dashboard Item Type</td>
<td>One or more Dashboard Item objects describe the contents of new items that you want to expose to the QRadar dashboard.</td>
</tr>
<tr>
<td><strong>configuration_pages</strong></td>
<td>No</td>
<td>Array of Configuration Page Type</td>
<td>One or more Configuration Page objects describe new complete pages of the app that represent configuration. In QRadar, configuration pages are opened from the Admin tab.</td>
</tr>
<tr>
<td><strong>gui_actions</strong></td>
<td>No</td>
<td>Array of GUI Action Type</td>
<td>One or more GUI Action objects describe new actions that can be performed on items in the user interface by page toolbars or by right-click menus.</td>
</tr>
<tr>
<td><strong>page_scripts</strong></td>
<td>No</td>
<td>Array of Page Script type</td>
<td>One or more Page Script objects describe new JavaScript files that you want included within an existing page in QRadar. By default, these scripts run in their own namespace.</td>
</tr>
</tbody>
</table>
Table 3. Application manifest fields (continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata_providers</td>
<td>No</td>
<td>Array of Metadata Provider type</td>
<td>One or more Metadata Provider objects describe REST methods that can be called to fetch new metadata information for certain data types in QRadar. Metadata is shown in tooltips when a mouse is hovered-over an item.</td>
</tr>
<tr>
<td>resource_bundles</td>
<td>No</td>
<td>Array of Resource Bundle type</td>
<td>One or more Resource Bundle objects. You use these objects for language locales and locale properties file locations.</td>
</tr>
<tr>
<td>dev_opts</td>
<td>No</td>
<td>Array of Developer Options type</td>
<td>One or more Developer Option objects. You use these objects to specify values that are used when you develop locally.</td>
</tr>
<tr>
<td>resources</td>
<td>No</td>
<td>Integer</td>
<td>One or more Resource objects. You use these objects to configure the amount of memory in megabytes that is available for the app to use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Supported by QRadar V7.2.7 and later.</td>
</tr>
<tr>
<td>fragments</td>
<td>No</td>
<td>Array of Fragments type</td>
<td>Use these objects to determine the injection point in the QRadar UI where content is added and the rest endpoint that is used to retrieve the content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Supported by QRadar V7.2.8 and later.</td>
</tr>
<tr>
<td>custom_columns</td>
<td>No</td>
<td>Array of Custom Columns type</td>
<td>One or more Custom column objects. You use these objects to identify the context (the page and table in the QRadar UI) where a custom column is added, a label for the column header, the type of data to be added, and the rest endpoint that is used to add the column content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Supported by QRadar V7.2.8 and later.</td>
</tr>
</tbody>
</table>

Source dependencies

If your app requires dependencies, such as RPMs or Python libraries, you can add them in the src_deps sub directory of the app folder.

The src_deps directory can contain these optional sub directories:

**pip**

Use the pip folder to install extra Python libraries. For example, if your application requires the observable-0.01.00 Python library, add the observable-0.01.00.tar.gz file to the pip folder.

Don't use .tar files for Python libraries that include extra C-based extensions. Instead, add libraries as Python wheel files (.whl), which have C-based extensions pre-compiled.

You must install Python wheel files on the same system architecture they were compiled upon. To work with IBM QRadar application framework, wheel files must be compiled on Centos 6.7 x86_64. If it uses compatible architecture, you can use the Python bdist_wheel command to create wheel files from a library's source code on your own system. The command python setup.py sdist bdist_wheel creates the wheel file when you run it from within the root directory of the Python library's source folder.
A useful alternative to manually downloading Python packages for your app is the pip2pi Python package. It requires pip and you can install it on your development computer by using the `pip install pip2pi` command. After you install this package, you run the following command:

```
pip2tgz <target-directory> <Python package>
```

For example, the following command downloads the package's wheel, along with its dependencies, into the specified folder.

```
pip2tgz python_packages/pytest/ pytest==2.8.2
```

The `pytest` parameter is optional and you can use it to download specific versions of a package.

For Python libraries that have dependencies, you can include an optional `ordering.txt` file in the `pip` folder to specify the order in which Python libraries are installed. This text file must include the names of files that are in the `/pip` folder. File names must be separated with a new line (UNIX line endings) in the order that you want them installed.

```
rmps
```

Use the `rmps` folder to install extra Red Hat Enterprise Linux (RHEL) RPMs. The RPMs must be CentOs 6.7 x86_64 compatible.

For RPMs that have dependencies, you can include an optional `ordering.txt` file in the `rmps` folder to specify the order in which RPMs are installed. This text file must include the names of files that are in the `rmps` folder. File names must be separated with a new line (UNIX line endings) in the order you want them installed.

```
init
```

Add dependencies files that do not fit into the `pip` or `rmps` folders to the `init` folder. You must also include an `ordering.txt` file in the `init` folder. The lines in this text file (UNIX line endings) are run as shell commands during the installation of the app.

For example, you might want to install a collection of RPMs that has a complex dependency chain that is not explicitly known. In this use case, you add a `.tar` file that is called `dependant_rmps.tar.gz` to the `init` folder. You add the following commands to the `ordering.txt` file:

```
mkdir /src_deps/init/dependant_rmps
cd /src_deps/init
tar -xzvf dependant_rmps.tar.gz
yum -y localinstall --disablerepo=dependant_rmps/*rpm
rm -rf dependant_rmps
```

**Note:** The `--disablerepo=` switch in this example is used to prevent the `yum` from attempting to contact remote repositories on QRadar consoles that have no internet access.

This example uses `yum`'s RPM auto-dependency resolution that installs a set of specified RPMs in the required order. If the RPMs are included in the `rmps` folder, you must specify the installation order.

### Installing Node.js as a source dependency

You can install Node.js as a web application framework to replace the Flask framework that is included with the QRadar GUI Application Framework SDK.
Procedure

1. Download the Node.js archive (.tar) that you want to use, and copy it to app/src_deps/init directory.

2. Create an installation script that is similar to the following example in the same folder that references the archive node-v6.3.0-linux-x64.tar.gz that you want to use (in this case):

```bash
#!/bin/bash

##
## install node and npm from source tarball, and make available on the path
##
cd /usr/local

tar --strip-components 1 -xzf /src_deps/init/node-v6.3.0-linux-x64.tar.gz
```

3. Create a file ordering.txt in the same folder as the following content:

```
/src_deps/init/install_nodejs_npm.sh
```

   The ordering.txt notifies QRadar to run the Node.js installation script.
5 Manifest object types

You can define several objects types in the IBM QRadar Application Framework manifest.

Areas type

Use the Areas type to add a tab to the IBM QRadar.

The following table describes the areas block fields in the manifest.json file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Yes</td>
<td>String</td>
<td>A unique ID.</td>
</tr>
<tr>
<td>text</td>
<td>Yes</td>
<td>String</td>
<td>Concise text to display that describes the area. This field can optionally point at a resource bundle key if the application is globalized.</td>
</tr>
<tr>
<td>description</td>
<td>No</td>
<td>String</td>
<td>Detailed text to display that describes the area. This field can optionally point at a resource bundle key if the application is globalized.</td>
</tr>
<tr>
<td>url</td>
<td>Yes</td>
<td>String</td>
<td>A URL to load, relative to the application root. Only URLs that live within the QRadar application can be referenced.</td>
</tr>
<tr>
<td>required_capabilities</td>
<td>No</td>
<td>Array of String</td>
<td>A set of capabilities that a user must affiliate with their user role to access this area.</td>
</tr>
</tbody>
</table>

The following code is a sample areas block from manifest.json:

```
...
areas: [
  {
    "id":"QHelloWorld",
    "text":"Hello World",
    "description":"A Hello World app",
    "url":"index",
    "required_capabilities": ["ADMIN"]
  }
],
...
```

REST method type

The REST method that the app provides. REST methods can be used by other objects in this manifest, including dashboard items, GUI actions, and metadata providers.

IBM QRadar expects the response of a REST method to be [RFC 4627](https://www.ietf.org/rfc/rfc4627.txt) compliant JSON. Arguments are passed to the method either as a query string argument or URI encoded parameters in the PUT or POST body.

REST methods are typically implemented in Python by using the Flask framework.

The following implementation of a REST method retrieves a type of metadata:
```python
@app.route('/getMetaData', methods=['GET'])
def getMetaData():
    ip = request.args.get('ip')
    # Do something with this IP and populate a variable called 'value'
    return json.dumps({'key': 'myMetaData', 'label': 'Item Label', 'value': 'value'})
```

This method is then exposed in the manifest in the following way:

```json
rest_methods: [
    {
        "name": "getMetaData",
        "url": "/getMetaData",
        "method": "GET",
        "argument_names": ["context"]
    },
],
```

The following table describes the `rest_methods` block fields in the `manifest.json` file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Yes</td>
<td>String</td>
<td>A unique name for this REST method within the app.</td>
</tr>
<tr>
<td>method</td>
<td>Yes</td>
<td>String</td>
<td>An HTTP method on named endpoint (GET/POST/DELETE/PUT).</td>
</tr>
<tr>
<td>url</td>
<td>Yes</td>
<td>String</td>
<td>The URL to access the REST method, relative to the app root. Only URLs within the app are supported.</td>
</tr>
<tr>
<td>argument_names</td>
<td>No</td>
<td>String</td>
<td>The names of arguments that this method expects. Arguments that are passed to the method are URL-encoded, as either query string parameters or in the PUT or POST body.</td>
</tr>
<tr>
<td>required_capabilities</td>
<td>No</td>
<td>Array of String</td>
<td>A set of capabilities that a user must affiliate with their user role to access this method.</td>
</tr>
</tbody>
</table>

**Dashboard items type**

A new item on the IBM QRadar dashboard. These items are available to users, who must manually add the items to their dashboard.

The contents of the item come from the response of the REST method execution. The expected format of the response is outlined in the following way:

```json
{
    "id": "Unique ID of item",
    "title": "Title text to display in the item",
    "HTML": "Contents of dashboard item, including any HTML and JavaScript you want to use."
}
```

Each time the dashboard item is refreshed, the REST method runs, so methods have a short response time. The current default refresh rate for dashboard items in QRadar is 60 seconds.

The following table describes the `dashboard_items` block fields in the `manifest.json` file.
Table 6. Dashboard_items block fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>Yes</td>
<td>String</td>
<td>A unique ID for this area.</td>
</tr>
<tr>
<td>description</td>
<td>Yes</td>
<td>String</td>
<td>Concise text to display that describes the area. Can optionally point at a resource bundle key if the application is globalized.</td>
</tr>
<tr>
<td>rest_method</td>
<td>Yes</td>
<td>String</td>
<td>Name of the REST method to load this item. Must be declared in the rest_methods section of the manifest.</td>
</tr>
<tr>
<td>required_capabilities</td>
<td>No</td>
<td>Array of String</td>
<td>A set of capabilities that a user must affiliate with their user role to have access to this item.</td>
</tr>
</tbody>
</table>

The following code is a sample dashboard_items block from the manifest.json file:

```json
...,
  "dashboard_items": [
    {
      "text": "Sample Item",
      "description": "Sample dashboard item that is a copy of most recent offenses",
      "rest_method": "sampleDashboardItem",
      "rest_arguments": null,
      "required_capabilities": ["ADMIN"]
    }
  ],
...
```

Configuration pages type

A new configuration page to be added to IBM QRadar. An app can define any user interaction that is required.

In QRadar, configuration_pages are represented as icons in the Admin tab.

The following table describes the configuration_pages block fields in the manifest.json file.

Table 7. Configuration_pages block fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>String</td>
<td>Concise text to display that describes the configuration page. This field can optionally point at a resource bundle key if the application is globalized.</td>
<td>Yes</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>Detailed text to display that describes the area. This field can optionally point at a resource bundle key if the application is globalized.</td>
<td>Yes</td>
</tr>
<tr>
<td>icon</td>
<td>String</td>
<td>Path to the icon for your app on the Admin tab, relative to the application root. Icons must be 32x32 pixels.</td>
<td>Yes</td>
</tr>
<tr>
<td>url</td>
<td>String</td>
<td>A URL to load, relative to the application root. Only URLs that exist within the QRadar application can be referenced.</td>
<td>Yes</td>
</tr>
<tr>
<td>required_capabilities</td>
<td>Array of String</td>
<td>A set of capabilities that a user must affiliate with their user role to access this configuration page.</td>
<td>No</td>
</tr>
</tbody>
</table>
The following code is a sample configuration_pages block from the manifest.json file:

```json
"configuration_pages": [  
  {  
    "text": "Open mycompany.com",  
    "description": "Loading mycompany.com in a new window",  
    "icon": null,  
    "url": "my_config_page",  
    "required_capabilities": ["ADMIN"]  
  }  
],
```

**GUI Action type**

An action that the user can do in IBM QRadar.

In QRadar, GUI Actions are represented as either buttons in page toolbars, or as right-click menu options. On touchscreen devices, GUI Actions are for items that are pressed for a long time.

When run, GUI Actions run a block of JavaScript, or invoke a REST method, or a combination of both. If you use both the `rest_method` and `javascript` attributes, the GUI Action invokes the server-side REST method within your app. It then runs the client-side JavaScript.

The following table describes the gui_actions block fields in the manifest.json file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Yes</td>
<td>String</td>
<td>A unique ID for this area within the application.</td>
</tr>
<tr>
<td>text</td>
<td>Yes</td>
<td>String</td>
<td>Concise text to display that describes the area. Can optionally point at a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>resource bundle key if the application is globalized.</td>
</tr>
<tr>
<td>description</td>
<td>No</td>
<td>String</td>
<td>Detailed text to display that describes the area. Can optionally point at a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>resource bundle key if the application is globalized.</td>
</tr>
<tr>
<td>icon</td>
<td>Yes</td>
<td>String</td>
<td>Path to the toolbar or right-click menu icon to load, relative to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>application root. Icons must be 16x16 pixels.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If you do not want to add an icon, set the value of this parameter to null.</td>
</tr>
</tbody>
</table>

---

IBM Security QRadar Application Framework: Developer Quick Start Guide
Table 8. GUI_Actions block fields (continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rest_method</td>
<td>No</td>
<td>String</td>
<td>A REST method to call when this action is performed. The context parameter must be specified as an argument. The REST method is populated with whatever the context of the GUI Action group is. The GUI Action group context varies, depending on what GUI Action group the action is invoked from. As an example, on the right-click menu of an IP address, the context parameter contains the IP address. Either this argument or the JavaScript argument is required. If both are specified, then the REST method is run first, the results of which can be passed back into the JavaScript code block by using the result variable. This method must be declared in the rest_methods block of the manifest.</td>
</tr>
<tr>
<td>javascript</td>
<td>No</td>
<td>String</td>
<td>A JavaScript code block to run when this action is performed. Either this argument or the REST method argument is required. If both are specified, then the REST method is executed first, the results of which are passed into the JavaScript code block that uses the result variable. If only the JavaScript argument is specified, a context variable that contains the context of the GUI action, is passed into the JavaScript code block.</td>
</tr>
<tr>
<td>groups</td>
<td>Yes</td>
<td>Array of String</td>
<td>A list of one or more GUI Action groups to install the action into (in other words, the identifier of the toolbar or right-click menu). You must provide at least 1 group. You can also use a group name in this format ariel:&lt;FIELD_NAME&gt;, where &lt;FIELD_NAME&gt; is the name of a field in the QRadar Event or Flow viewer. If this field is specified, the action is installed into the menu of that field, and the context parameter is the contents of the field.</td>
</tr>
<tr>
<td>required_capabilities</td>
<td>No</td>
<td>Array of String</td>
<td>A set of capabilities that a user must affiliate with their user role to access this area.</td>
</tr>
</tbody>
</table>

The following code is a sample gui_actions block from the manifest.json file:

```json
...

"gui_actions": [  
    {  
      "id":"addToReferenceSet",  
      "text":"Add To Reference Set",  
      "description":"Adds to a reference set",  
      "icon":"static/images/Btn1.png",  
      "rest_method":"addToReferenceSet",  
      "javascript":"alert(result)"  
    },  
    {  
      "id":"viewReferenceSet",  
      "text":"View Reference Set",  
      "description":"Views all the values in a reference set",  
      "icon":"static/images/Btn2.png",  
      "rest_method":"viewReferenceSet",  
      "javascript":null"  
    }  
  ]

..."
"groups": ["ipPopup"],
"required_capabilities": ["ADMIN"]
},
{
"id": "sampleToolbarButton",
"text": "Sample Toolbar Button",
"description": "Sample toolbar button that calls a REST method, passing an offense ID along",
"icon": "static/images/Btn2.png",
"rest_method": "sampleToolbarMethod",
"javascript": "alert(result)",
"groups": ["OffenseListToolbar"],
"required_capabilities": ["ADMIN"]
}
],
...

**Page scripts type**

Describes a new JavaScript file that the app includes inside an existing IBM QRadar page.

Script files that are included by the QRadar GUI Application Framework are run from the root QRadar namespace context. Scripts can interact with each other within the same app and have access to top-level functions that are defined in QRadar.

The following table describes the page_scripts block fields in the manifest.json file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_name</td>
<td>String</td>
<td>The name of the QRadar app that you want to include the scripts in. The asterisk wildcard &quot;*&quot; is also supported if it is used with the page_id field. Use the wildcard to include a file on every QRadar page.</td>
<td>Yes</td>
</tr>
<tr>
<td>page_id</td>
<td>String</td>
<td>The page ID that includes the scripts. The asterisk wildcard character &quot;*&quot; is also supported if used with the app_name field. Use the wildcard to include a file on every QRadar page.</td>
<td>Yes</td>
</tr>
<tr>
<td>scripts</td>
<td>Array of String</td>
<td>The relative path to scripts that you want to include on the page. You can add more than one script to each page. Paths to each script must be separated by a comma.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The following code is a sample page_scripts block from manifest.json:

```
...
"page_scripts": [
{
"app_name": "SEM",
"page_id": "OffenseList",
"scripts": ["/static/js/sampleScript1.js",
```


Metadata providers type

Describes metadata providers that are used to show context-sensitive information in IBM QRadar.

Metadata is shown when the user's mouse pointer hovers over an item in QRadar. The contents of the metadata comes from the response of the REST method. The following code example shows the expected format of the response:

```
{
    key: "Unique key for this metadata item",
    label: "Description of what this metadata is",
    value: "Plain-text context-sensitive data to be provided",
    html: "HTML context-sensitive data to be provided"
}
```

The following table describes the metadata_providers block fields in the manifest.json file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rest_method</td>
<td>Yes</td>
<td>String</td>
<td>Name of the REST method that is used to fetch the metadata. Must be declared in the rest_methods block of the manifest. Requires a context argument that specifies the item to fetch metadata for.</td>
</tr>
</tbody>
</table>
| metadata_type | Yes      | String  | Type of metadata that can be fetched for. The following list provides the valid values for this field:  
|               |          |         | • ip  
|               |          |         | • userName  
|               |          |         | • ariel:<FIELD_NAME>, where <FIELD_NAME> is the name of a field in the QRadar Event or Flow viewer |

The following code is a sample metadata_providers block from the manifest.json file:

```
"metadata_providers": [
    {
        "rest_method": "sampleIPInformation",
        "metadata_type": "ip"
    },
    {
        "rest_method": "sampleUserInformation",
        "metadata_type": "userName"
    },
    {
        "rest_method": "sampleURLInformation",
        "metadata_type": "ariel:URL"
    }
],
```

...
## Resource bundles type

Describes the language locales and locations of the locale properties file locations that you use when you globalize your app.

The following table describes the Resource_bundles block fields in the manifest.json file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locale</td>
<td>Yes</td>
<td>String</td>
<td>Language locale code</td>
</tr>
<tr>
<td>bundles</td>
<td>Yes</td>
<td>String</td>
<td>Path to the globalization resource bundle properties files. Files are stored the /app/resources/ folder. Properties files must use the following naming convention: application_&lt;LANG&gt;.properties.</td>
</tr>
</tbody>
</table>

The following code is a sample resource_bundles block from the manifest.json file:

```json
"resource_bundles": [
  {
    "locale": "en_US",
    "bundle": "resources/hello_en_US.properties"
  },
  {
    "locale": "es",
    "bundle": "resources/hello_es.properties"
  },
  {
    "locale": "fr",
    "bundle": "resources/hello_fr.properties"
  },
  {
    "locale": "en",
    "bundle": "resources/hello_en.properties"
  }
],
```

Text strings for globalization are stored as key/value pairs in Java format properties files. If you configured text strings for globalization, they appear in IBM QRadar when the user sets their preferences for the relevant locale.

## Developer options type

Describes values that you use to develop and test your app locally.

The dev_opts object type is used only for local testing. Parameters in dev_opts blocks are not passed to the IBM QRadar Console when you deploy your app.

You use the dev_opts block in the manifest.json file to specify the IP address of a remote QRadar Console. If your application uses QRadar API endpoints, you can use the dev_opts object to contact the QRadar API and test the application locally before you upload to your QRadar production instance.

The following table describes the parameters of the Dev_opts block:
Table 12. Dev_opts block fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>console_ip</td>
<td>Yes</td>
<td>String</td>
<td>The IP address of the remote QRadar Console that contacts the API endpoints</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>that your app uses.</td>
</tr>
</tbody>
</table>

The following code is a sample dev_opts block from the manifest.json file:

```json
...
"dev_opts": [
  {
    "console_ip":"10.11.12.13",
  }
],
...
```

When you run your app, you are prompted for your QRadar user name and password. You can also store those credentials for your local development. Credentials are stored in clear text in the `<USER_HOME>/.qradar_appfw.auth` file on Unix and Linux, and in the `C:/Users/<USER_HOME>/.qradar_appfw.auth` file on Windows.

### Resources type

Defines the memory resources to allocate to your app.

The Resources object is not an IBM QRadar app type. You use it to configure the amount of memory, in megabytes, that your app can use.

The following table describes the Resources block fields in the manifest.json file.

Table 13. Resources block fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>No</td>
<td>Integer</td>
<td>The amount of memory in megabytes that is available for the app to use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If the resources block is omitted from manifest.json, 200 megabytes of memory is allocated to your app by default.</td>
</tr>
</tbody>
</table>

The following code is a sample resources block from the manifest.json file:

```json
...
"resources": [  
  {   
    "memory":"500",
  }  
],
...
```

**Note:** The combined memory requirements of all the apps that are installed on a QRadar Console cannot exceed 10 per cent of the total available memory. If you install an app that causes the 10 per cent memory limit to be exceeded, the app does not work.
If your app requires a minimum memory allocation, you must provide information about it in your app’s documentation.

**Fragments type**

Use to inject custom content fragments into IBM QRadar.

The fragments block contains fields that define the tab, page, and page area where you want to inject custom content. It also defines the REST endpoint that QRadar uses to generate custom content. Different apps can inject content into the same page location. However, each app can insert only one set of HTML content for each location.

The following table describes the fragments block fields in the manifest.json file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_name</td>
<td>Yes</td>
<td>String</td>
<td>The QRadar app into which the content is to be injected.</td>
</tr>
<tr>
<td>page_id</td>
<td>Yes</td>
<td>String</td>
<td>The identifier of the page in QRadar UI into which the content is injected.</td>
</tr>
<tr>
<td>location</td>
<td>No</td>
<td>String</td>
<td>The location on the QRadar page where the custom content is injected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To inject content at the top of a page, if permitted, the value for this parameter is header. To inject content at the bottom of a page, if permitted, the value for this parameter is footer. If the page has only one injection point, don’t include this field.</td>
</tr>
<tr>
<td>rest_endpoint</td>
<td>Yes</td>
<td>String</td>
<td>Identifies the REST API endpoint that QRadar invokes to retrieve the custom content. Can be any REST endpoint, not necessarily one that is provided by this app. An app’s custom fragment REST endpoint must return a JSON response body.</td>
</tr>
</tbody>
</table>

The following example specifies content that is retrieved from the /myoffensesheadercontent endpoint to be injected into the header area of the Offense List page.

```json
{
  ...
  "fragments": [
    {
      "app_name": "SEM",
      "page_id": "MyOffenseList",
      "location": "header",
      "rest_endpoint": "/myoffensesheadercontent"
    },
    ...
  ]
}
```

For more information about adding custom content fragments to the QRadar user interface, see “Custom fragments example” on page 72.
**Custom columns type**

Adds custom columns to tables in IBM QRadar.

You can add columns in various locations in QRadar. Different apps can add a column to the same table. However, each app can add one column only for each table.

The following table describes the custom_columns block fields in the manifest.json file.

<table>
<thead>
<tr>
<th>Field</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Yes</td>
<td>String</td>
<td>The name of the column. This string can also be a resource bundle key for globalization purposes.</td>
</tr>
<tr>
<td>rest_endpoint</td>
<td>Yes</td>
<td>String</td>
<td>Identifies the REST API endpoint that QRadar invokes to retrieve the custom content for the column. An app’s custom fragment REST endpoint must return a JSON response body.</td>
</tr>
<tr>
<td>page_id</td>
<td>Yes</td>
<td>String</td>
<td>The ID of the page that contains the table where the column is to be added.</td>
</tr>
</tbody>
</table>

The following code sample shows how to configure the custom_columns block in the manifest file.

```json
{
  ...
  "custom_columns": [
    {
      "label": "labelfrommanifest",
      "rest_endpoint": "test_method",
      "page_id": "AssetList"
    }
  ],
  ...
}
```

**Related concepts:**

- "Custom column example" on page 74

You can add columns that contain custom content to tables in QRadar.
6 The Hello World sample app

When you add an app in the Application Development Manager window, a simple "Hello World" sample app is also created.

The Hello World sample app adds a Hello World tab to QRadar.

The following image shows an example of the Hello World tab that is added to QRadar.

![Hello World tab in QRadar](image)

You can use this sample app as a simple template from which to build your own apps that require tabs. When you run the development environment script, the files that are described in the following table are added to your application development folder:

<table>
<thead>
<tr>
<th>Files/Folders</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>The root directory contains the following files:</td>
</tr>
<tr>
<td></td>
<td>qpylib contains the Python library files that your app uses to connect to the QRadar API endpoints.</td>
</tr>
<tr>
<td></td>
<td><strong>init</strong>.py a sample initialization file that creates a Flask instance, imports views from the views.py script and functions from the qpylib library.</td>
</tr>
<tr>
<td></td>
<td>views.py the main entry point into the web application. This file and the manifest.json file are the only files that are required in every app. Contains sample code for the Hello World app.</td>
</tr>
<tr>
<td>qradar_appfw_venv</td>
<td>Contains the Python virtual environment where the dependencies are installed.</td>
</tr>
<tr>
<td><strong>init</strong>.py</td>
<td>Creates an instance of the Flask micro-framework that is used to serve content to QRadar.</td>
</tr>
<tr>
<td>manifest.json</td>
<td>Describes to QRadar what the sample Hello World app does.</td>
</tr>
<tr>
<td>run.py</td>
<td>Contains instructions to run the code that is stored in the app sub directory.</td>
</tr>
</tbody>
</table>

### manifest.json

The manifest.json file contains the following code:

```json
{
   "name":"Hello World",
   "description":"Application to display hello world",
   "version":"1.0",
   "uuid":"558d7935-f00b-42da-a278-c82abdb12b34",
   "areas": [ |
      {  
         "id":"QHelloWorld",
```
"text":"Hello World",
"description":"A Hello World app",
"url":"index",
"required_capabilities": ["ADMIN"]
],
"dev_opts": [
{
"console_ip": ""
}
]
}

The first four objects, name, description, version, and uuid provide basic app information.

The areas object describes the capabilities of the Hello World app. The QRadar GUI Application Framework uses areas objects to describe new complete pages of the app. Areas objects are represented as tabs in the user interface.

The areas block contains the fields that are described in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The ID of the new tab</td>
<td>QHelloWorld</td>
</tr>
<tr>
<td>text</td>
<td>The name of the tab that is displayed in the user interface.</td>
<td>Hello World</td>
</tr>
<tr>
<td>description</td>
<td>A description of the tab that is displayed.</td>
<td>A Hello World app</td>
</tr>
<tr>
<td>url</td>
<td>Describes the route that is defined in the views.py script that QRadar uses so it can display the &quot;Hello, World!&quot; text in the body of the new tab.</td>
<td>index</td>
</tr>
<tr>
<td>required_capabilities</td>
<td>Instructs QRadar to display the Hello World tab only to users with Administrator privileges.</td>
<td>[&quot;ADMIN&quot;]</td>
</tr>
</tbody>
</table>

The dev_opts block is used to provide the IP address of a networked instance of QRadar Console for testing purposes. This block is not required for this sample app.

views.py

The views.py file contains the following code:

```python
__author__ = 'IBM'

from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

The code creates the default '/' and '/index' routes, both of which return a simple string. The index route is declared in the url field of the manifest.json file.

**Note:** You do not have to include the __author__ tag, but it is considered good practice to use it.
App startup

When QRadar starts your app, it calls the run.py and _init_.py scripts. The _init_.py file creates an instance of the Flask microframework environment that imports your views module. Your views modules define all the necessary endpoints and routes that serve content back to QRadar.

```python
__author__ = 'IBM'

from flask import Flask

app = Flask(__name__)
from app import views
```

The run.py file creates a new Flask application (by starting the Flask web server), from the app directory.

```python
__author__ = 'IBM'

from app import app
app.run(debug = True, host='0.0.0.0')
```

What can you do with the Hello World sample app?

You can use the Hello World sample app to test the QRadar SDK in these ways:

- Run the Hello World app locally.
- If you have a test instance of QRadar Console, you use the SDK to package and upload the Hello World app to it.

However, most importantly you can use the Hello World sample app files as a template to start developing your own QRadar apps.

New tab example

You can build on the Hello World sample app that the IBM QRadar SDK installs in your app workspace to add a tab to QRadar.

You can build an app that uses a Jinja2 template to serve HTML content to a new tab as shown in the following image.

![Image](image-url)

The files and folders that are described in the following table are required for the tab example:
<table>
<thead>
<tr>
<th>Files/Folders</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>The main directory for application files. The app folder contains the following files:</td>
</tr>
<tr>
<td></td>
<td>qpylib remains unchanged from the Hello World sample app example.</td>
</tr>
<tr>
<td></td>
<td><strong>init</strong>.py remains unchanged from the Hello World sample app example.</td>
</tr>
<tr>
<td></td>
<td>views.py updated to include code to render the Jinja2 template. Here's snippets of the additional code that is added to views.py that is used to return the render_template:</td>
</tr>
<tr>
<td></td>
<td>from flask import render_template</td>
</tr>
<tr>
<td></td>
<td>def index():</td>
</tr>
<tr>
<td></td>
<td>return render_template(&quot;index.html&quot;,</td>
</tr>
<tr>
<td></td>
<td>title = &quot;QApp1: Hello World!&quot;)</td>
</tr>
<tr>
<td></td>
<td>The Flask app route uses a Flask-Jinja2 templated HTML page to build the content for the Hello World tab.</td>
</tr>
<tr>
<td></td>
<td>/templates/index.html - The Jinja2 template to render when requests are routed to app.route annotated endpoints.</td>
</tr>
<tr>
<td></td>
<td>/static/css/style.css Renders the content that is served by the Jinja2 template, index.html.</td>
</tr>
<tr>
<td><strong>init</strong>.py</td>
<td>This file creates an instance of the Flask micro-framework that is used to serve content to QRadar. This file remains unchanged from the original Hello World example.</td>
</tr>
<tr>
<td>manifest.json</td>
<td>Describes to QRadar that your app creates a new tab. Here's a snippet of the code, which is changed slightly from the Hello World sample app example.</td>
</tr>
<tr>
<td></td>
<td>&quot;areas&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;id&quot;:&quot;QHelloWorld&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;text&quot;:&quot;Hello World&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;description&quot;:&quot;A Hello World app&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;url&quot;:&quot;index&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;required_capabilities&quot;: [&quot;ADMIN&quot;]</td>
</tr>
<tr>
<td></td>
<td>The devOpts is removed and is needed only for testing the app on a networked QRadar Console.</td>
</tr>
<tr>
<td></td>
<td>Functionally, this manifest file is identical to the manifest.json that is provided in the basic Hello World sample app example.</td>
</tr>
<tr>
<td>run.py</td>
<td>Remains unchanged from the Hello World sample app example.</td>
</tr>
</tbody>
</table>

**manifest.json**

```json
{
  "name":"QHelloWorld_1",
  "description":"Application to display QHelloWorld",
  "version":"1.0",

  "areas": [ |
    { |
      "id":"QHelloWorld_1",
      "text":"QHelloWorld_1",
      "description":"An Hello World app with some styling",
      "url":"index",
      "required_capabilities": ["ADMIN"] |
    }
  ]
}
```
Functionally, this manifest file is identical to the manifest.json that is provided in the basic "Hello World" sample. The fields that describe the app are updated and the dev_opts block is removed because it is not needed.

views.py

The views.py file contains the following code:

```python
__author__ = 'IBM'

from flask import render_template
from app import app

@app.route('/
@app.route('/index'

def index():
    return render_template("index.html", title = "QApp1 : Hello World !")

The views.py file imports the render_template method from Flask to render the index.html template.

Like the views.py file in the 'Hello World' sample, the code creates default routes '/ ' and '/index', both of which return a simple string. The index route is declared in the url field of manifest.json.

templates/index.html

This Jinja2 template contains the HTML content that is displayed on the new tab. It includes a variable that uses the value of the title parameter that is defined in views.py for the browser window title text.

```html
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>{{title}} - Main</title>
    <link rel="stylesheet" href="static/css/style.css">
</head>
<body>
    <div id="pageheader" class="pagebanner">
        IBM QRadar Application : Hello World !...
    </div>

    <div id="contentpane">
        Hello! Welcome to the first Qradar app served from the AppFramework
    </div>

</body>
</html>
```
7 QRadar App Editor

Develop and edit apps directly in IBM QRadar by using the QRadar App Editor.

The QRadar App Editor is an easy-to-use editor that you start from the QRadar Admin tab. Any app developer can use the QRadar App Editor.

You can do any of the following tasks with the QRadar App Editor:
• Develop new apps from a template.
• Add, edit, and remove files and folders.
• Delete apps that you create with the App Editor.
• Import and edit existing apps.
• Clone an app from a Git repository.
• Deploy apps that you are editing and developing.
• Export apps.

What’s new in the QRadar App Editor

Learn about new features in each version of the IBM QRadar App Editor.

Version 2.1
• Configure proxy settings to connect to a GIT repository.
• Download external app templates from Github (https://github.com/ibm-security-intelligence/sample-apps). You can submit Github pull requests for your sample apps that you think are helpful to the development community.
• Search for files in the app folder navigation.
• Copy or move files in the app folder navigation.
• View gif, jpeg, bmp, and png images in the editor workspace.

Version 2.0
• Clone apps from a GIT repository.
• View app installation status.
• Upload files to your app.
• Delete apps.
• Help pages added.

Installing the QRadar App Editor

Download and install the IBM QRadar App Editor from the IBM App Exchange.

Before you begin

You must have an IBM ID to access the App Exchange (https://exchange.xforce.ibmcloud.com/) and download the app. You can register for an IBM ID at IBM id registration (https://www.ibm.com/account/profile/). You must have internet access for the app to access other QRadar resources such as videos and information when it is installed.

Note:
To install the QRadar App Editor your installed version of QRadar must have Python V2.7, which is available with the following releases:

- IBM® Security QRadar® 7.2.8 software update 7 (7.2.8.20170530170730)
- IBM® Security QRadar® 7.3.0 software update 2 (7.3.0.20170620100024)
- IBM® Security QRadar® 7.3.1 GA (7.3.1.20171206222136)

**Procedure**

1. Download the QRadar App Editor extension from the [App Exchange](https://exchange.xforce.ibmcloud.com/).

2. Open the **Admin** settings:
   a. In IBM QRadar V7.3.0 or earlier, click the **Admin** tab.
   b. In IBM QRadar V7.3.1 and later, click the navigation menu ( ), and then click **Admin** to open the admin tab.

3. Click **Extensions Management**.
   a. Click **Add**.
   b. In the Add a New Extension window, click **Browse** to find the app extension that you downloaded.
   c. Select **Install immediately**, and then click **Add**.
   d. Click **Install**.

   The extension appears in the Extensions Management window after it is installed.

4. Refresh your browser to see the **Develop Applications** icon on the **Admin** tab.

---

**Starting the QRadar App Editor**

Start the IBM QRadar App Editor and edit an existing app, create a new app, or access helpful resources.

**Procedure**

1. Open the **Admin** settings:
   a. In IBM QRadar V7.3.0 or earlier, click the **Admin** tab.
   b. In IBM QRadar V7.3.1 and later, click the navigation menu ( ), and then click **Admin** to open the admin tab.

2. On the navigation menu, click **Apps**, and then click the **Develop Applications** icon.

   The following table shows the tiles that you can select on the IBM Application Development Manager window.

<table>
<thead>
<tr>
<th>Tile</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New App</td>
<td>Develop a new app by using the Hello World built-in template or by selecting one of the other templates that you clone from a Git repository.</td>
</tr>
<tr>
<td>Existing App</td>
<td>Edit or develop an existing app that you import into the QRadar App Editor, or clone from a Git repository.</td>
</tr>
<tr>
<td>Resources</td>
<td>Development resources</td>
</tr>
<tr>
<td>Getting Started Video</td>
<td>Learn to use the QRadar App Editor.</td>
</tr>
</tbody>
</table>

3. To develop a new app that uses the one of the available templates, click the **New App** tile.
   a. Select a template.
   b. Type a name, description, and version for your new app.
c. Click **Install** to add the new app development tab that includes the editor to your QRadar Console.

d. Refresh your browser to see the tab for the app in development mode in QRadar.

The Hello World app is an app that features a custom tab, so you can see a second tab that shows the app in live mode without the editor. You might have to wait for a short time before the tabs appear in the user interface. Any app that you install shows the development tab with the editor and whatever functionality the app brings with it, for example the app might show a custom column or a custom tab.

If you install the **Hello World** template, you can see the **Hello World** app in normal install mode without the editor on the first tab, and in development mode on the second tab. Use the App Editor to develop your app from the development tab.

Some app templates might not be QRadar tabs so you only get the development mode tab, for example, you only see one tab when you select the **dashboard_items** template.

The following screen capture shows an example of the Hello World app in the App Editor tab.

![IBM QRadar Security Intelligence](image)

**Figure 2. Hello World app in the editor**

4. To edit and develop existing apps by importing the files into the QRadar App Editor, click the **Existing App** tile.

   - To select an existing app (.zip file), click **Browse** to find the local app, or drag and drop an app into the app drop box.

   **Note:** You can’t import an app that is packaged as an extension that you download from the App Exchange. Apps that are packaged as extensions are only installed through the **Extensions Management** tool in the QRadar Console and are not editable in the App Editor. You can import and edit the app only in the app package (.zip) format.

   - To clone an app from a Git repository, type the Git repository URL in the **Enter the git repository url** field that is shown in the following image:
Select the **Authentication Required** check box if you are required to provide credentials to clone the app.

Select the **Ignore Certificate Validation** check box to turn off certificate validation when you download the app from the Git repository. This feature is useful when you're using a local repository that you trust and you know the certificates are not current but you want to turn off the warnings.

5. To configure a proxy to connect the App Editor to the GIT repository for sample apps or to clone apps, click **Proxy Settings**. Then, type a http address and a port number to connect to your proxy.

6. Click **Install** to add the app development tab for the existing app to your QRadar user interface, and then refresh your browser to see the tab.

You might have to wait for a short time before the tab for the app appears in QRadar.

### Editing apps in the editor

Edit and create files and folders in the app editor tab, which is created on the QRadar Console.

**About this task**

When you install a new app, the **Hello World** development tab that includes the editor is added to the Console, and a second tab shows the **Hello World** app in normal installed mode without the editor.

The following diagram shows the first **Hello World** app tab without the editor, and the **Hello World - Custom Tab Development** tab that includes the editor on QRadar.

**Procedure**

1. To edit files in the app folder, double-click a file, or right-click to view the File Edit menu.

2. To create files and folders, save files, and close files, click **File**, and then select an option from the menu.

3. To search for files in the app folder navigation, type the file name in the search box, and then click the search icon to search for your file.

4. Copy or move files from one folder to folder in the app folder navigation.
   a. To copy a file from one folder to another folder, select the file and press the CTRL key, and then use your mouse to drag the file to the destination folder.
   b. To move a file from one folder to another folder, select the file by using your mouse, and drag the file to the destination folder.
5. To upload files to the app folder, use any of the following three ways to open the Upload File screen:
   • Click > File > Upload, and then drag a file in the box, or browse to a file.
   • In the app folder navigation, select a folder, and then right-click to open the Upload File screen.
   • In the app folder navigation, select a folder, and then drag a file over the folder to open the Upload File screen.

6. To deploy an app or export an app, click the Actions menu and choose one of the following options:
   • Click Actions > Deploy App > in Development Mode to deploy and upgrade the current app in development mode.
     When you make changes, such as editing the manifest or other files, you deploy and upgrade the app in development mode to apply your changes. The app version in the manifest file increments when you deploy an app.
     When you deploy your app in development mode, you have the option to save a copy of the current app in .zip format.
   • Click Actions > Deploy App > in Live Mode to view the app in QRadar.
     Before, you deploy the app in Live Mode, it's good practice to save your app locally, otherwise you can't access your changes after the app is deployed because the editor tab is removed.
     The development mode tab that includes the App Editor is removed from the Console when you deploy the app in Live Mode. If you want to edit the app again, you must export it as a .zip file so that you can import it to QRadar and edit it at a later time.
   • Click Actions > Export App > as Zip to save your app in a local folder.
     The app is saved as an archive (.zip) file, which you can import as an existing app later. The .zip file that you export is not an app extension that can be installed by using the Extensions Management tool on the Admin tab.
     When you export the app, the app version in the manifest displays in the export dialog box. When you change the version that appears in the export dialog box, the app version in the manifest file version is changed to the same version.
   • Click Actions > Delete App to delete the app that you're editing. The app is removed from the QRadar Console.

Note: If you make changes to the app, ensure that you export the app so that your changes are saved in the exported zip file and you can install the app again later. The File > Save function saves the session only in QRadar, and does not write to the compressed file, so these changes are lost because the app files are removed when you delete the app.
8 Software development kit overview

The IBM QRadar application framework comes with its own software development kit (SDK).

Use the QRadar Application Framework software development kit (SDK) to do the following application development tasks:

You [download](https://developer.ibm.com/qradar/) the SDK and extract the QRadar Application Framework SDK archive (.zip file) from DeveloperWorks.

Create a development workspace

The QRadar Application Framework SDK installs a development workspace that has a sample application that you use as a template to build your own extension.

Run your app locally for test purposes

You don't need to upload your extension code to a live QRadar Console instance to test your app. The QRadar application framework SDK includes a virtual development environment that you can use to run your application locally.

If your app uses QRadar API endpoints, you can configure the virtual environment to connect to the API on the QRadar Console instance and test locally. You do not need to upload to QRadar Console.

Package your app

The QRadar Application Framework SDK includes a packaging utility that you use to create an archive (.zip file) that contains your extension files.

Deploy your app to QRadar Console

The QRadar Application Framework SDK includes a deployment utility that you can use to upload your packaged app directly to a live QRadar Console instance.

Optimize app memory usage

Tune the IBM QRadar Application Framework to optimize app memory usage.

Apps that run on the QRadar Console are limited to 200 MB of memory. When an app exceeds this threshold, logs are generated in the `/var/log/qradar.error` file. Eventually, the container that hosts the app shuts down and is restarted if the app continues to use memory beyond this threshold.

You can help prevent your app from exceeding the 200 MB threshold in any of the following methods:

- Avoid allocating large amounts of memory by chunking (or staggering) the work into small memory footprints.
- Change the memory model that is used by the Application Framework.
- Call for garbage collection when you’re finished with code that uses large amounts of memory.

Changing the Application Framework memory model

By default, the Application Framework configures the Werkzeug WSGI web application server that Flask uses to run as a single process. Threads are used to handle each request. You can configure the application server to create a separate process to handle each new request. When the request is completed, the process is removed, and all of the memory that is allocated by the Python interpreter to process this request is released.
To override this behavior, edit the run.py file and add `threaded=False` and `process=N` where N is greater than 1. In the following example, a value of `process=3` allocates approximately 25 MB per interpreter and leaves some room for growth.

```python
__author__ = 'IBM'

from app import app
from app.qpylib import qpylib

qpylib.create_log()
app.run(debug=True, host='0.0.0.0', threaded=False, process=3)
```

Include the source to the run.py in the template folder within your app archive file (.zip). The run.py file that is created during the installation is then overwritten with your settings.

**Note:** When you package an app with the SDK, the run.py is skipped and you must manually add it to your app archive file (.zip).

For more information about parameters that can be passed to the Werkzeug WSGI web application server, see [http://werkzeug.pocoo.org/docs/0.11/serving/](http://werkzeug.pocoo.org/docs/0.11/serving/)

### Calling for garbage collection

The Python interpreter might not know when to free the memory. You can speed up garbage collection by placing the following code right after sections where large amounts of memory are no longer needed:

```python
import gc
gc.collect()
```

**Note:** Python does not ensure that any memory that your code uses gets returned to the OS. Garbage collection ensures that the memory used by an object is free to be used by another object at some future time. Changing the Application Framework memory model option is important for apps that run for a long time. Killing the process ensures the memory is freed for use by other components.

### Tools

Some tools that can help you identify memory problems:

#### Memory Profiler

A Python module for monitoring memory consumption of a process. For more information, see [https://pypi.python.org/pypi/memory_profiler](https://pypi.python.org/pypi/memory_profiler)

#### Linux utilities

The command-line utility `top` can be used to monitor all Python processes running on the machine:

```
top -p $(pgrep -d ',' python)
```

You can also use the following command to get the total MB used by all Python interpreters on your system:

```
ps -e -o pid,comm,rss | grep python | awk '/python/{print $3}' | awk '{sum+=$1} END
```

#### Resource Module

You can log the amount of memory your process uses by adding the following code to your module:
import resource
print 'Memory usage: %s (kb)' % resource.getusage
(resource.RUSAGE_SELF).ru_maxrss

Related concepts:
"Resources type" on page 25
Defines the memory resources to allocate to your app.
9 Installing the SDK

The IBM QRadar Application Framework SDK can be installed on Windows, Linux, or OSX operating system.

Before you begin

The following operating systems versions are supported:
- Windows 7
- RHEL 6.6 or 6.7
- OSX 10.11

The SDK supports only the 64-bit version of Python on Windows.

Before you install the SDK, you must install at least Python version 2.7.9.

The SDK does not support Python 3.0 or later versions.

To find out which version of Python is installed on your QRadar instance, use SSH to log in to your QRadar Console and use the python -version command. The SDK installation script for Windows is python.exe, which is located in the c:\Python26\ directory.

You can download Python from [Python Downloads](https://www.python.org/downloads/).

Procedure

1. Download and extract the QRadar Application Framework SDK archive (.zip file).
2. Perform the following task:
   - If you are installing on Windows, double-click install.bat to run the installation process. The SDK is installed to the c:\QRadarAppSDK folder.
   - If you are installing on Linux or Unix,
     a. Open a terminal window
     b. Use the cd command to change to the QRadarAppSDK folder.
     c. Type the following command:
        sudo ./install.sh
     d. Press Enter

What to do next

After you install the QRadar Application Framework, create your development environment
10 Use Python 2.7 in your app

Use environmental variables to make your app use Python 2.7 instead of Python 2.6.

This feature was introduced in IBM QRadar 7.2.8 Patch 7 (7.2.8.20170530170730) SFS.

Add the following environmental variables to the manifest file to instruct the app to use Python 2.7:

```
"environment_variables": [
    {
        "name": "PATHSTART",
        "value": "/usr/local/bin"
    }
],
```

The following manifest file example includes the environment variables that make the app use Python 2.7.

```
{
    "environment_variables": [
        {
            "name": "PATHSTART",
            "value": "/usr/local/bin"
        }
    ],
    "console_ip": "10.11.12.13",
    "configuration_pages": [
        {
            "required_capabilities": [
                "ADMIN"
            ],
            "description": "Application Development",
            "text": "Develop Applications",
            "url": "bootstrapper"
        }
    ],
    "name": "Application Development",
    "log_level": "debug",
    "description": "Application for installing other applications in Development-mode.",
    "resources": {
        "memory": 200
    },
    "version": "1.0.2",
    "uuid": "ae1c83d0-be94-11e5-a837-0800200c9a66",
    "app_id": 1006
}
```
11 Creating your development environment

Use the QRadar GUI Application Framework SDK (Software Development Kit) to create a basic development environment for your app.

About this task

The SDK provides a sample template app that you use as a template to create your own app. Use Python 2.7.9 to develop your QRadar apps.

Procedure

1. Create a folder on your computer for the QRadar sample app. Name the directory according to the following format:
   `<Author_Name>.<App_Name>.<App_Version>`

   For example, the following directory name is an example of a good naming convention:
   com.me.myApp.1.0.0

2. Depending on your operating system, type the following command: `qradar_app_creator create -w <path_to_app_folder>/com.me.myApp.1.0.0`
   - On UNIX and Linux operating systems, type the following command:
     `qradar_app_creator create -w <path_to_app_folder>/com.me.myApp.1.0.0`
     **Important:** On Linux operating systems, keep the absolute path to your workspace short. File paths might be truncated because of the BINPRM_BUF_SIZE kernel constant (79 or 127 characters, depending on kernel version), leading to failure of this command.
   - On Windows operating systems, type the following command:
     `qradar_app_creator create -w <path_to_app_folder>\com.me.myApp.1.0.0`
     **Important:** Folder and file names in paths must not contain spaces.

Results

When you run the development environment script, the following folders and files that are described in the table are added to your app development folder.

**Table 19. QRadar App files and folders in the development folder**

<table>
<thead>
<tr>
<th>Files/Folders</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>The root directory for application files. This directory contains the following files: The <code>qpylib</code> folder contains the Python library files that your app uses to run QRadar tasks. For example, you can use the <code>qpylib</code> library to connect to API endpoints, and get the storage path. The <code>__init__.py</code> sample initialization file for your app. Creates a Flask instance, imports views from <code>views.py</code> and functions from the <code>qpylib</code> library. The <code>views.py</code> file is the main entry point into the web app. This file and the <code>manifest.json</code> file are the only files that are required for every app. This file contains sample code for the &quot;Hello World&quot; application.</td>
</tr>
<tr>
<td>store</td>
<td>The directory where the app data is stored. It is not packaged into your app.</td>
</tr>
<tr>
<td>qradar_appfw_venv</td>
<td>Contains the Python virtual environment where the dependencies are installed.</td>
</tr>
</tbody>
</table>
Table 19. QRadar App files and folders in the development folder (continued)

<table>
<thead>
<tr>
<th>Files/Folders</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>.py</td>
<td>Creates an instance of the Flask micro-framework that is used to serve content to QRadar.</td>
</tr>
<tr>
<td>manifest.json</td>
<td>Describes what the sample &quot;Hello World&quot; app does.</td>
</tr>
<tr>
<td>run.py</td>
<td>Contains instructions to run the code that is in the app subdirectory.</td>
</tr>
</tbody>
</table>

What to do next

You are now ready to begin coding your app. Familiarize yourself with the requirements for the app and manifest file structures.

Developing apps in Eclipse

After you set up your development environment, you import it into Eclipse to use that Eclipse integrated development environment (IDE) features to develop your app.

About this task

For Python development, install PyDev, the Eclipse IDE that is used in Python development. The latest version of Eclipse can be found on the Eclipse website [https://eclipse.org/downloads/](https://eclipse.org/downloads/).

Procedure

1. Install PyDev into Eclipse.
   - To install from the Eclipse Marketplace, click Help > Eclipse Marketplace on the main Eclipse Help panel.
   - To install the PyDev repository from (http://pydev.org/updates), click Help > Install New Software on the main Eclipse Help panel.
2. After you install PyDev, switch the perspective to PyDev.
3. In the PyDev perspective, click File > New > PyDev project and do the following tasks in the PyDev Project dialog box.
   a. Enter a project name.
   b. Select 2.6 from the Grammar Version list.
   c. Select Create links to existing sources
   d. To configure the interpreter to use the virtual environment in the SDK, click Click here to configure an interpreter not listed.
   e. Click New in the Python Interpreters dialog and enter a name and path to the Python executable file on your system.
   f. Click OK.
      Ensure that the site-packages folder path (.../<app_name>/qradar_appfw_venw/lib/python2.7/site-packages) check box is selected when you select the folders to be added to the system python path.
   g. Click Apply, and OK to return to the PyDev Project dialog.
   h. Click Create links to existing sources, and then click Next.
   i. Click Add external source folder and go to the root directory of your development environment, click OK, and then click Finish.
      The new Eclipse PyDev project that contains your development environment appears in your Package Explorer.
Installing Python 2.7.9 on OSX

You must install Python 2.7.9 to run IBM QRadar Application Framework SDK on the OSX operating system.

About this task

OSX usually comes with Python 2.7.x. The QRadar app framework SDK requires Python 2.7.9.

Procedure

1. To install the HomeBrew package manager, type this command:
   ruby -e "$\{curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install\}"
2. To install the Python version manager, pyenv, type this command:
   brew install pyenv
3. To use pyenv to install Python 2.7.9, type this command:
   pyenv install 2.7.9
4. To check that Python 2.7.9 was installed correctly, type this command:
   pyenv versions
5. To use Python 2.7.9 locally, type this command:
   pyenv global 2.7.9
6. To check your Python version, type this command:
   python --version
   If this command returns 2.7.9, the installation was successful.
   If this command returns 2.7.x, it might indicate an issue with pyenv. To solve this problem, open or create the "$HOME/.bash_profile file in a text editor and add the following lines:
   export PATH=/usr/local/bin:$PATH
   export PYENV_ROOT="$HOME/.pyenv"
   export PATH="$PYENV_ROOT/shims:$PATH"
   Type the python --version command to check that Python 2.7.9 was installed.

Packaging and deploying your app

Use the QRadar GUI Application Framework to package your app as an archive (.zip file) and deploy it to your IBM QRadar test environment.

Procedure

1. Open a shell prompt and use the cd command to go to the bin sub folder of your SDK installation folder.
2. Package your app by using the following command syntax:
   qradar_app_creator package -w <path_to_app_folder>/<app_name_folder> -p <app_name.zip>
   • To package an app on UNIX or Linux computers, type the following command:
     qradar_app_creator package -w <path_to_app_folder>/<app_name_folder> -p <app_name.zip>
   • To package an app on a Windows computer, type the following command:
     qradar_app_creator package -w <path_to_app_folder>\<app_name_folder> -p <app_name.zip>
3. Upload your app (extension) to your QRadar Console by using the following command, and then press Enter.
   qradar_app_creator deploy -q <QRadar_console_IP> -u admin -p <app_name.zip>
   Here is an example of the command:
   qradar_app_creator deploy -q 10.11.12.13 -u admin -p com.me.myapp.zip
Note: Make sure that you record the application ID for your app from Application Creation Task state output that is returned when you issue the `qradar_app_creator deploy` command. You need it when you export your app as a QRadar extension.

In the following example, the application ID is 1023.

Application Creation Task state: {u'status': u'COMPLETED', u'application_id': u'1023', u'error_messages': u'[]'}

If you're viewing your QRadar Console user interface in a browser, refresh your browser to see your app running.

What to do next

After you successfully test your app, you can use the QRadar Content Management Tool to export the app as an extension. You then use the QRadar Extensions Management tool on the Admin settings page to install your extension on your QRadar Console.

Running your application locally

Rather than packaging and uploading your app every time you change something, you can test it by running it locally in a browser window.

About this task

The IBM QRadar GUI Application Framework includes a virtual environment that you can use to run your application locally for testing purposes.

If your app uses QRadar API endpoints, you must connect to a QRadar Console instance. You configure the IP address of the QRadar Console test instance in the `console_ip` field of the `dev_opts` object block in your app's `manifest.json` file.

Procedure

1. Run your app locally.
   - To run on Unix or Linux, type the following command:
     `qradar_app_creator run -w <path_to_app_folder>/<app_folder_name>`
   - To run on Windows, type the following command:
     `qradar_app_creator run -w <path_to_app_folder>\<app_folder_name>`

2. Open a browser and type `http://0.0.0.0:5000` in the address bar. If your browser does not support the `http://0.0.0.0` URL, type `http://127.0.0.1:5000`.
   Your app is displayed in the browser window, and the application output is sent to the command line or to your terminal.
12 Creating an extension from your app

After you deploy and test your app, you must export it as an extension to deploy it in a production environment.

About this task

The IBM QRadar GUI Application Framework SDK deploy command is intended for use only in test environments. To create an extension that you can use in a production environment, you must use the QRadar Content Management Tool to export your application as an extension.

Procedure

1. Use the QRadar GUI Application Framework SDK to package and deploy your application to your QRadar Console test environment. Make sure that you obtain the application ID for your app from Application Creation Task state output that is returned when you issue the qradar_app_creator deploy command.

2. Use SSH to log in to QRadar Console where your app is running as root user.

3. Type the following command at the command line:

   /opt/qradar/bin/contentManagement.pl --action export --content-type 100 --id <application_ID>

   This example shows an app that has an ID of 1023.

   /opt/qradar/bin/contentManagement.pl --action export --content-type 100 --id 1023

   The extension is created as a .zip file in the /opt/qradar/bin folder. The file name has the following format:

   installed_application-ContentExport-YYYYMMDDhhmmss.zip

   Here is an example:

   installed_application-ContentExport-20160911141607.zip

What to do next

Download the .zip file from the /opt/qradar/bin folder on your QRadar Console. You can then use QRadar Extensions Management on the Admin tab to install your extension on your QRadar Console.

For more information about exporting apps and content in extensions, and Extensions Management, see the QRadar SIEM Administration Guide.

Adding multiple apps in an extension

Package and export multiple apps and other content in an extension so users can download related apps or content in one file.

About this task

You can include multiple apps or content types in an extension, which you export as a compressed (.zip) file.

Procedure

1. Use SSH to log in to QRadar as the root user.

2. To get a list of the content types and search parameters, type the following command:

   /opt/qradar/bin/contentManagement.pl -h export
In the following example, you search for apps that are represented by the content type ID of 100, and the regex .*, which matches everything.

```
/opt/qradar/bin/contentManagement.pl --action search -c 100 -r .*
```

You must specify the content type (-c) and a search regex (-r).

In the following example, you search for the custom content type "dashboard" that is represented by the content type ID of 4, and the regex .*, which matches everything for dashboards content.

```
/opt/qradar/bin/contentManagement.pl --action search -c 4 -r .*
```

Use regex to narrow your search, for example, you use the following search to find dashboards content that includes 'Threat' in the name.

```
/opt/qradar/bin/contentManagement.pl --action search -c 4 -r Threat
```

3. Add the content type IDs or string and the IDs of the apps or content in a text file. The content type can be represented by the string or ID. Use the following format:

```
<content_type_ID_or_string>,<Content_or_app_ID>
```

Use the following rules to create the package text file:

- Use a separate line for each content type.
- Make sure that the first value that you enter on a line is the content type.
- Make sure that the value that follows the content type is the ID of the app or the content.
- Use commas to separate values.

Here's an example of packaging an app that has a content type of 100 and a dashboard that has a content type of 4.

```
4,22
100,1051
```

Here's the same example where the string is used instead of the content type ID to represent the content type.

```
dashboard,22
installed_application,1051
```

In the following example, strings are used for the content type, which are followed by the content IDs.

```
installed_application,1001
customrule,1274,1275
dashboard,10
```

4. Save the text file as `<my_package>.txt`

5. Type the following command to assemble and export your content in an extension file:

```
/opt/qradar/bin/contentManagement.pl -a export -c package -f `<mypackage>.txt`
```

The extension is created as a .zip file in the `/opt/qradar/bin` folder. The file name has the following format:

```
<file_name>-ContentExport-YYYYMMDDhhmmss.zip
```

### What to do next

Download the compressed (.zip) file from the `/opt/qradar/bin` folder on your QRadar Console. You can then use QRadar Extensions Management on the Admin tab to install your extension on your QRadar Console.

For more information about using the `contentManagement.pl` script to export content, and about Extensions Management, see the QRadar SIEM Administration Guide.

### IBM QRadar content extensions

Use content extensions to update QRadar security template information or add new content such as rules, reports, searches, logos, reference sets, custom properties.
Types of QRadar content extensions

All apps and content extensions are hosted on the [IBM X-Force Exchange](https://exchange.xforce.ibmcloud.com/), where you can filter by content type such as custom AQL function or custom property. You can use content extensions can be used in conjunction with apps.

The following table describes the types of content extension that you can deploy in QRadar.

<table>
<thead>
<tr>
<th>Content extension type</th>
<th>Enhancement type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dashboard</td>
<td>Data</td>
<td>An associated set of dashboard items, which you view on the Dashboard tab in QRadar. Dashboard items are widgets are visual representations of saved search results.</td>
</tr>
<tr>
<td>Reports</td>
<td>Data/Functionality</td>
<td>Templates for reports that are built upon saved event or flow searches. Generate on-demand reports or schedule them to run at repeating intervals.</td>
</tr>
<tr>
<td>Saved searches</td>
<td>Data</td>
<td>A set of search criteria (filters, time window, columns to display or group data by). By saving the criteria of commonly run searches, you don’t need to define them repeatedly. Saved searches are required for reports and dashboards.</td>
</tr>
<tr>
<td>FGroup</td>
<td>Data</td>
<td>A group of similar items by type, such as a group of log sources, a group of rules, a group of searches, or a group of report templates. FGroups are used as organizational units.</td>
</tr>
<tr>
<td>Custom rules</td>
<td>Data</td>
<td>A set of tests that are run against events or flows that enter the system. The rule is triggered when the tests match the input. Rules can have responses which are actions that are triggered when the rule is triggered. Responses can include actions such as generating an offense, generating a new event, sending an email, annotating the event, or adding data to a reference data collection.</td>
</tr>
<tr>
<td>Custom properties</td>
<td>Data</td>
<td>Defines a property that is extracted or derived from an inbound event or flow. Can be based on a regular expression that extracts a subset of a particular event or flow payload as a textual property. They can be based on calculations, and perform an arithmetic operation on existing numeric properties of the event or flow.</td>
</tr>
<tr>
<td>Log source</td>
<td>Data</td>
<td>A representation of a source of events such as a server, mainframe, workstation, firewall, router, application, or database. Any events that enter QRadar and originate from that source are attributed to the log source. Log sources contain the configuration information that is needed to receive inbound events, or to pull event data from the event source. Log sources contain information that is specific to your environment such as IP address or host name and other possible configuration parameters.</td>
</tr>
<tr>
<td>Log source extensions</td>
<td>Data</td>
<td>A parsing logic definition that is used to synthesize a custom DSM for an event source for which there is no existing DSM. Use log source extensions to enhance or override the parsing behavior of an existing DSM.</td>
</tr>
<tr>
<td>Custom QID map entries</td>
<td>Data</td>
<td>A combination of Event name, Event description, Severity, and Low-level category values that are used to represent a particular type of event that a log source might receive. Custom Qid map entries are created to supplement the default QID map that QRadar provides for events that are not officially supported by QRadar.</td>
</tr>
<tr>
<td>Content extension type</td>
<td>Enhancement type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Reference Data Collection</td>
<td>Data</td>
<td>A container definition that is represented as either a set, a map, a map of sets, a map of maps, or a table for holding reference data. Searches and rules can reference Reference data collections.</td>
</tr>
<tr>
<td>Historical Correlation Profile</td>
<td>Data</td>
<td>A combination of a saved search and a set of one or more rules. Use historical correlation profiles to test rules by rerunning a set of historical events through an offline version of the custom rule engine that has a subset of rules enabled.</td>
</tr>
<tr>
<td>Custom Functions</td>
<td>Functionality</td>
<td>A SQL-like function (defined in JavaScript) that you can use in an Advanced search to enhance or manipulate data</td>
</tr>
<tr>
<td>Custom Actions</td>
<td>Functionality</td>
<td>A custom response for a rule to run, when the rule is triggered. Custom actions are defined by a Python, Perl, or Bash script that can accept arguments from the event or flow data that triggered the rule.</td>
</tr>
</tbody>
</table>

For more information about content management, see the *IBM QRadar Administration Guide*

**Extensions management**

Use the Extensions Management tool on the IBM QRadar Console Admin tab to manage your extensions in your deployment.

Use the QRadar Content Management Tool to export your application as an extension. Then, use the Extensions Management tool on the Admin tab to upload, install, uninstall, and delete it. For more information about content management and extensions, see the *IBM QRadar Administration Guide*.

**About extensions**

Before you install an extension, the content items are compared to content items that are already in the deployment. If the content items exist, you can choose to overwrite them or to keep the existing data.

When you uninstall an extension, apps that are included in the extension are automatically uninstalled. Content items that are included in the extension must be removed manually.

After the extension is added, a yellow caution icon in the Status column indicates potential issues with the digital signature. Hover the mouse over the triangle for more information. Extensions that are unsigned or are signed by the developer, but not validated by your vendor, might cause compatibility issues in your deployment.
13 Sample apps

Build apps in Python that integrate with, and add extra functions to, QRadar.

The mini-tutorials for the sample apps help to explain the basics of what you need to know to build and integrate your own apps into QRadar.

Sample apps are available:

- com.ibm.AppMan.1.0.0.zip creates an icon on the QRadar Admin tab, and opens an HTML page when you click it that shows all of the apps that are currently installed.
- com.ibm.BugzillaOffenses.0.1.0.zip polls QRadar for offenses and pushing those offenses as tickets to the Bugzilla REST API.
- com.ibm.configpage-with-image.1.0.0.zip creates an icon on the QRadar Admin tab and opens an HTML page when you click it.
- com.ibm.HelloWorld.1.0.0.zip creates a new tab in the QRadar UI with the text "Hello World!". This simple app is also created when you use the SDK to create a development environment.
- com.ibm.si.HelloWorldG11n.1.0.0.zip a globalized version of the simple Hello World app.
- com.ibm.IPMetadataProvider.1.0.0.zip demonstrates how to add IP metadata to QRadar tooltips.
- com.ibm.IPMetadataProviderWithImage.1.0.0.zip - an app that demonstrates how to add IP address metadata and images to QRadar tooltips.
- com.ibm.MyFirstRESTApp.1.0.0.zip calls a REST API and add the data to a dashboard item.
- com.ibm.OffenseVisualizer.1.0.0.zip displays offenses on the QRadar Offenses tab.
- com.ibm.si.dashboard-example.1.0.0.zip adds a dashboard item to the QRadar Dashboard tab.
- com.ibm.si.dashboard-with-images.1.0.0.zip adds a dashboard item with images to the QRadar Dashboard tab.
- com.ibm.si.multi-components-example.1.0.0.zip adds a tab, a new toolbar button in QRadar, and an icon on the Admin tab.
- com.ibm.si.offense-toolbar-button.1.0.0.zip adds a button to the QRadar Offenses tab toolbar that opens a JavaScript alert dialog that contains offense data.
- com.ibm.store.1.0.0.zip creates a new tab in QRadar that saves the data that is entered in a database within the docker container.

Dashboard item example

You can use IBM QRadar GUI Application Framework to add a dashboard item to your QRadar dashboard.

You might use dashboards to display data that you want to view or use often, for example, you might want to monitor disk usage on your QRadar appliances.

The sample dashboard item app adds a basic dashboard item to the QRadar Dashboard tab.

The following image is a dashboard example that is created by an app in QRadar.
On the **Dashboard** tab, the sample dashboard item is accessed by using the **Add Item** menu.

The following image shows the sample dashboard in the **Add Item** menu.

The dashboard sample app contains the files that are described in the following table:

<table>
<thead>
<tr>
<th>Files/Folders</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>app</strong></td>
<td>The root directory for application files. The app folder contains the following files: qpylib contains the Python library files that your application uses to connect to the QRadar API endpoints. <strong>init</strong>.py - a sample initialization that creates a Flask instance, imports views from views.py script and functions from the qpylib library. views.py the main entry point into the web application. This file and the manifest.json file are the only files that are required in every app. Contains sample code for the Dashboard example app. The /static/sampleDashboardItemResponse.json file contains the JSON object that contains the dashboard ID, title, and HTML string.</td>
</tr>
<tr>
<td><strong>qradar_appfw_venv</strong></td>
<td>Contains the Python virtual environment where the dependencies are installed.</td>
</tr>
<tr>
<td><strong>init</strong>.py</td>
<td>Creates an instance of the Flask micro-framework that is used to serve content to QRadar.</td>
</tr>
<tr>
<td><strong>manifest.json</strong></td>
<td>Describes details about the sample Dashboard Example, which QRadar uses.</td>
</tr>
<tr>
<td><strong>run.py</strong></td>
<td>Contains instructions to run the code that is in the /app sub directory.</td>
</tr>
</tbody>
</table>

**manifest.json**

The manifest.json file contains the following code:

```
{
    "name":"DashBoard Example",
    "description":"Application to display a new dashboard item",
    "version":"1.0",
    "uuid":"558d7935-f00b-42da-a27b-c82adbd12d21",

    "dashboard_items": [
    {
       "text":"QDashBoardExample Item",
       "description":"Sample dashboard item that is a copy of most recent offenses",
       "rest_method":"sampleDashboardItem",
       "required_capabilities": ["ADMIN"]
    }
    ],
    "rest_methods": [
    {
       "name":"sampleDashboardItem",
       "url":"/static/sampleDashboardItemResponse.json",
    }
    ]
}
```

58 IBM Security QRadar Application Framework: Developer Quick Start Guide
The first four objects, name, description, version, and uuid, provide basic application information.

The dashboard_items object describes a new item on the QRadar Dashboard tab. These items are available to users, who can manually add the items to their dashboard.

The dashboard_items block contains the fields that are described in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The name of the dashboard item that is displayed.</td>
<td>QDashBoardExample Item</td>
</tr>
<tr>
<td>description</td>
<td>A description of the dashboard item that is displayed when your mouse hovers over the dashboard item.</td>
<td>Sample dashboard item that is a copy of most recent offenses</td>
</tr>
<tr>
<td>rest_method</td>
<td>The name of the REST method to load this item. This method must be declared in the rest_methods section of the manifest.</td>
<td>sampleDashboardItem</td>
</tr>
<tr>
<td>required_capabilities</td>
<td>Instructs QRadar to display the Dashboard Example dashboard item only to users who have administrator privileges.</td>
<td>[&quot;ADMIN&quot;]</td>
</tr>
</tbody>
</table>

The rest_methods block contains the fields that are described in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A unique name for this REST method within the app.</td>
<td>sampleDashboardItem</td>
</tr>
<tr>
<td>url</td>
<td>The URL to access the REST method, relative to the application root. Only URLs within their own application are supported.</td>
<td>/static/sampleDashboardItemResponse.json</td>
</tr>
<tr>
<td>method</td>
<td>Concise text to display that describes the area. Can optionally point at a resource bundle key, if the application is globalized.</td>
<td>GET</td>
</tr>
<tr>
<td>argument_names</td>
<td>The names of arguments that this method supports. Arguments are passed to the method URL encoded, as either query string parameters or in the PUT/POST body.</td>
<td>[]</td>
</tr>
<tr>
<td>required_capabilities</td>
<td>This field instructs QRadar to display the Dashboard Example dashboard item only to users with Administrator privileges.</td>
<td>[&quot;ADMIN&quot;]</td>
</tr>
</tbody>
</table>
views.py

For this sample app, creates the default routes '/index', both of which return a simple string.
The index route is declared in the url field of manifest.json.
__author__ = 'IBM'

from app import app

def index():
    return ''

/app/static/sampleDashboardItemResponse.json

The /app/static/sampleDashboardItemResponse.json file contains the following code:
{"id":"sampleDashboardItem","title":"Sample Dashboard Item",
 "HTML":"<div>This item could contain <b><u>any HTML!</u></b>!</div>"

The JSON object that is returned needs the following data:
• An ID for the dashboard item you are creating
• A title for the dashboard item
• HTML to render the dashboard item

Page script / toolbar button example

The page script / toolbar button example adds a button to the toolbar on the Offenses or Offense Summary tabs in IBM QRadar. The new button calls a page script when you click it.

When you select an offense in the Offenses or Offense Summary tab, and click the button, information about the selected offense is displayed in an alert.

The following image shows the My Offense Button.

The following image shows the output from using the My Offense Button.

The following table shows the files that are used for this sample app:
Table 23. Sample app files

<table>
<thead>
<tr>
<th>Files/Folders</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>The root directory for application files. The app folder contains the following files:</td>
</tr>
<tr>
<td>qpylib folder contains the Python library files that your application can use to connect with QRadar API endpoints.</td>
<td></td>
</tr>
<tr>
<td><strong>init</strong>.py - a sample initialization file your app. It creates a Flask instance, imports views from views.py and functions from the qpylib library.</td>
<td></td>
</tr>
<tr>
<td>views.py - The main entry point into the web application. This file and the manifest.json file are the only files that are required in every application. This file contains sample code for the sample application.</td>
<td></td>
</tr>
<tr>
<td>The /static/js/custom_script.js script that is run when the button is clicked.</td>
<td></td>
</tr>
<tr>
<td><strong>init</strong>.py</td>
<td>Creates an instance of the Flask micro-framework that is used to serve content to QRadar.</td>
</tr>
<tr>
<td>manifest.json</td>
<td>This file tells QRadar what the sample app does.</td>
</tr>
</tbody>
</table>

manifest.json

The manifest.json file contains the following code:

```
{
    "name":"Page script test App",
    "description":"An example of to test page scripts",
    "version":"1.0",
    "uuid":"4a5d50cc-b9f1-4526-b356-5cb2d60e9467",
    "rest_methods": [
        {
            "name":"offenseListFunction",
            "url":"/offenseListFunction",
            "method":"GET",
            "argument_names":["context"]
        }
    ],
    "gui_actions": [
        {
            "id":"OffenseListToolbarButton",
            "text":"My Offense Button",
            "description":"My Offense Button",
            "icon":"
            "rest_method":"offenseListFunction",
            "javascript":"my_offense_toolbar_button_action(result)",
            "groups":["OffenseListToolbar"]
        },
        {
            "id":"OffenseSummaryToolbarButton",
            "text":"My Offense Button",
            "description":"My Offense Button",
            "icon":"
            "javascript":"my_offense_toolbar_button_action(context)",
            "groups":["OffenseSummaryToolbar"]
        }
    ],
    "page_scripts": [
        {
            "app_name":"SEM",
            "page_id":"OffenseList",
            "scripts":["static/js/custom_script.js"]
        }
    ]
}```
The first three objects, name, description, and version, provide basic app information.

The gui_actions object describes a new GUI Action that the user can perform in the QRadar user interface. It is abstracted from the underlying representation.

GUI Actions are represented as buttons on page toolbars, or as right-click menu options.

GUI Actions can run a block of JavaScript, invoke a REST method, or both.

The gui_actions block contains sections for both the Offenses page toolbar and the Offense Summary page toolbar.

The Offenses page toolbar section contains the fields that are described in the following table:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>A unique ID for this area within the application</td>
<td>OffenseListToolbarButton</td>
</tr>
<tr>
<td>text</td>
<td>The name of the GUI Action that is displayed in the user interface.</td>
<td>My Offense Button</td>
</tr>
<tr>
<td>description</td>
<td>A description of the GUI Action that is displayed when your mouse hovers over the item.</td>
<td>My Offense Button</td>
</tr>
<tr>
<td>icon</td>
<td>A URL to load, relative to the application root. Only URLs that exist within the app can be referenced.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>rest_method</td>
<td>The name of the REST method to load this item. This method must be declared in the rest_methods block of the manifest.</td>
<td>offenseListFunction</td>
</tr>
<tr>
<td>javascript</td>
<td>A JavaScript code block to run when this action is performed. Either this argument or the REST method argument is required. If both are specified, then the REST method is run first, the results are then passed into the JavaScript code block by using the variable result. If only the JavaScript argument is specified, a context variable that contains the context of the GUI Action, is passed into the JavaScript code block.</td>
<td>my_offense_toolbar_button_action(result)</td>
</tr>
<tr>
<td>groups</td>
<td>A list of one or more GUI Action groups to install the action into. This string is the identifier of the toolbar or right-click menu in QRadar. At least one group must be provided. You can also use a group name in this format ariel:&lt;FIELD_NAME&gt;, where &lt;FIELD_NAME&gt; is the name of a field in the QRadar Event or Flow viewer. If this field is specified, the action is installed into the menu of that field, and the context parameter is the contents of the field.</td>
<td>[&quot;OffenseListToolbar&quot;]</td>
</tr>
</tbody>
</table>
The Offenses Summary page toolbar section contains the fields that are described in the following table:

**Table 25. Offenses summary page toolbar fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>A unique ID for this area within the application</td>
<td>OffenseSummaryToolBarButton</td>
</tr>
<tr>
<td>text</td>
<td>The name of the GUI Action that is displayed in the user interface.</td>
<td>My Offense Button</td>
</tr>
<tr>
<td>description</td>
<td>A description of the GUI Action that is displayed when your mouse hovers over the item.</td>
<td>My Offense Button</td>
</tr>
<tr>
<td>icon</td>
<td>A URL to load, relative to the application root. Only URLs that exist within the QRadar app can be referenced.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>javascript</td>
<td>A JavaScript code block to run when this action is performed. Either this argument or the REST method argument is required. If both are specified, then the REST method is run first, the results are then passed into the JavaScript code block by using the variable <code>result</code>. If only the JavaScript argument is specified, a <code>context</code> variable that contains the <code>context</code> of the GUI Action, is passed into the JavaScript code block.</td>
<td>my_offense_toolbar_button_action(context)</td>
</tr>
<tr>
<td>groups</td>
<td>A list of one or more GUI Action groups to install the action into. This string is the identifier of the toolbar or right-click menu in QRadar. At least 1 group must be provided. You can also use a group name in this format <code>ariel:&lt;FIELD_NAME&gt;</code>, where <code>&lt;FIELD_NAME&gt;</code> is the name of a field in the QRadar Event or Flow viewer. If this field is specified, the action is installed into the menu of that field, and the <code>context</code> parameter is the contents of the field.</td>
<td>&quot;[&quot;OffenseSummaryToolbar&quot;]&quot;</td>
</tr>
</tbody>
</table>

The `rest_methods` block contains the fields that are described in the following table:

**Table 26. rest_methods block fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A unique name for this REST method within the application.</td>
<td>offenseListFunction</td>
</tr>
<tr>
<td>url</td>
<td>The URL to access the REST method, relative to the application root. Only URLs within the app are supported.</td>
<td>/offenseListFunction</td>
</tr>
<tr>
<td>method</td>
<td>The HTTP method on the named endpoint (GET/POST/DELETE).</td>
<td>GET</td>
</tr>
<tr>
<td>argument_names</td>
<td>The names of arguments that this method expects. Arguments that are passed to the method URL are encoded, as either query string parameters or in the PUT/POST body.</td>
<td>[&quot;context&quot;]</td>
</tr>
</tbody>
</table>

The `page_scripts` block contains the fields that are described in the following table:
Table 27. Page scripts block fields

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_name</td>
<td>The name of the QRadar application you want to include the script into. The wildcard &quot;*&quot; is also supported if it is used with the page_id field, to have a file included in every QRadar page.</td>
<td>SEM</td>
</tr>
<tr>
<td>page_id</td>
<td>The ID of the QRadar page where the script is included. The wildcard &quot;*&quot; is also supported when it is used with the app_name field. When the wildcard character is used, a file is included on every QRadar page.</td>
<td>OffenseList</td>
</tr>
<tr>
<td>scripts</td>
<td>The scripts that you want to include, relative to the application root.</td>
<td>&quot;static/js/custom_script.js&quot;</td>
</tr>
</tbody>
</table>

views.py

The views.py for this example establishes the Flask routes that are used by the application. The script injects context information from the REST API endpoint into the offense variable. This content is used by the page script.

```python
__author__ = 'IBM'

from flask import request
from app import app
from qpylib import qpylib
import json

@app.route('/offenseListFunction', methods=['GET'])
def offenseListFunction():
    qpylib.log("offenseListFunction", "info")
    offense = request.args.get("context")
    qpylib.log("context" + offense, "info")

    # You can process the data and return any value here, # that will be passed into javascript
    return json.dumps({'context_passed_to_python_route':offense})
```

static/js/custom_script.js

Contains the function that turns the content of the offense variable into a string and displays it within a JavaScript alert dialog.

```javascript
function my_offense_toolbar_button_action(offense)
{
    alert(JSON.stringify(offense));
}
```

Script files must be fragments and not fully formed HTML because they are injected into a fully rendered page.

**Note:** Be careful how you name functions because apps can share scripts from QRadar. It is good practice to add a prefix to the JavaScript function such as the app name.

Page script samples

You can use page scripts to link between apps, select rows in a QRadar table, and get the IP address of an asset.
How to link between apps

You use the `setActiveTab(tabId, url)` function from the qradar.js library to link between two apps that are installed on the same QRadar Console.

For example, you want to link from MyApp1 to MyApp2. MyApp1 uses the following HTML template with a div tag that calls a JavaScript function:

```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>myApp1</title>
  <script type="text/javascript" src="static/js/myApp2.js"></script>
</head>

<body>
  <div onclick="somejs();">
    Link to myApp2
  </div>
</body>
</html>
```

The `somejs()` function in the `static/js/myApp2.js` file uses the `setActiveTab(tabId, url)` function from the qradar.js library:

```javascript
function somejs() {
  alert('set active tab');
  top.setActiveTab("myApp2_1057","plugins/1057/app_proxy/index");
}
```

The `tabId` parameter uses the following format: `areasID_applicationID`. The areas ID is the value of the `id` field in the `areas` block of the `myApp2.manifest.json` file.

The `url` parameter uses the following format:

`plugins/<applicationID>/app_proxy/<areasURL>`

The `applicationID` is assigned when you install your app. You can use the `POST /gui_app_framework/application_creation_task` endpoint to retrieve the `applicationID`.

The `areasURL` is the value of the `url` field in the `areas` block of the following `myApp2.manifest.json` file.

```json
"areas": [  
  {  
    "id": "myApp2",  
    "text": "myApp2",  
    "description": "The app I am linking to",  
    "url": "index",  
    "required_capabilities": ["ADMIN"]  
  }],
```

How to get the selected rows in a table

To get all the selected rows in a table on a QRadar page, use the following code in your page script:

```javascript
var selectedIds = getSelectedRowIds();
var grid = getGrid();
var store = grid.store;
var row = grid.row(selectedIds[0], true );
```
How to get the IP address of the selected asset from the Asset page

To get the IP address of the selected asset from the QRadar Asset tab use the following code in your page script:

```javascript
var assetId = selectedRows[0].id;
```

Passing context-specific information to a page script

You use a GUI Action to pass information about offenses, assets, vulnerabilities from a QRadar table to a page script.

For example, you can extract selected information from QRadar and pass it to a script for further processing. You can create a simple app that allows users to select offenses that are listed as rows in the table on the Offenses tab and pass that information to a page script. By clicking a button on the Offenses tab toolbar, a JavaScript alert that contains the extracted information is displayed.

You can also pass information about vulnerabilities, network activity, assets, and information from any table in most QRadar pages.

The following manifest.json file example shows details of entries that you make in this file:

manifest.json

In the app's manifest.json file, the REST method that is used by the views.py script is defined. The REST method also specifies that application name, page ID, and application context must be defined.

appName

The application name is defined in the page_scripts block and indicates to QRadar which tab contains the data to be passed to the script.

pageId

The page ID is defined in the page_scripts block and indicates to QRadar which page within the tab contains the data to be passed to the script.

appContext

The application context refers to the row or rows that are manually selected by the user on the tab and page that is defined by the application name and page ID. In this case, it is the table on the Offense List page on the Offenses tab. Each row contains data on a particular offense and it is this data that is passed to the custom script.

```json
{
    "name":"offense log pass ids",
    "description":"An example of passing ids",
    "version":"1.0",
    "uuid":"a4095969-1c88-4e35-aecb-4eea7b061cd3",
    "rest_methods": [1]
    {
        "name":"listFunction",
        "url":"/listFunction",
        "method":"GET",
        "argument_names":["appName","pageIndex","appContext"]
    }
}
```

```json
"gui_actions": [2]
{
    "id":"OffenseListToolbarButton",
    "text":"Offense pass Ids !",
    "description":"Pass Ids for offenses !",
    "icon":"static/images/bookmarks_small.png",
    "rest_method":"listFunction",
    "javascript":"my_toolbar_button_action(result)"
}
```
The following list describes the contents in the code snippet from the manifest.json file.

1. Use the `rest_methods` block to define the API name, `app.route` URL, and API method that you add to `views.py`.

2. Use the argument names to precisely locate the data that is passed to the script. The app name and page ID values are defined in the QRadar tab and page. In this case, the Offenses tab and main Offense List page.

   The `appContext` argument is a placeholder that holds the actual offense data that is selected when the user clicks a row or rows in the Offense List table. This data is passed to JavaScript for processing.

3. Use the `gui_action` block to define the button that passes data to the screen when the user clicks it.

4. Use the `rest_methods` block to define the REST methods that are used in the `views.py` script to list the data from each selected row.

5. Use the JavaScript function to create an alert dialog that displays the JSON string that is passed by the `appContext` argument from the listFunction method.

6. Use the GUI Action group location to define where the button appears. In this case, the button appears on the main Offense List page toolbar.

7. Use the `page_scripts` block for the `app_name` and `page_id` arguments that are passed to the REST API method, and for the location of the JavaScript file that processes the data.

### app/views.py

The app's `app/views.py` defines the application route and function that retrieves the `appContext` data and passes it as JSON to the custom script.

```python
__author__ = 'IBM'

from flask import render_template, request
from app import app
from qpylib import qpylib
import json

@app.route('/

@app.route('/index')
def index():
    return render_template("index.html", title = "Offense Context App!")

@app.route('/listFunction', methods=['GET'])
def listFunction():
    qpylib.log("listFunction", "info")
    rows = request.args.get("appContext")
    qpylib.log("selectedRows" + rows, "info")

    #You can process the data and return any value here,
    #It is passed into JavaScript.
    return json.dumps({'context_passed_to_python_route':rows})
```

The following list describes the contents in the code snippet from the `views.py` script.

13 Sample apps  67
1. The function URL and REST method that is defined in the `rest_methods` block of the app manifest file.

2. The row variable that gets the content of the selected rows by using the appContext argument.

3. The Flask `json.dumps` method formats the data that is contained in the `rows` variable as JSON. This data is passed to the `custom_script.js` file that displays it in a simple alert.

```javascript
app/static/js/custom_script.js

The following short JavaScript creates an alert and formats the JSON content into strings for each row.

```javascript
function my_toolbar_button_action(offense) {
    alert(JSON.stringify(offense));
}
```

### Context-specific metadata provider example

Metadata providers are used to show extra context-sensitive information in IBM QRadar.

Metadata is displayed when the user's mouse pointer hovers over an item in the user interface. For example, you might want to know the user name that is associated with an IP address, which you can view by hovering your mouse pointer over the IP address.

The following image shows an example of context-sensitive information that is displayed when you hover your mouse pointer over an IP address in the table.

You can create custom metadata providers for three types of metadata:

- **ip** displayed for an IP address throughout QRadar.
- **userName** displayed for a user name throughout QRadar.
- **ariel:<FIELD_NAME>**, where `<FIELD_NAME>` is the name of a field in the QRadar Event or Flow viewer - displayed for a field in the event or flow viewer.

The following example demonstrates how to write a simple app that provides extra metadata when you hover your mouse pointer over IP addresses in QRadar.

```json
manifest.json

The manifest.json file contains the following code:
```
The following table describes the block fields that are included in the `Manifest.json` file.

**Table 28. Manifest.json file block fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rest_methods</code></td>
<td>Describes the rest method that is used to return the additional metadata.</td>
</tr>
<tr>
<td><code>metadata_providers</code></td>
<td>Describes the metadata type and the rest method that returns the additional metadata.</td>
</tr>
</tbody>
</table>

The manifest file shows that the app provides metadata for QRadar IP addresses that use the REST method `getIPMetadata`. This REST method is exposed by the `/ip_metadata_provider` endpoint within the app. This endpoint represents a GET request and expects an argument `context`. The context argument is appended to the GET request as a query string that contains the context-specific IP address (`/ip_metadata_provider?context=127.0.0.1`).

**views.py**

The `views.py` script contains the following code:

```python
import json
from app import app
from flask import render_template, request
from qpylib import qpylib

@app.route('/ip_metadata_provider', methods=['GET'])
def getIPMetadata():
    context = request.args.get('context')

    metadata_dict = {
        'key': 'exampleIPMetadataProvider',
        'label': 'Extra Metadata:',
        'value': 'Metadata value',
        'html': render_template('metadata_ip.html', ip_address=context)
    }

    return json.dumps(metadata_dict)
```

The `views.py` file shows a simple implementation for the `getIPMetadata` REST method that was defined previously in the `manifest.json`. As noted, the context is passed to the endpoint by a query string. Use
the Flask request module to parse out the value of the context query. The value of the context query is parsed by using the script: `context = request.args.get('context')`. This variable contains the context-specific IP address.

Metadata providers are required to return JSON with the fields that are shown in the following table:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>Unique key for the metadata provider</td>
</tr>
<tr>
<td>label</td>
<td>Description of the metadata, which is displayed in a pop-up layer in QRadar.</td>
</tr>
<tr>
<td>value</td>
<td>Plain text context-sensitive data to be provided</td>
</tr>
<tr>
<td>html</td>
<td>HTML context-sensitive data to be provided.</td>
</tr>
</tbody>
</table>

In this example, a Python dictionary object is created to contain these fields. For the `html` field, a template file, `metadata_ip.html`, is created in the `/app/templates` directory. The context that is retrieved from the query string as the variable `ip_address` is passed to it.

`metadata_ip.html`

In this example, you render the context-specific IP address. The `metadata_ip.html` file contains a Jinja template string that is replaced with the value that is passed in from the `render_template` call in `views.py`.

```html
{{ ip_address }}
```

**Add right-click functionality**

Add right-click functionality to a tab your app created in IBM QRadar.

This example shows how to use the right-click GUI Action to capture an IP address and pass the information to custom JavaScript. Use the right-click menu on the QRadar Log Activity tab to capture an IP address of an event. Pass the IP address to a custom tab. Use a button on the custom tab to initiate a search for events that contain the captured IP address.

`manifest.json`

```json
...  
  "areas": [
    {
      "url": "index",
      "text": "RtClick",
      "required_capabilities": ["ADMIN"],
      "id": "QRtClick",
      "description": "An app to POC Right Click"
    }
  ],
  "gui_actions": [
    {
      "id": "rtClickEventIP",
      "text": "Get row info from right click",
      "description": "Right click on a row, get all the info",
      "icon": null,
      "rest_method": "rtgetcontext",
      "javascript": "clickme(result)",
      "groups": ["ipPopup"],
      "required_capabilities": ["ADMIN"]
    }
  ]
...
```

IBM Security QRadar Application Framework: Developer Quick Start Guide
"rest_methods" : [ 5
    {  
      "name":"rtgetcontext",  
      "url":"/getcontext",  
      "method":"GET",  
      "argument_names":["context"]  
    }
  ],

"page_scripts" : [ 6
  {  
    "app_name":"EventViewer",  
    "page_id":"EventList",  
    "scripts" : ["static/clickme.js"]  
  }
]...

app/views.py
__author__ = 'IBM'

from app import app
from flask import jsonify, request, render_template
import json
from qpylib import qpylib

@app.route('/')
@app.route('/index')
def index():
    other_data = request.args.get("otherdata");
    context = request.args.get("context");
    if context is None:
        context = ""
    if other_data is None:
        other_data = ""
    qpylib.log("Displaying context" + str(context));
    return render_template("index.html", context=context, other=other_data)

@app.route('/getcontext', methods=['GET'])
def get_context():
    context = request.args.get("context")
    qpylib.log("Setting the results to: " + context)
    return json.dumps({"app_id":qpylib.get_app_id(),"context":context})

app/static/clickme.js
function clickme(result) {
    var app_id = ""
    var context = ""
    if (result) {
        app_id = encodeURI(result.app_id)
        context = encodeURI(result.context)
    }
    var d = new Date();
var n = d.getTime();

var otherData = "Something passed from Javascript"

console.log("Hey, you right clicked on me");
console.log(result)

app/templates/index.html
<html>
<body>
<script>
var gotoTab = function() {
    var url = "/console/do/arielSearch?appName=
EventViewer&pageId=EventList&dispatch=performSearch&value(searchMode)=
AQL&searchOrigin=SEARCH_RESULTS_AQL&value(timeRangeType)=
aqlTime&value(interval)=300000&value(searchName)
=value(searchId)=null&value(aql)

=select%20*%20from%20events%20where%20destinationip%20%3D%20%27{{context}}%27%20LAST%2012%20HOURS&value(aql)

=50%22select%20*%20from%20events%20where%20destinationip%20%3D%20%27{{context}}%27%20LAST%2012%20HOURS%22&value(recordsLimit)="top.setActiveTab("EventViewer", url )

</script>

<div>
<ul>
<li>Received context data from QRadar: {{context}}</li>
<li>Received other data from Javascript: {{other}}</li>
<li><button onclick="gotoTab()">Search for events with sourceip of {{context}}</button>
</li>
</ul>
</div>
</body>
</html>

Custom fragments example

The custom fragments feature allows an app to inject its own content into QRadar tabs and pages. The application determines what the content is and how it is rendered. The injection points are fixed, predetermined locations within the QRadar page set.

The sample app that is presented here injects an HTML table that contains data for an offense on the Offense Summary page. This example uses app/qpylib/offense_qpylib.py library functions to retrieve offense details in JSON format.

The following image shows custom content at the top of the Offense Summary page:
The following sections describe the manifest.json and app/views.py code that is used to inject offense data into the Offense Summary page header.

**manifest.json**

Use the fragments block in the application manifest file to define the location where you want to inject the content. You also define the REST endpoint that is used to retrieve the content in the fragments.

```json
{
  "name": "Offense Fragment New",
  "description": "Render offense using custom python",
  "version": "1.0",
  "uuid": "a4095969-1c88-4e35-aecb-4eea7b061ab4",
  "fragments": [
    {
      "app_name": "SEM",
      "page_id": "OffenseSummary",
      "rest_endpoint": "fragoffense"
    }
  ]
}
```

**Figure 5. Custom content injection**

The following sections describe the manifest.json and app/views.py code that is used to inject offense data into the Offense Summary page header.
The following list describes the contents in the code snippet from the **fragments** block.

1. The **app_name** and **page_id** fields define the tab and page where the content is injected. As content can be injected into the header of the Offense Summary page only, no **location** field is required.

2. The **fragoffense** rest endpoint is defined in the **app/views.py** file.

```python
@app.route('/fragoffense/<offense_id>', methods=['GET'])
def get_offense(offense_id):
    try:
        offense_json = get_offense_json_html(offense_id, custom_html_generator)
        return Response(response=offense_json, status=200, mimetype='application/json')
    except Exception as e:
        log('Error ' + str(e))
        raise

def custom_html_generator(offense_json):
    return ('<table><tbody>
    <tr><td><strong>Offense ID</strong></td><td>' + str(offense_json['id']) + '</td></tr>
    <tr><td><strong>Source IP</strong></td><td>' + str(offense_json['offense_source']) + '</td></tr>
    <tr><td><strong>Severity</strong></td><td>' + str(offense_json['severity']) + '</td></tr>
    </tbody></table>')
```

The following list describes the contents in the code snippet from the **views.py** script.

1. The **get_offense_json_html** function that is imported from the **app/qpylib/offense_qpylib.py** library retrieves the details of the offense ID in **JSON** format.

2. The **get_offense** function has an **@app.route** annotation that defines the endpoint's route. Its value is composed of the manifest's **rest_endpoint** field value /fragoffense, and the context information. In this case, the context information is the offense ID for the current offense on the Offense Summary page.

3. The **custom_html_generator** function formats the offense **JSON** that was retrieved into an HTML table.

---

**Custom column example**

You can add columns that contain custom content to tables in QRadar.

The following example describes how to add a column to the Offenses tab and to inject content into it. The content is in **JSON LD** format and is formatted by a custom JavaScript.

The following image shows an example of a custom column that is added to the Offenses tab.
The following manifest.json file example shows details of a custom_columns block.

**manifest.json**

Use the custom_columns block in the app's manifest.json file to define the injection point, column header, and rest endpoint that are used to retrieve content in JSON format for the new column.

In this example, the JSON content that is injected is formatted by using a custom JavaScript. The location of the script, and the tab and page on which it is run are defined in a page_scripts block.

```json
{
    "name": "Custom column offenses list table",
    "description": "Render custom column offense using custom javascript",
    "uuid": "7191b673-3225-4d5d-97ba-a41d72cc65f0",
    "version": "1.0"

    "custom_columns": [
        {
            "label": "Custom col custom javascript", 1
            "rest_endpoint": "custom_column_method", 2
            "page_id": "OffenseList" 3
        }
    ],

    "page_scripts": [
        {
            "app_name": "SEM",
            "page_id": "OffenseList",
            "scripts": [
                "static/qjslib/custom_offense.js"
            ]
        }
    ]
}
```

The following list describes the contents in the code snippet from the custom_columns block.

---

**Figure 6. Custom column on the Offenses tab.**
1. The label field contains the column header content.
2. The custom_column_method REST endpoint is defined in the app/views.py script. This custom_column_method REST endpoint is used to retrieve the information that is injected into the custom column.
3. The page_id field defines the page in which the new table column is added. In this case, it is the All Offenses page.

```
app/views.py
__author__ = 'IBM'

from flask import Response
from app import app
from qpylib.qpylib import log
from qpylib.offense_qpylib import get_offense_json_ld
import json

@app.route('/custom_column_method/<offense_id>', methods=['GET'])
def get_offense(offense_id):
    try:
        log("get offense")
        offense_json = get_offense_json_ld(offense_id)
        return Response(response=offense_json, status=200, mimetype='application/json')
    except Exception as e:
        log('Error ' + str(e))
        raise
```

The following list describes the contents in the code snippet from the views.py script.
1. The get_offense_json_ld function that is imported from the app/qpylib/offense_qpylib.py library retrieves the details for the offense ID in JSON LD format.
2. The get_offense function includes an @app.route annotation that defines the endpoint’s route. The @app.route includes the manifest’s rest_endpoint field value /custom_column_method, and the context information. In this case, the context information is the offense ID.

```
static/qjslib/custom_offense.js

The static/qjslib/custom_offense.js script renders the JSON content for the Offense ID and its source IP address.

function renderJsonContent(jsonTagId, targetDivTagId)
{
    var jsonTagContent = $("#" + jsonTagId).html();
    var json = JSON.parse(jsonTagContent);
    $("#" + targetDivTagId).html(renderOffense(json));
}

function renderOffense(json)
{
    return 'id is ' + json.data.id + ', ' +
    'Source IP is ' + json.data.offense_source;
14 Support functions

The IBM QRadar GUI Application Framework comes with several built-in routes, custom Jinja2 Flask functions, and other helper utilities that support app development.

Overview

All HTTP requests from client-side browsers that go to an app use the following format:

https://<console_ip>/console/plugins/{application_id}/app_proxy/{my_route}

The application_id is the integer value that is assigned during the process of using the installation RESTful endpoints for GUI app creation. The application_id value is recorded in the Application Creation Task state output that is returned when you run the qradar_app_creator deploy command.

In the following example, the application ID is 1023.

Application Creation Task state:

```json
{"status": 'COMPLETED', 'application_id': '1023', 'error_messages': '[]'}
```

Routes

You can create your own targeted web requests to the app for the routes in the following table:

<table>
<thead>
<tr>
<th>Route</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /debug</td>
<td>GET https://&lt;console_ip&gt;/console/plugins/{application_id}/app_proxy/debug</td>
<td>Download your /store/log/app.log file from inside the container for inspection.</td>
</tr>
<tr>
<td>POST /log_level</td>
<td>POST https://&lt;console_ip&gt;/console/plugins/{application_id}/app_proxy/log_level</td>
<td>Dynamically define the level of logging that you want your app to capture. Post a form, with an attribute level that is set to one of the log level values to this endpoint. QRadar dynamically reset the log collection levels in your /store/log/app.log file.</td>
</tr>
<tr>
<td>form body:</td>
<td>level = 'INFO' 'DEBUG' 'ERROR' 'WARNING' 'CRITICAL'</td>
<td></td>
</tr>
</tbody>
</table>

Accessing your Flask endpoints in views.py

You must use relative paths for your endpoints. Do not use the utility methods q_url_for and get_console_ip to create URLs to access your Flask endpoints. If a console uses a web URL instead of an IP address, all the Flask requests are denied because the request is cross-domain. In that case, your app might not work. If you want to access an endpoint from the main app template level, for an AJAX call, for example, to return some JSON data, use the following URL format:

```python
url_cpu_data = 'cpu_data'
```

This format routes the method in views.py:

```python
@app.route('/cpu_data', methods=['GET'])
```

If you are working from a folder at a deeper level, for example, use ../cpu_data to go back a level to reach this endpoint.

Do not use the following format to create a URL to access your endpoint:

```python
url_cpu_data = "{ { q_url_for('cpu_data') } }"
```

As before, if you go from a web URL to an IP address, this request might be denied because the request is cross-domain and so your app might not work.
If you begin the URL with a slash, it starts the URL from this root: https://consoleIPaddress/

Here's an example to open a page with the offense ID:

```
console/qradar/jsp/QRadar.jsp?appName=Sem&pageld=OffenseSummary&summaryId="
```

Here's an example to open a page by running the AQL search query:

```
/console/do/ariel/arielSearch?appName=EventViewer&pageld=EventList&dispatch=performSearch&value(searchMode)="AQL&value(timeRangeType)=aqlTime"+"value(aql)="+encodeURIComponent(aql_query)
```

**Custom Flask methods**

The following table describes the Flask custom methods that you can use:

<table>
<thead>
<tr>
<th>Method</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
</table>
| q_url_for()    | def q_url_for(endpoint, **values): | Use this Python method inside your routes in your app's views.py or Jinja2 templates. It hides and abstracts the QRadar proxy addresses that are needed to link together endpoints (routes) or resources (static files and images, for example). The method is essentially a wrapper around the Flask url_for(...) method. It applies a prefix that pertains to the correct application-specific proxy path URL portion to get to your app's endpoint. The following snippet shows how to use the method in a Jinja2 template to hide the QRadar proxy address of an image resource.
```
<img src="{{ q_url_for( static, filename='images/core_image.png' ) }}" width="256" height="256" alt="previous" title="Previous" border="0">
```
For more information about the Flask url_for(...) method, go to the Flask website.

| getAppBaseUrl() | def getAppBaseUrl(): | Function to get the full URL through QRadar, that will proxy any request to the appropriate application plug-in servlet. This routine returns a URL string that can be appended to create a URL to reference resources in the application. Typically, the q_url_for() function is used for this purpose but getAppBaseUrl() is also supplied for convenience. |

**QRadar CSRF token**

The QRadar CSRF (cross-site request forgery) token is generated by QRadar to prevent cross-site scripting attempts. For applications that are developed by using QRadar support libraries such as qpylib the use of the CSRF token is seamless.

If you create an app that does not use QRadar qpylib support, you must harvest the token from any application endpoints, or routes, and place it in your own HTTP headers before a REST or exported method call is made in QRadar.

**Related information:**

[http://flask.pocoo.org/docs/0.10/api/#flask.url_for](http://flask.pocoo.org/docs/0.10/api/#flask.url_for)
15 QRadar Python helper library functions

The QRadar Python helper library (qpylib) contains several useful functions that you can use to add logging, make REST API calls, and convert JSON objects to Python dictionaries.

All functions that you import into your app's views.py file can be called globally.

The following table describes functions that you can import into your app's views.py file.

**Table 32. Functions that you can import into your app**

<table>
<thead>
<tr>
<th>Function</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log()</td>
<td><code>def log(message, level='info'):</code></td>
<td>Import the qpylib helper library into your app's views.py to use the log() function. This function writes messages at your chosen log level to the /store/log/app.log file. By default, logging is turned on and set to INFO level. Lower level logging messages are ignored. Use the POST /log_level endpoint to change.</td>
</tr>
<tr>
<td>set_log_level</td>
<td><code>def set_log_level(log_level='info'):</code></td>
<td>Set the current log level. Used by the POST /log_level endpoint but can also be called programmatically.</td>
</tr>
<tr>
<td>REST()</td>
<td><code>def REST(RESTtype, requestURL, headers={}, data=None, params=None, json=None, version=None):</code></td>
<td>Import the qpylib library to use this function to make calls to the QRadar REST API endpoints. The endpoint takes care of authentication and authorization by reusing the security tokens that are passed on the request from QRadar.</td>
</tr>
<tr>
<td>to_json_dict</td>
<td><code>def to_json_dict(python_obj):</code></td>
<td>Converts a JSON object into a Python dictionary.</td>
</tr>
</tbody>
</table>
16 Jinja2 templates

Jinja2 is a Python library that you can use to construct templates for various output formats from a core template text file. It can be used to create HTML templates for IBM QRadar applications.

Jinja2 has a rich API, and large array of syntactic directives (statements, expressions, variables, tags) that allow the dynamic injection of content into the templated file.

Use the Flask render_template() method in the app's views.py file to inject data from your Python method, served by the route, into a Jinja2 templated HTML file. For example:

```python
__author__ = 'IBM'

from flask import render_template
from app import app

@app.route('/')
def hello_world():
    return render_template("hello.html", title = "QRadar")
```

The hello.html template must be stored in the /app/templates folder. The hello.html file is described in the following section:

```html
<!doctype html>
<title>Hello from Flask</title>
<h1>Hello {{ title }}!</h1>
```

The template produces the following output:

```html
<!doctype html>
<title>Hello from Flask</title>
<h1>Hello QRadar!</h1>
```

**Note:** Do not use the Flask-Jinja2-mandated url_for functionality within your app Jinja2 template. The QRadar GUI Application Framework uses relative addressing for request paths. If you use url_for, it creates an absolute request path from the container itself.

For more information about Jinja2 templates, see the [Jinja2 documentation](#).

**Edit Jinja2 templates in Eclipse**

You can use the Django template editor plug-in in Eclipse to develop Jinja2 templates.

PyDev Eclipse does not come with a Jinja2 template editor by default. The Django template editor plug-in offers useful features that you can employ to develop Jinja2 templates for your app.

Install the Django repository [http://pydev.org/updates](http://pydev.org/updates) by clicking Help > Install New Software on the main Eclipse Help panel.

This plug-in offers useful syntax-highlighting and auto-completion features for Jinja2 template development.

**Integrate JavaScript libraries into your template**

Add CSS and JavaScript libraries to your HTML templates to style and enhance your apps user interface.
You can use the Dojo and JQuery JavaScript libraries that are integrated into IBM QRadar to add CSS styling, widgets, and other UI features to your app.

**Dojo**

To integrate the Dojo JavaScript toolkit into your app's HTML template, add the following tags to your template's HEAD element:

```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My app!</title>
  <link type="text/css" rel="stylesheet" href="/console/idt/dojo/resources/dojo.css"></link>
  <link type="text/css" rel="stylesheet" href="/console/idt/dijit/themes/dijit.css"></link>
  <link type="text/css" rel="stylesheet" href="/console/idt/dijit/themes/claro/claro.css"></link>
  <link type="text/css" rel="stylesheet" href="/console/idt/idx/themes/oneui/oneui.css"></link>
  <script type="text/javascript" src="/console/idt/dojo/dojo.js"
    data-dojo-config="async:true, parseOnLoad: true"></script>
</head>

<body></body>
</html>
```

QRadar uses Dojo 1.9.3.

**JQuery**

To integrate the JQuery JavaScript library into your app's HTML template, add the following tag to your template's HEAD element:

```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My app!</title>
  <script type="text/javascript" src="/console/core/js/jquery/jquery.min.js"></script>
</head>

<body></body>
</html>
```
17 App Framework JavaScript library

The App Framework JavaScript library provides helper functions for common QRadar calls that you can integrate into your own scripts.

The helper functions that are included in the App Framework JavaScript library (app/static/qjslib/qappfw.js) can be used to do the following tasks:
- Get information on the current application, user, item, or selected table rows.
- Open the Details page for an asset.
- Search for an event or flow by using a specified AQL query.
- Open the Details page for an offense.
- Call a REST method by using an XMLHttpRequest.

You can view the HTML documentation for the JS library in the jsdoc directory after you unzip the SDK (.zip) file.

The helper functions in the App Framework JavaScript Library are described in the following table:

Table 33. App Framework JavaScript Library helper functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getApplicationBaseUrl(id)</td>
<td>Returns the base URL of an application.</td>
</tr>
<tr>
<td>getApplicationId()</td>
<td>Returns the ID of the current application.</td>
</tr>
<tr>
<td></td>
<td>Note: The functions getApplicationId() and getApplicationBaseUrl(id) depend on a variable called CURRENT_SCOPE, which is injected into the JavaScript context by the app framework when it imports a JavaScript file that is defined in the page_scripts section of an app manifest file. The functions can only be used if you are using the page_scripts mechanism to pull in your custom JavaScript. For example, if you use a script tag in a Jinja template the functions won’t work.</td>
</tr>
<tr>
<td>getCurrentUser()</td>
<td>Returns the current user.</td>
</tr>
<tr>
<td>getItemId()</td>
<td>Returns the ID of the item that is viewed (for example, an asset or offense).</td>
</tr>
<tr>
<td>getSelectedRows()</td>
<td>Returns the IDs of selected rows on a list page, such as the offense or asset list.</td>
</tr>
<tr>
<td>openAsset(assetId, openWindow)</td>
<td>Opens the Details page of an asset, either in a new window or in the Assets tab.</td>
</tr>
<tr>
<td>openAssetForIpAddress(ipAddress, openWindow)</td>
<td>Opens the Details page of an asset for an IP address, either in a new window or in the Assets tab.</td>
</tr>
<tr>
<td>openEventSearch(aql, openWindow)</td>
<td>Runs an event search with the specified AQL string, either in a new window or the Event Viewer tab.</td>
</tr>
<tr>
<td>openFlowSearch(aql, openWindow)</td>
<td>Runs a flow search with the specified AQL string, either in a new window or the Flow Viewer tab.</td>
</tr>
<tr>
<td>openOffense(offenseId, openWindow)</td>
<td>Opens the Details page of an offense, either in a new window or in the Offenses tab.</td>
</tr>
<tr>
<td>rest(args)</td>
<td>Calls a REST method by using an XMLHttpRequest.</td>
</tr>
</tbody>
</table>
18 Communicating with QRadar hosts from Python

You can communicate with IBM QRadar Hosts from Python by using the REST endpoints that QRadar exposes.

As part of any communication from QRadar to an app, QRadar provides the following headers:

**QRADAR-USER**
This header contains the QRadar user name.

**QRADAR-USER-ROLE**
This header contains the user role assigned to the user.

**QRADAR-SECURITY-PROFILE**
This header contains the security profile that defines which networks, log sources, and domains that the user can access.

**qpylib library**

The qpylib library provides functions that encapsulate much of the logic that is required to initiate this communication.

For REST API calls, use the qpylib.REST(RESTtype, request_url, headers, data, json, params, version) function. This function prepends the IP address of the host console to the request URL.

This REST function acts as a wrapper for the Python requests library. It returns a `requests.Response` object.

The following table describes the fields that you can access from this object:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>status_code</strong></td>
<td>Status code for the response. Useful for determining the success of a request. For example, if you are checking for a 200 response.</td>
</tr>
<tr>
<td><strong>url</strong></td>
<td>URL of the request.</td>
</tr>
<tr>
<td><strong>headers</strong></td>
<td>Dictionary object that contains the response headers.</td>
</tr>
<tr>
<td><strong>text</strong></td>
<td>Raw text output of the response. Useful for debugging purposes.</td>
</tr>
</tbody>
</table>

The function parameters are explained in the following table:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESTtype</strong></td>
<td>String REST request type. Accepts 'GET', 'PUT', 'POST' and 'DELETE'.</td>
</tr>
<tr>
<td><strong>request_url</strong></td>
<td>URL of the REST endpoint. The qpylib library prepends the appropriate console IP address to the URL so that only the URL from /api/ is needed. For example: /api/gui_app_framework/applications.</td>
</tr>
<tr>
<td><strong>headers</strong> (optional)</td>
<td>Optional headers to be added to the request. Headers must be contained within a Python dictionary object, for example, {'Accept': 'application/json'}.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>data (optional)</td>
<td>Optional data that can be contained within a request's body. Data must be in the format that is appropriate to the REST endpoint. For example, data must be converted to a JSON string by using the JSON Python library (json.dumps()) when a REST endpoint accepts application/json.</td>
</tr>
<tr>
<td>json (optional)</td>
<td>Optional parameter that accepts Python dictionary objects that are converted to a JSON String that is included in the request's body.</td>
</tr>
<tr>
<td>params (optional)</td>
<td>Optional parameter that accepts Python dictionary objects that are converted to URL query parameters.</td>
</tr>
<tr>
<td>version (optional)</td>
<td>Optional parameter that specifies which version of the QRadar RESTful API to use. The value must be a string that matches a supported version of the QRadar RESTful API (for example, 5.0). If no version is specified, no version header is sent and the most recent version is used by default.</td>
</tr>
</tbody>
</table>

The Response object also contains functions that simplify access to the data contained in the response body. You can use the json() function to retrieve a dictionary object that contains the response body, or a list of dictionary objects if the endpoint returns a collection.

**Example: Get QRadar Offenses**

```python
import qpylib
offenses_endpoint = '/api/siem/offenses'
headers = {'content-type': 'application/json'}
response = qpylib.REST('GET', offenses_endpoint, headers=headers)
offenses_json_list = response.json()
# List containing dictionary objects for each QRadar offense

# Iterate over each offense JSON in the list and print its id.
format_string = 'Found offense id [{0}].'
for offense_json in offenses_json_list:
    offense_id = str(offense_json['id'])  # Access fields
    print(format_string.format(offense_id))
```

**Example: Get QRadar Offenses With Queries**

```python
import qpylib
offenses_endpoint = '/api/siem/offenses'
headers = {'content-type': 'application/json'}
params = { 'filter': 'inactive=false' }
response = qpylib.REST('GET', offenses_endpoint, headers=headers, params=params)
offenses_json_list = response.json()
```

**Example: Post QRadar Offense Closing Reason**

```python
import qpylib
offense_closing_reasons_endpoint = '/api/siem/offense_closing_reasons'
headers = {'content-type': 'application/json'}
json_dict = { 'reason': 'Demonstrating posting data to QRadar' }
response = qpylib.REST('POST', offense_closing_reasons_endpoint, headers=headers, json=json_dict)
```
19 GUI Application Framework REST API endpoints

GUI Application Framework endpoints are available in the latest version of the IBM QRadar API that you can use to develop apps.

Apps are packaged as compressed archives (.zip), within the extension archive. Extensions can be installed or uninstalled by using RESTful endpoints. Apps within an extension (.zip) file use extra RESTful endpoints to control the lifecycle of an App with QRadar (install, delete, start, stop).

The following table lists the GUI Application Framework REST API endpoints:

Table 36. GUI Application Framework REST API endpoints

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /gui_app_framework/application_creation_task</td>
<td>Application ID</td>
<td>Retrieves a list of status details for all asynchronous requests to create apps.</td>
</tr>
<tr>
<td>GET /gui_app_framework/application_creation_task/{application_id}</td>
<td>Application ID</td>
<td>Retrieves a list of status details of an asynchronous request to create apps.</td>
</tr>
<tr>
<td>POST /gui_app_framework/application_creation_task</td>
<td>Application (.zip) bundle file</td>
<td>Creates an app within the application framework, and registers it with QRadar. The app is created asynchronously. A reference to the application_id is returned and must be used in subsequent API calls to determine the status of the app installation.</td>
</tr>
<tr>
<td>POST /gui_app_framework/application_creation_task/{application_id}</td>
<td>Application ID, cancel status</td>
<td>Updates a new app installation within the application framework. The application_id and a status parameters are required.</td>
</tr>
<tr>
<td>GET /gui_app_framework/applications</td>
<td></td>
<td>Retrieves a list of apps that are installed on the QRadar console, and their manifest JSON structures and status.</td>
</tr>
<tr>
<td>GET /gui_app_framework/applications/{application_id}</td>
<td>Application ID</td>
<td>Retrieves a specific app that is installed on the console and its manifest JSON structure and status.</td>
</tr>
<tr>
<td>POST /gui_app_framework/applications/{application_id}</td>
<td>Application ID, start/stop status</td>
<td>Updates an app. Starts or stops an app by setting status to RUNNING or STOPPED respectively.</td>
</tr>
<tr>
<td>PUT /gui_app_framework/applications/{application_id}</td>
<td>Application ID</td>
<td>Upgrade an application.</td>
</tr>
<tr>
<td>DELETE /gui_app_framework/applications/{application_id}</td>
<td>Application ID</td>
<td>Deletes an application.</td>
</tr>
</tbody>
</table>

For more information, see the API documentation page on your QRadar Console: https://<Console_IP>/api_doc. Alternatively, see the IBM QRadar API Guide.
20 App logs

App logs are stored in the /store/log directory of your application’s Docker container.

The /store/log directory contains 2 log files:
- startup.log is the initial start-up log for the application. This log is useful for checking the installation of dependencies added to your app’s app/src_deps/ folder.
- app.log is the log file that is created by the qpylib library. Logging calls to the qpylib.log() method are written in the app.log file.

Adding logging to your app

The IBM QRadar Python helper library (qpylib) contains two useful functions that you can use to add logging to your app.

The log() function

Import the qpylib helper library into your app’s views.py to use the log() function. This function writes messages at your chosen log level to the /store/log/app.log file.

By default, logging is turned on and set to INFO level. Lower level logging messages are ignored. Use the POST /log_level endpoint to change log level.

The log() function uses the following format:

```python
def log(message, level='info'):
```

For example:
```python
from qpylib import qpylib
...
# in precedence order from lowest level to highest
qpylib.log('debug message', 'debug')
qpylib.log('info message', 'info')
qpylib.log('warning message', 'warning')
qpylib.log('error message', 'error')
qpylib.log('critical message', 'critical')
```

Note:

All qpylib functions that you import into your app’s views.py file can be called globally. For that reason, it is a good idea to add a namespace to the qpylib functions you import to prevent clashes with other functions.

The set_log_level() function

You can use this function to set the current log level. This function is used by the POST /log_level endpoint but can also be called programmatically.
```python
def set_log_level(log_level='info'):
```
**Viewing your app logs**

Use built-in routes to create HTTP requests download, view, and set log collection levels.

You can create your own targeted web requests to the app for the following routes:

<table>
<thead>
<tr>
<th>Table 37. Request routes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Route</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>POST /log_level</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

The **application_id** is the integer value that is assigned when you use the installation RESTful endpoints for GUI app creation. The **application_id** value is recorded in the Application Creation Task state output that is returned when you run the `qradar_app_creator deploy` command.

**Viewing logs within the host directory**

Alternatively, you can use the `/opt/qradar/support/qapp_utils.py` utility to view your app logs by using its internal shell.

All logs are located in the `/store/log` directory of the container.

For more information about how to obtain the container ID, see 22, “Accessing your app’s command line,” on page 93.
21 Stopping, restarting, and uninstalling an app

You can stop, restart, or uninstall your app by using the IBM QRadar GUI Application Framework REST API endpoints.

**Stopping your app**

You can disable your application that is running by sending the following empty POST request:

POST /api/gui_app_framework/applications/{app_id}?status="STOPPED"

**Restarting your app**

You can restart your app by sending the following POST request:

POST /api/gui_app_framework/applications/{app_id}?status="RUNNING"

**Uninstalling your app**

Use the following DELETE request to uninstall and remove an app:

DELETE /api/gui_app_framework/applications/{app_id}
22 Accessing your app's command line

Access the command line of your installed apps by using the app's container ID.

The command `/opt/qradar/support/qapp_utils.py ls` displays a list of running app containers. The following output sample shows an app that is installed and is running.

```
[root@hostname ~]# /opt/qradar/support/qapp_utils.py ls
ID NAME PORT CONTAINER IMAGE STATUS
1065 QHelloWorld_1 49182 c2671ae36247 7f614d3fa6de4a66b5ad7b5ed:latest RUNNING
```

You can use the **container ID** to execute the command `/opt/qradar/support/qapp_utils.py connect <id>` to access the bash terminal within the app.

```
/opt/qradar/support/qapp_utils.py connect 1065
bash-4.1#
```

The app terminal has access to many base Linux programs such as `vi` and `tar`. You can use the app terminal to look at app logs in the `/store/log` file. You can also use the app terminal to make dynamic changes to code located in the `/app` directory. Use the `exit` command to close the app container terminal and to return to the host.
23 App upgrades

Upgrade your app by using the IBM QRadar Extensions Management.

Note: As a best practice, store your app configuration and data in /store because data in this directory is protected during app upgrades.

Important: If you want to upgrade an installed app that uses a lot of memory, you might need to stop the app before you do the upgrade to avoid memory resource issues, and then restart the app after you complete the upgrade.
24 Available user role capabilities

Capabilities are sets of permissions that are tied to user roles that are defined in the IBM QRadar Admin tab.

The following table lists the capabilities that are supported by the QRadar GUI Application Framework. Use the values in the **Capability** column to define the user privileges for your app in the `required_capabilities` field of the object type block in your application’s `manifest.json` file.

The following table describes supported user role capabilities.

<table>
<thead>
<tr>
<th>Capability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADMIN</td>
<td>System administrator. Grants permission to access all areas of the user interface. Users who have this access cannot edit other administrator accounts.</td>
</tr>
<tr>
<td>VIEWADMIN</td>
<td>Remote networks and services configuration. Grants permission to configure remote networks and services on the Admin tab.</td>
</tr>
<tr>
<td>SEM</td>
<td>Offense management</td>
</tr>
<tr>
<td>SEM.VIEWRULES</td>
<td>View custom rules</td>
</tr>
<tr>
<td>SEM.RULECREATION</td>
<td>Maintain custom rules</td>
</tr>
<tr>
<td>SEM.ASSIGNOFFENSE</td>
<td>Assign offenses to users</td>
</tr>
<tr>
<td>SEM.MANAGECLOSINGREASONS</td>
<td>Manage offense closing reasons</td>
</tr>
<tr>
<td>EventViewer</td>
<td>Event viewer</td>
</tr>
<tr>
<td>EventViewer.VIEWRULES</td>
<td>View custom rules</td>
</tr>
<tr>
<td>EventViewer.RULECREATION</td>
<td>Maintain custom rules</td>
</tr>
<tr>
<td>EventViewer.CUSTOMARIELPROPERTY</td>
<td>User-defined event properties</td>
</tr>
<tr>
<td>EventViewer.MANAGE TimESERIES</td>
<td>Manage time series</td>
</tr>
<tr>
<td>ASSETS</td>
<td>Asset management</td>
</tr>
<tr>
<td>ASSETS.VA DATA</td>
<td>View VA data</td>
</tr>
<tr>
<td>ASSETS.VASCAN</td>
<td>Perform VA scans</td>
</tr>
<tr>
<td>ASSETS.SERVERDISCOVERY</td>
<td>Server discovery</td>
</tr>
<tr>
<td>ASSETS.REMOVEVULNS</td>
<td>Remove vulnerabilities</td>
</tr>
<tr>
<td>SURVEILLANCE</td>
<td>Network Surveillance</td>
</tr>
<tr>
<td>SURVEILLANCE.VIEWRULES</td>
<td>View custom rules</td>
</tr>
<tr>
<td>SURVEILLANCE.DATAMINECONTENT</td>
<td>View flow content</td>
</tr>
<tr>
<td>SURVEILLANCE.CUSTOMFLOWPROPERTY</td>
<td>User-defined flow properties</td>
</tr>
<tr>
<td>SURVEILLANCE.MANAGE TimESERIES</td>
<td>Manage time series</td>
</tr>
<tr>
<td>SURVEILLANCE.RULECREATION</td>
<td>Maintain custom rules</td>
</tr>
<tr>
<td>REPORTING</td>
<td>Reporting</td>
</tr>
<tr>
<td>REPORTING.MAINTAINTEMPLATES</td>
<td>Maintain templates</td>
</tr>
<tr>
<td>REPORTING.DISTRIBUTE</td>
<td>Distribute reports via email</td>
</tr>
<tr>
<td>FORENSICS</td>
<td>Incident Forensics</td>
</tr>
</tbody>
</table>
Table 38. Supported user role capabilities (continued)

<table>
<thead>
<tr>
<th>Capability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORENSICS.CASECREATION</td>
<td>Create cases in Incident Forensics</td>
</tr>
<tr>
<td>QRM</td>
<td>QRM permission</td>
</tr>
<tr>
<td>QVM</td>
<td>QVM permissions</td>
</tr>
<tr>
<td>QVM.ASSIGNASSETOWNER</td>
<td>Assign asset owner</td>
</tr>
<tr>
<td>QVM.VULNERABILITY</td>
<td>Assign vulnerability permissions</td>
</tr>
<tr>
<td>QVM.EXCEPTION</td>
<td>Exception vulnerability permissions</td>
</tr>
<tr>
<td>QVM.SCANPOLICY</td>
<td>Scan policy permissions</td>
</tr>
<tr>
<td>QVM.SCANPROFILE</td>
<td>Scan profile permissions</td>
</tr>
</tbody>
</table>

For more information about capabilities and user roles in QRadar, see the *IBM QRadar Administration Guide*. 
25 App names, GUI action groups, and page IDs

App names, GUI action groups, and page IDs are identifiers that IBM QRadar uses for QRadar products, GUI actions, and UI pages.

The following manifest blocks use these identifiers.

The groups field of the gui_actions block uses GUI action group identifiers to specify the toolbar or right-click menu where the GUI action is added:

```
"gui_actions": [  
    {  
      "id":"sampleToolbarButton",  
      "text":"Sample Toolbar Button",  
      "description":"Sample toolbar button that calls a REST method, passing an offense ID along",  
      "icon":null,  
      "rest_method":"sampleToolbarMethod",  
      "javascript":"alert(result)"  
    },  
    {  
      "id":"sampleRightClickMenuItem",  
      "text":"Sample Right Click Menu Item",  
      "description":"Sample right-click menu item that calls a REST method, passing an offense ID along",  
      "icon":null,  
      "rest_method":"sampleRightClickMethod",  
      "javascript":"alert(result)"  
    }  
},
```

The app_name and page_id fields of the page_scripts block use the app name and page ID identifiers to specify the QRadar application tab and sub page into which the page script is added:

```
"page_scripts": [  
    {  
      "app_name":"SEM",  
      "page_id":"OffenseList",  
      "scripts": ["/static/sampleScriptInclude.js"]  
    }  
  ],
```

The app_name and page_id fields of the page_scripts block use the app name and page ID identifiers to specify the QRadar application tab and sub page into which the page script is added.

```
"page_scripts": [  
    {  
      "app_name":"SEM",  
      "page_id":"OffenseList",  
      "scripts": ["/static/sampleScriptInclude.js"]  
    }  
  ],
```

**Supported app names**

The following table shows a list of supported app names with descriptions.

<table>
<thead>
<tr>
<th>App Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assetprofile</td>
<td>Asset Profile (for example, Vulnerabilities: Manage Vulnerabilities, search)</td>
</tr>
<tr>
<td>Assets</td>
<td>Assets Manager</td>
</tr>
<tr>
<td>EventViewer</td>
<td>Event Viewer (for example, syslogdestination)</td>
</tr>
<tr>
<td>Forensics</td>
<td>Incident Forensics</td>
</tr>
<tr>
<td>QRadar</td>
<td>QRadar (for example, Reference Data)</td>
</tr>
<tr>
<td>QVM</td>
<td>QRadar Vulnerability Manager</td>
</tr>
<tr>
<td>Reports</td>
<td>Reports</td>
</tr>
</tbody>
</table>
Table 39. Supported app names (continued)

<table>
<thead>
<tr>
<th>App Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sem</td>
<td>Offense Management</td>
</tr>
<tr>
<td>SRM</td>
<td>QRadar Risk Manager</td>
</tr>
<tr>
<td>Surveillance</td>
<td>Network Surveillance (Flows)</td>
</tr>
</tbody>
</table>

Supported GUI Actions and corresponding page IDs

The following table describes the supported GUI Actions and corresponding page IDs.

Table 40. Supported GUI Actions and Page IDs

<table>
<thead>
<tr>
<th>GUI Action group</th>
<th>App name</th>
<th>Page ID</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssetDetailsToolbar</td>
<td>Assets</td>
<td>AssetDetails/Valuelist</td>
<td>Assets - click on IP Address / Vulnerabilities - By Asset - click on IP Address</td>
</tr>
<tr>
<td>AssetListToolbar</td>
<td>Assets</td>
<td>AssetList</td>
<td>Asset tab list</td>
</tr>
<tr>
<td>AssetOwnerToolbar</td>
<td>assetprofile</td>
<td>AssetOwner</td>
<td>Vulnerabilities tab - left panel - Vulnerability Assignment</td>
</tr>
<tr>
<td>AttackerList</td>
<td>SEM</td>
<td>OffenseSummary</td>
<td>Offense Tab - All Offenses - double-click offense row - right-click row in Top 5 Source IPs section</td>
</tr>
<tr>
<td>ByAssetListFormToolbar</td>
<td>assetprofile</td>
<td>ByAssetListForm</td>
<td>Vulnerabilities tab - Manage vulnerabilities - By Asset</td>
</tr>
<tr>
<td>ByNetworkListToolbar</td>
<td>assetprofile</td>
<td>ByNetworkList</td>
<td>Vulnerabilities tab - Manage vulnerabilities - By Network</td>
</tr>
<tr>
<td>ByOpenServiceListToolbar</td>
<td>assetprofile</td>
<td>ByOpenServiceList</td>
<td>Vulnerabilities tab - Manage vulnerabilities - By Open Service</td>
</tr>
<tr>
<td>ByVulnerabilityInstanceListToolbar</td>
<td>assetprofile</td>
<td>ByVulnerabilityInstanceList</td>
<td>Vulnerabilities tab - Manage vulnerabilities - By vulnerability instance - main screen</td>
</tr>
<tr>
<td>ByVulnerabilityListToolbar</td>
<td>assetprofile</td>
<td>ByVulnerabilityList</td>
<td>Vulnerabilities tab - Manage vulnerabilities - By vulnerability</td>
</tr>
<tr>
<td>CategoryList</td>
<td>SEM</td>
<td>OffenseSummary</td>
<td>Offenses tab - All Offenses - double-click row - Top 5 Categories table - right-click a row</td>
</tr>
<tr>
<td>DomainListToolbar</td>
<td>QRadar</td>
<td>DomainList</td>
<td>Domain - Domain Management</td>
</tr>
<tr>
<td>EventDetailsToolbar</td>
<td>Event Viewer</td>
<td>EventDetails</td>
<td>Log Activity tab - pause - click row - Events detail toolbar</td>
</tr>
<tr>
<td>ExceptionRulesListToolbar</td>
<td>assetprofile</td>
<td>ExceptionRulesList</td>
<td>Vulnerabilities tab - Vulnerability exception</td>
</tr>
<tr>
<td>FlowDetailsToolbar</td>
<td>Surveillance</td>
<td>FlowDetails</td>
<td>Network Activity - double-click on a flow</td>
</tr>
<tr>
<td>FlowSourceListToolbar</td>
<td>Surveillance</td>
<td>FlowSourceList</td>
<td>Admin tab - Flow Sources</td>
</tr>
<tr>
<td>MyAssignedVulnerabilitiesListToolbar</td>
<td>assetprofile</td>
<td>MyAssignedVulnerabilitiesList</td>
<td>Vulnerability tab - My assigned vulnerabilities</td>
</tr>
<tr>
<td>NetworkHierarchyListToolbar</td>
<td>QRadar</td>
<td>NetworkHierarchyList</td>
<td>Admin tab - Network Hierarchy</td>
</tr>
<tr>
<td>NetworkListToolbar</td>
<td>SEM</td>
<td>NetworkList</td>
<td>Offenses - By Network</td>
</tr>
<tr>
<td>NetworkSummaryListToolbar</td>
<td>SEM</td>
<td>NetworkSummaryList</td>
<td>Offenses Tab - By Network - double-click an offense.</td>
</tr>
<tr>
<td>OffenseListToolbar</td>
<td>SEM</td>
<td>OffenseList</td>
<td>Offenses tab main page</td>
</tr>
<tr>
<td>OffenseSummaryToolbar</td>
<td>SEM</td>
<td>OffenseSummary</td>
<td>Offenses tab, double-click Offense - toolbar</td>
</tr>
<tr>
<td>ReferenceSetElementsContextMenu</td>
<td>QRadar</td>
<td>ReferenceSetElements</td>
<td>Admin tab - Reference Set Management - View a reference set - Content tab - right-click row</td>
</tr>
<tr>
<td>ReferenceSetElementsToolbar</td>
<td>QRadar</td>
<td>ReferenceSetElements</td>
<td>Admin tab - Reference Set Management - View a reference set - Content tab</td>
</tr>
<tr>
<td>ReferenceSetRulesToolbar</td>
<td>QRadar</td>
<td>ReferenceSetRules</td>
<td>Admin tab - Reference Set Management - View a reference set - References tab, click toolbar button</td>
</tr>
<tr>
<td>ReferenceSetsToolbar</td>
<td>QRadar</td>
<td>ReferenceSets</td>
<td>Admin tab - Reference Set Management</td>
</tr>
<tr>
<td>GUI Action group</td>
<td>App name</td>
<td>Page ID</td>
<td>Location</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>ReportTemplateListToolbar</td>
<td>Reports</td>
<td>ReportTemplateListAll</td>
<td>Reports tab toolbar</td>
</tr>
<tr>
<td>ScanPolicyVulnerabilityListToolbar</td>
<td>assetProfile</td>
<td>ScanPolicyVulnerabilityList</td>
<td>Vulnerabilities tab - Left panel - Administrative - Scan Policies - Add - Scan Type Patch - Vulnerabilities Tab - Add. GUI action appears in toolbar</td>
</tr>
<tr>
<td>SensorDeviceListToolbar</td>
<td>EventViewer</td>
<td>SensorDeviceList</td>
<td>Admin tab - Log Sources</td>
</tr>
<tr>
<td>TargetList</td>
<td>SEM</td>
<td>OffenseSummary</td>
<td>Offenses tab - double-click offense, Top 5 Destination IPs table, right-click row.</td>
</tr>
<tr>
<td>TargetListToolbar</td>
<td>SEM</td>
<td>TargetList</td>
<td>Offenses tab - By Destination IP.</td>
</tr>
<tr>
<td>TenantListToolbar</td>
<td>QRadar</td>
<td>TenantList</td>
<td>Admin tab - Tenant Management</td>
</tr>
<tr>
<td>VaScannerSchedulesListToolbar</td>
<td>Assets</td>
<td>VaScannerSchedulesList</td>
<td>Admin tab - Schedule VA Scanners</td>
</tr>
<tr>
<td>VaScannersListToolbar</td>
<td>Assets</td>
<td>VaScannersList</td>
<td>Admin tab - VA Scanners</td>
</tr>
<tr>
<td>VulnerabilityManagementListPopup</td>
<td>assetProfile</td>
<td>ByAssetListForm, ByNetworkList, ByOpenServiceList, ByVulnerabilityList, MyAssignedVulnerabilitiesList, ByVulnerabilityInstanceList</td>
<td>Vulnerabilities tab - Manage Vulnerabilities - By Network - By Asset - By Vulnerability - By Open Service - My Assigned Vulnerabilities</td>
</tr>
<tr>
<td>ArielListToolbar</td>
<td>Surveillance</td>
<td>FlowList</td>
<td>Network Activity tab</td>
</tr>
<tr>
<td>customEventListToolbar</td>
<td>EventViewer</td>
<td>EventList</td>
<td>Log Activity tab</td>
</tr>
<tr>
<td>ipPopup</td>
<td>Assets</td>
<td>AssetList</td>
<td>Right-click IP address on Assets tab, Vulnerability tab - Manage Vulnerabilities - Vulnerability Exception</td>
</tr>
<tr>
<td></td>
<td>assetProfile</td>
<td>MyAssignedVulnerabilitiesList</td>
<td></td>
</tr>
<tr>
<td></td>
<td>assetProfile</td>
<td>ByVulnerabilityInstanceList</td>
<td></td>
</tr>
<tr>
<td></td>
<td>assetProfile</td>
<td>ByAssetListForm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>assetProfile</td>
<td>ExceptionRulesList</td>
<td></td>
</tr>
<tr>
<td>userNamePopup</td>
<td>Assets</td>
<td>AssetDetailsVulnList</td>
<td>For example, Assets Tab, click IP Address to View Asset Details, select a row in Vulnerabilities table, right-click Technical User, see right-click View Assets /View Events - Also used in Offenses, Log Activity.</td>
</tr>
</tbody>
</table>

**Deprecated GUI actions**
26 Application globalization

Globalization refers to implementing multi-language support for your app that is built into the application without the need for further engineering efforts.

Consider two aspects when you globalize your app:

- Globalization of IBM QRadar specific elements such as tab labels, toolbar button text, and tooltip content.
- Globalization of application-specific content

Globalization of QRadar specific elements

QRadar elements are object types that can be defined in the application manifest.json file.

Globalization of QRadar elements includes the use of translated keys that are injected into your app's manifest.json, and ingested into the QRadar globalization store.

Globalization of application-specific content

The GUI Application Framework does not support any single pre-defined globalization approach nor does it provide an implementation stack for app developers to use for globalization. You can use common Flask and python packages and approaches to globalize your application-specific content.

Globalization of QRadar elements

Add language-specific text strings to your app to globalize IBM QRadar elements such as tab labels, toolbar button text, and tooltip content.

To globalize you app, you must create locale-specific properties files for the locales you want to use.

In the manifest.json file, you must configure the resource_bundles block to define the locales and the location of properties files that contains locale-specific text strings your app uses.

Globalization resource bundle properties files for the specified language code are stored in the app's app/static/resources folder. The string elements are displayed in the current IBM QRadar user’s preferred locale, as defined within the QRadar GUI itself.

In the following example, the "Hello World" app includes locale-specific text.

manifest.json

```json
{
   "name":"com.ibm.helloGLn.name",
   "description":"Application to display QHelloWorld",
   "version":"1.0.2",

   "areas": [
   {
      "id":"com.ibm.helloGLn.name",
      "text":"com.ibm.helloGLn.name",
      "description":"com.ibm.helloGLn.desc",
      "url":"index",
      "required_capabilities": ["ADMIN"]
   }
   ],
}
```
Three globalization keys are referenced by the app manifest metadata definition file:
- com.ibm.hellog11n.id
- com.ibm.hellog11n.name
- com.ibm.hellog11n.desc

The resource_bundles block defines locales for standard English, American English, French, and Spanish. The bundle field points to the properties file for each language locale.

**Globalization key naming conventions**

Use a consistent naming format for any globalization keys that are made available by resources files within your app code. One useful approach is to employ a fully qualified company-app prefix to all your keys. This strategy prevents the replication of existing QRadar globalization store keys or keys made available by other apps.

Here's an example:
- com.ibm.myapp.key1=value1
- com.ibm.myapp.key2=value2

**Properties files**

The resource_bundles block defines locales for standard English, American English, French, Spanish and Japanese. The bundle field points to the properties file for each language locale.

The properties files are stored in the app/static/resources folder.

The id, text, and description fields in the areas block use the key that is defined in the properties files. The key contains the value that is used for the app’s user interface tab name in each language.

Text strings for globalization are stored as key/value pairs in Java format properties files.
Language locale properties file must use the following naming convention:
application_<LANG>.properties. For the purposes of this example, four properties files were created.

These properties files are consumed by the main QRadar SIEM codebase, within a Java virtual machine (JVM). It must adhere to Java processing support for properties files. Currently, Latin-1 font characters, and Unicode characters are supported. Language sets that contain non-Latin-1 font characters must be represented by their Unicode equivalents. Install the Java 7 Java Development Kit (JDK) and use the Java Native-To-ASCII converter (native2ascii) to convert your content.

static/resources/hello_en.properties:
com.ibm.hello11n.name=Hello World
com.ibm.hello11n.id=Hello_World_G11n
com.ibm.hello11n.desc=A fully globalized Hello World App

static/resources/hello_en_US.properties:
com.ibm.hello11n.name=Hello World
com.ibm.hello11n.id=Hello_World_G11n
com.ibm.hello11n.desc=A fully globalized Hello World App

static/resources/hello_es.properties:
com.ibm.hello11n.name=Hola Mundo
com.ibm.hello11n.id=Hello_World_G11n
com.ibm.hello11n.desc=Un totalmente globalizado Hello World App

static/resources/hello_fr.properties:
com.ibm.hello11n.name=Bonjour Le Monde
com.ibm.hello11n.id=Hello_World_G11n
com.ibm.hello11n.desc=Un App World Bonjour totalement globalisé

static/resources/hello_ja.properties:
com.ibm.hello11n.name=\u3053\u3093\u306b\u3061\u4e16\u754c
com.ibm.hello11n.id=Hello_World_G11n
com.ibm.hello11n.desc=\u5b8c\u5168\u306b\u30b0\u30ed\u30fc\u30d0\u30eb\u306aHello World\u30a2\u30d7\u30ea\u30b1\u30fc\u30b7\u30e7\u30f3

Your Flask endpoint/services code: views.py

The views.py file defines a flask route, or service endpoint, that uses a Jinja2 template HTML page to build an HTML string that is to be returned from the service endpoint.

```python
__author__ = 'IBM'

from flask import render_template, send_from_directory
from app import app
from qpylib import qpylib

@app.route('/')
@app.route('/index')
def index():
    return render_template("index.html", title = "QApp1 : Hello World!")
```

The rendered views: app/templates/index.html

The following Jinja2 template is a simple web page, a static HTML string that is returned by the endpoint. No globalization occurs within your view with the following example.

```html
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>{{title}} - Main</title>
</head>
```

26 Application globalization 105
QRadar user locale preferences

To view the locale-specific content in QRadar, you must set your locale preferences.

Click Admin > User Preferences on the upper left of the QRadar user interface, and then select your locale from the Locale menu.

The following image shows the user locale preferences dialog.

For the current example, if you choose any of the English language locales, you see:

For the current example, if you choose any of the Spanish language locales, you see:

You see the following image, if you choose any of the English language locales.
You can determine the following information from these globalized examples:

- The new tab uses the value in the `manifest.json` `name` field as the text value for the tab.
- The tab tooltip the value in the `manifest.json` `description` field as the text value for its content.
- Globalization resource files that are bundled with the app are ingested by the QRadar globalization store.
- QRadar user session preferences for each locale use key look-ups to reflect the language-specific text you want to display.

**Globalization of application-specific content**

Globalize application-specific content by using Python Babel, Flask Babel, and Jinja2 templates to.

Use the links to the following technologies to help you globalize your application-specific content:

- Babel (http://babel.pocoo.org/) is the standard globalization package for Python.
- Flask-Babel (https://pythonhosted.org/Flask-Babel/) is an extension to Flask.
- Jinja2/Babel integration (http://jinja.pocoo.org/docs/dev/integration/) provides instructions on how to use Babel with Jinja2 templates.
Key concepts

For this example, you install the Flask-Babel pip package and its dependents into your app's src_deps/pip/ directory. The Flask-Babel package, provides the pybabel tool, which you can use to create translation files for your Flask-based app.

QRadar passes the user's preferred locale in the Accept-Languages header attribute through in the request header to your app.

To use the Flask-Babel package to create globalization text values that your app employs, use the following workflow:

1. Use the pybabel tool to extract out locale keys, typically to a .pot file.
2. Use the pybabel tool to build templated .po files for each language set you want to support.
3. Edit and complete the .po file. In other words, translate all the keys to the language-specific variant of the text value.
4. Use the pybabel tool to compile the completed .po files into a binary set of .mo files that can be employed by your Flask python code, or your Jinja2 templates.

Pre-requisites

To work through this example, you must install the following Python packages into your app's src_deps/pip/ directory.

- pytz-2015.6-py2.py3-none-any.whl
- Babel-2.1.1-py2.py3-none-any.whl
- speaklater-1.3.tar.gz
- Flask-Babel-0.9.tar.gz

You must also create an ordering.txt file in the src_deps/pip/ directory that contains the following content:

pytz-2015.6-py2.py3-none-any.whl
Babel-2.1.1-py2.py3-none-any.whl
speaklater-1.3.tar.gz
Flask-Babel-0.9.tar.gz

Note: Use the pip install -d src_deps/pip flask-babel command from within the virtual environment that the SDK provides to download the dependencies for Flask-Babel.

Bundle python wheel (whl) files instead of .tar file (tar.gz) source files wherever possible. Some raw python source package .tar files need to use the gcc compiler or other tools that the base docker container that hosts your app code might not have.

Build python wheel files from package source tarballs on your local system. Here's an example:

tar -xvzf some_package.tar.gz
python setup.py sdist bdist_wheel

Jinja2 template: app/templates/index.html

This example builds on the HelloWorld sample app. The original HTML template was built with hardcoded English language-specific text values. The following example wraps the English locale strings values with Jinja2 directives that use the gettext functions from Flask-Babel. Here's an example:

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>{{title}} - Main</title>
</head>
<body>
  <!-- Your content here -->
</body>
</html>
This example uses the shorthand alias for gettext function. You can also use the full form. For example:

```html
{{ gettext( '...' ) }}
```

This method provides a useful mechanism to quickly build and prototype your app. You can return later to make it locale aware. By using the actual initial text as keys, you keep the template readable.

**Note:** Eliminate white space around the directive. For example, consider the use of an HTML `<span>`, `<div>` or other element. The `pybabel` tool has some difficulties to extract all key values.

**Configure pybabel**

You must configure the `pybabel` tool so that it is aware of what source files to examine.

```bash
[python: **.py]
[jinja2: **/templates/**.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_
```

In this example, `pybabel` is configured to examine all Python source files in the `app/` folder, and all HTML files in any sub directory of the `app/templates/` folder.

The `pybabel` tool uses the `babe1.cfg` file to know which directories or files to examine within your app for potential translatable entries. It looks for `gettext(..)` and `_()` entries in your `*.py` files and your Jinja2 templates to build into a local `.pot` file.

**Create the .pot file**

To create a `.pot` file, open a command line and type the following command from within the `app/` folder:

```bash
pybabel extract -F babel.cfg -o messages.pot
```

A `messages.pot` file is created in the `app/` folder. Its content is similar to the following file:

```
# Translations template for PROJECT.
# Copyright (C) 2015 ORGANIZATION
# This file is distributed under the same license as the PROJECT project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2015.
#
msgid ""
msgstr ""
"Project-Id-Version: PROJECT VERSION\n"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS\n"
"POT-Creation-Date: 2015-10-07 21:27+0100\n"
```
You can use the `msgid` entries as your keys.

**Create the .po files**

You create individual language-specific .po files for Spanish, French, and English. From a command line, type the following commands from within the app/ folder:

```bash
pybabel init -i messages.pot -d translations -l es
pybabel init -i messages.pot -d translations -l fr
pybabel init -i messages.pot -d translations -l en
pybabel init -i messages.pot -d translations -l ja
```

These commands are used to create the translation files in the following locations:

- app/translations/es/LC_Messages/messages.po
- app/translations/fr/LC_Messages/messages.po
- app/translations/en/LC_Messages/messages.po
- app/translations/ja/LC_Messages/messages.po

The following example is the app/translations/es/LC_Messages/messages.po file that is generated:

```plaintext
msgid "" msgstr ""

"Project-Id-Version: PROJECT VERSION"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS"
"POT-Creation-Date: 2015-10-07 16:02+0100"
"PO-Revision-Date: 2015-10-07 16:04+0100"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>"
"Language-Team: es <LL@li.org>"
"Plural-Forms: nplurals=2; plural=(n != 1)"
"MIME-Version: 1.0"
"Content-Type: text/plain; charset=utf-8"
"Content-Transfer-Encoding: 8bit"
"Generated-By: Babel 1.3"

#: templates/index.html:11
msgid "IBM QRadar Application: Hello World"
msgstr ""
```

You can use the `msgid` entries as your keys.

**Create the .po files**

You create individual language-specific .po files for Spanish, French, and English. From a command line, type the following commands from within the app/ folder:

```bash
pybabel init -i messages.pot -d translations -l es
pybabel init -i messages.pot -d translations -l fr
pybabel init -i messages.pot -d translations -l en
pybabel init -i messages.pot -d translations -l ja
```

These commands are used to create the translation files in the following locations:

- app/translations/es/LC_Messages/messages.po
- app/translations/fr/LC_Messages/messages.po
- app/translations/en/LC_Messages/messages.po
- app/translations/ja/LC_Messages/messages.po

The following example is the app/translations/es/LC_Messages/messages.po file that is generated:

```plaintext
msgid "" msgstr ""

"Project-Id-Version: PROJECT VERSION"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS"
"POT-Creation-Date: 2015-10-07 16:02+0100"
"PO-Revision-Date: 2015-10-07 16:04+0100"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>"
"Language-Team: es <LL@li.org>"
"Plural-Forms: nplurals=2; plural=(n != 1)"
"MIME-Version: 1.0"
"Content-Type: text/plain; charset=utf-8"
"Content-Transfer-Encoding: 8bit"
"Generated-By: Babel 1.3"

#: templates/index.html:11
msgid "IBM QRadar Application: Hello World"
msgstr ""
```

You can use the `msgid` entries as your keys.

**Create the .po files**

You create individual language-specific .po files for Spanish, French, and English. From a command line, type the following commands from within the app/ folder:

```bash
pybabel init -i messages.pot -d translations -l es
pybabel init -i messages.pot -d translations -l fr
pybabel init -i messages.pot -d translations -l en
pybabel init -i messages.pot -d translations -l ja
```

These commands are used to create the translation files in the following locations:

- app/translations/es/LC_Messages/messages.po
- app/translations/fr/LC_Messages/messages.po
- app/translations/en/LC_Messages/messages.po
- app/translations/ja/LC_Messages/messages.po

The following example is the app/translations/es/LC_Messages/messages.po file that is generated:

```plaintext
msgid "" msgstr ""

"Project-Id-Version: PROJECT VERSION"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS"
"POT-Creation-Date: 2015-10-07 16:02+0100"
"PO-Revision-Date: 2015-10-07 16:04+0100"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>"
"Language-Team: es <LL@li.org>"
"Plural-Forms: nplurals=2; plural=(n != 1)"
"MIME-Version: 1.0"
"Content-Type: text/plain; charset=utf-8"
"Content-Transfer-Encoding: 8bit"
"Generated-By: Babel 1.3"

#: templates/index.html:11
msgid "IBM QRadar Application: Hello World"
msgstr ""
```

You can use the `msgid` entries as your keys.

**Create the .po files**

You create individual language-specific .po files for Spanish, French, and English. From a command line, type the following commands from within the app/ folder:

```bash
pybabel init -i messages.pot -d translations -l es
pybabel init -i messages.pot -d translations -l fr
pybabel init -i messages.pot -d translations -l en
pybabel init -i messages.pot -d translations -l ja
```

These commands are used to create the translation files in the following locations:

- app/translations/es/LC_Messages/messages.po
- app/translations/fr/LC_Messages/messages.po
- app/translations/en/LC_Messages/messages.po
- app/translations/ja/LC_Messages/messages.po

The following example is the app/translations/es/LC_Messages/messages.po file that is generated:

```plaintext
msgid "" msgstr ""

"Project-Id-Version: PROJECT VERSION"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS"
"POT-Creation-Date: 2015-10-07 16:02+0100"
"PO-Revision-Date: 2015-10-07 16:04+0100"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>"
"Language-Team: es <LL@li.org>"
"Plural-Forms: nplurals=2; plural=(n != 1)"
"MIME-Version: 1.0"
"Content-Type: text/plain; charset=utf-8"
"Content-Transfer-Encoding: 8bit"
"Generated-By: Babel 1.3"

#: templates/index.html:11
msgid "IBM QRadar Application: Hello World"
msgstr ""
```

You can use the `msgid` entries as your keys.

**Create the .po files**

You create individual language-specific .po files for Spanish, French, and English. From a command line, type the following commands from within the app/ folder:

```bash
pybabel init -i messages.pot -d translations -l es
pybabel init -i messages.pot -d translations -l fr
pybabel init -i messages.pot -d translations -l en
pybabel init -i messages.pot -d translations -l ja
```

These commands are used to create the translation files in the following locations:

- app/translations/es/LC_Messages/messages.po
- app/translations/fr/LC_Messages/messages.po
- app/translations/en/LC_Messages/messages.po
- app/translations/ja/LC_Messages/messages.po

The following example is the app/translations/es/LC_Messages/messages.po file that is generated:

```plaintext
msgid "" msgstr ""

"Project-Id-Version: PROJECT VERSION"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS"
"POT-Creation-Date: 2015-10-07 16:02+0100"
"PO-Revision-Date: 2015-10-07 16:04+0100"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>"
"Language-Team: es <LL@li.org>"
"Plural-Forms: nplurals=2; plural=(n != 1)"
"MIME-Version: 1.0"
"Content-Type: text/plain; charset=utf-8"
"Content-Transfer-Encoding: 8bit"
"Generated-By: Babel 1.3"

#: templates/index.html:11
msgid "IBM QRadar Application: Hello World"
msgstr ""
```

You can use the `msgid` entries as your keys.

**Create the .po files**

You create individual language-specific .po files for Spanish, French, and English. From a command line, type the following commands from within the app/ folder:

```bash
pybabel init -i messages.pot -d translations -l es
pybabel init -i messages.pot -d translations -l fr
pybabel init -i messages.pot -d translations -l en
pybabel init -i messages.pot -d translations -l ja
```

These commands are used to create the translation files in the following locations:

- app/translations/es/LC_Messages/messages.po
- app/translations/fr/LC_Messages/messages.po
- app/translations/en/LC_Messages/messages.po
- app/translations/ja/LC_Messages/messages.po

The following example is the app/translations/es/LC_Messages/messages.po file that is generated:

```plaintext
msgid "" msgstr ""

"Project-Id-Version: PROJECT VERSION"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS"
"POT-Creation-Date: 2015-10-07 16:02+0100"
"PO-Revision-Date: 2015-10-07 16:04+0100"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>"
"Language-Team: es <LL@li.org>"
"Plural-Forms: nplurals=2; plural=(n != 1)"
"MIME-Version: 1.0"
"Content-Type: text/plain; charset=utf-8"
"Content-Transfer-Encoding: 8bit"
"Generated-By: Babel 1.3"

#: templates/index.html:11
msgid "IBM QRadar Application: Hello World"
msgstr ""
```

You can use the `msgid` entries as your keys.
Hello! and Welcome to the first Qradar app served from the "AppFramework/Docker instance on your console"

Edit the .po files

You edit the .po files to add the language-specific text strings that QRadar uses to translate your app's content. For each msgid in the .po file, you must enter a corresponding msgstr value in the target language.

The following example is a code snippet from the app/translations/es/LC_Messages/messages.po file:

Create the .mo files

To create the .mo files, open a command line and type the following command from within the app/ folder:

```
pybabel compile -d translation
```

This command compiles all your .po files into .mo files in the following locations:

- app/translations/es/LC_Messages/messages.mo
- app/translations/fr/LC_Messages/messages.mo
- app/translations/en/LC_Messages/messages.mo
- app/translations/ja/LC_Messages/messages.mo

The QRadar GUI app framework provides a default Flask environment that looks for locale-specific files in the sub directories of the app/translations/ folder.

To specify the UTF-8 encoded locales that your app supports, you can add a config.py file to the app/ folder. The file contains content similar to the following example:

```
# -*- coding: utf-8 -*-
#
# available languages
LANGUAGES = {
    'en': 'English',
    'es': 'Español',
    'fr': 'Français',
    'ja': '日本語'
}
```

This globalization support file helps QRadar to find the .mo file for each locale you specify.

After you create the .mo files, you can remove the .po, .pot, and babel.py files if you do not want these resources to be packaged with your app.
views.py

__author__ = 'IBM'

from flask import render_template, send_from_directory, request
from app import app
from flask.ext.babel import gettext
from config import LANGUAGES
from qpylib import qpylib

from flask.ext.babel import Babel
babel = Babel(app)  # 2

@babel.localeselector  # 3
def get_locale():
    return request.accept_languages.best_match(LANGUAGES.keys())

@app.route('/
@app.route('/index')
def index():
    qpylib.log(request.headers.get('Accept-Language', ''))
    return render_template("index.html", title = "QApp1 : Hello World !")

The following list describes content from the views.py code snippet:
1. The gettext method is imported from the Babel package. This line is optional but it is useful if you want to use locale text away from the python tier. In the Jinja2 template for this example, the gettext methods were used to extract key values.
2. The Flask app is injected into a Babel context so that your app can render locale-specific text.
3. This code applies the Babel localeselector decorator pattern across all your routes (in other words, any request that comes in from QRadar). The decorator uses the locales that are defined in the app/config.py file to connect the best-fit language-specific keys file to the incoming request.
27 Custom fragments injection points

Use custom fragment injection points to display custom information in QRadar.

You add the injection point information to the **app_name**, **page_id**, and **location** fields in the **fragments** block in the manifest file. The following table lists the injection point locations that you use in the **fragments** block for each tab and page.

For some QRadar pages, HTML fragments can be injected at the top (header) or bottom (footer) of the page. If injection points are not specified in the injection point name, the content is injected at the top of the page (header). On the **Admin** tab, you can inject content at each section header.

The following table shows examples of custom fragments.

<table>
<thead>
<tr>
<th>Tab and page</th>
<th>Sample manifest entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offenses &gt; My Offenses header</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;MyOffenseList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;header&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; My Offenses footer</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;MyOffenseList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;footer&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; All Offenses &gt; header</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;OffenseList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;header&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; All Offenses footer</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;OffenseList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;footer&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; By Category header</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;CategoryTypeSummaryList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Tab and page</td>
<td>Sample manifest entry</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Offenses &gt; By Source IP header</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;AttackerList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;header&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; By Source IP footer</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;AttackerList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;footer&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; By Destination IP</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td>header</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;TargetList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;header&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; By Destination IP</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td>footer</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;TargetList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;footer&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; By Network header</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;NetworkList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;header&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; By Network footer</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;NetworkList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;footer&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; Rules header</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;QRadar&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;RulesWizardExistingRules&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Offenses &gt; Offense Summary</td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td>header</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;SEM&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;OffenseSummary&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somerestendpoint&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Tab and page</td>
<td>Sample manifest entry</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Assets &gt; Asset Profiles header</strong></td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;Assets&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;AssetList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;header&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somertestendpoint&quot;</td>
</tr>
<tr>
<td><strong>Assets &gt; Asset Profiles footer</strong></td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;Assets&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;AssetList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;footer&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somertestendpoint&quot;</td>
</tr>
<tr>
<td><strong>Assets &gt; Server Discovery header</strong></td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;Assets&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;ServerDiscovery&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somertestendpoint&quot;</td>
</tr>
<tr>
<td><strong>Assets &gt; VA Scan header</strong></td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;Assets&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;VaScannerSchedulesList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;header&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somertestendpoint&quot;</td>
</tr>
<tr>
<td><strong>Assets &gt; VA Scan footer</strong></td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;Assets&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;VaScannerSchedulesList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;footer&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somertestendpoint&quot;</td>
</tr>
<tr>
<td><strong>Assets &gt; Asset Profiles &gt; Id &gt;</strong></td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td><strong>Asset Details dialog header</strong></td>
<td>&quot;app_name&quot;: &quot;Assets&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;AssetDetailsVulnList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;header&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somertestendpoint&quot;</td>
</tr>
<tr>
<td><strong>Assets &gt; Asset Profiles &gt; Id &gt;</strong></td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td><strong>Asset Details dialog footer</strong></td>
<td>&quot;app_name&quot;: &quot;Assets&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;AssetDetailsVulnList&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;location&quot;: &quot;footer&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somertestendpoint&quot;</td>
</tr>
<tr>
<td><strong>Admin &gt; System Configuration</strong></td>
<td>&quot;fragments&quot;: [</td>
</tr>
<tr>
<td></td>
<td>&quot;app_name&quot;: &quot;QRadar&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;page_id&quot;: &quot;systemConfiguration&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;rest_endpoint&quot;: &quot;/somertestendpoint&quot;</td>
</tr>
</tbody>
</table>
The following table describes custom fragments injection points that are not included in the Custom fragments injection points examples table.

### Table 42. Custom fragments injection points

<table>
<thead>
<tr>
<th>Tab and page</th>
<th>Injection point</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Admin</strong></td>
<td>QRadar.allTabs</td>
</tr>
<tr>
<td>Asset Details dialog</td>
<td>Assets.AssetDetailsVulnList.header</td>
</tr>
<tr>
<td></td>
<td>Assets.AssetDetailsVulnList.footer</td>
</tr>
<tr>
<td>Assets &gt; Server Discovery</td>
<td>Assets.ServerDiscovery</td>
</tr>
<tr>
<td>Dashboard</td>
<td>QRadar.Dashboard</td>
</tr>
<tr>
<td>Log Activity</td>
<td>EventViewer.EventList.header</td>
</tr>
<tr>
<td></td>
<td>EventViewer.EventList.footer</td>
</tr>
<tr>
<td>Network Activity</td>
<td>Surveillance.FlowList.footer</td>
</tr>
<tr>
<td>Reports &gt; All</td>
<td>Reports.ReportTemplateListAll.header</td>
</tr>
<tr>
<td></td>
<td>Reports.ReportTemplateListAll.footer</td>
</tr>
<tr>
<td>Reports &gt; Daily</td>
<td>Reports.ReportTemplateListDaily.header</td>
</tr>
<tr>
<td></td>
<td>Reports.ReportTemplateListDaily.footer</td>
</tr>
<tr>
<td>Reports &gt; Hourly</td>
<td>Reports.ReportTemplateListHourly.header</td>
</tr>
<tr>
<td></td>
<td>Reports.ReportTemplateListHourly.footer</td>
</tr>
<tr>
<td></td>
<td>Reports.ReportTemplateListManual.footer</td>
</tr>
<tr>
<td>Reports &gt; Monthly</td>
<td>Reports.ReportTemplateListMonthly.header</td>
</tr>
<tr>
<td></td>
<td>Reports.ReportTemplateListMonthly.footer</td>
</tr>
<tr>
<td>Tab and page</td>
<td>Injection point</td>
</tr>
<tr>
<td>------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Reports &gt; Weekly</td>
<td>Reports.ReportTemplateListWeekly.header</td>
</tr>
<tr>
<td></td>
<td>Reports.ReportTemplateListWeekly.footer</td>
</tr>
<tr>
<td>Risks &gt; Configuration Monitor</td>
<td>SRM.ConfigDeviceList</td>
</tr>
<tr>
<td>Risks &gt; Connections</td>
<td>QRadar.ArcList.header</td>
</tr>
<tr>
<td></td>
<td>QRadar.ArcList.footer</td>
</tr>
<tr>
<td>Risks &gt; Policy Management &gt; By Asset</td>
<td>SRM.ByAssetList</td>
</tr>
<tr>
<td>Risks &gt; Policy &gt; By Policy</td>
<td>SRM.ByPolicyList</td>
</tr>
<tr>
<td>Risks &gt; Policy Management &gt; By Policy Check</td>
<td>SRM.ByPolicyCheckList</td>
</tr>
<tr>
<td>Risks &gt; Policy Monitor</td>
<td>SRM.MaintainQuestions</td>
</tr>
<tr>
<td>Risks &gt; Simulation &gt; Simulations</td>
<td>SRM.SimulationList.header</td>
</tr>
<tr>
<td></td>
<td>SRM.SimulationList.footer</td>
</tr>
<tr>
<td>Risks &gt; Simulation &gt; Topology Models</td>
<td>SRM.ModelList.header</td>
</tr>
<tr>
<td></td>
<td>SRM.ModelList.footer</td>
</tr>
<tr>
<td>Risks &gt; Topology</td>
<td>SRM.NetworkTopology</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Administrative &gt; Operational Window</td>
<td>QVM.OperationalWindowList.header</td>
</tr>
<tr>
<td></td>
<td>QVM.OperationalWindowList.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Administrative &gt; Scan Exclusions</td>
<td>QVM.ExclusionList.header</td>
</tr>
<tr>
<td></td>
<td>QVM.ExclusionList.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Administrative &gt; Scan Policies</td>
<td>QVM.ScanPolicies.header</td>
</tr>
<tr>
<td></td>
<td>QVM.ScanPolicies.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Administrative &gt; Scan Profiles</td>
<td>QVM.ScanProfileList.header</td>
</tr>
<tr>
<td></td>
<td>QVM.ScanProfileList.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Administrative &gt; Scanners</td>
<td>QVM.ScannersAdminList.header</td>
</tr>
<tr>
<td></td>
<td>QVM.ScannersAdminList.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Administrative &gt; Scheduled Scans</td>
<td>QVM.ScheduledScans</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Manage</td>
<td>assetprofile.ByVulnerabilityInstanceList.header</td>
</tr>
<tr>
<td></td>
<td>assetprofile.ByVulnerabilityInstanceList.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Manage &gt; By Asset</td>
<td>assetprofile.ByAssetListForm.header</td>
</tr>
<tr>
<td></td>
<td>assetprofile.ByAssetListForm.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Manage &gt; By Network</td>
<td>assetprofile.ByNetworkList.header</td>
</tr>
<tr>
<td></td>
<td>assetprofile.ByNetworkList.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Manage &gt; By Open Service</td>
<td>assetprofile.ByOpenServiceList.header</td>
</tr>
<tr>
<td></td>
<td>assetprofile.ByOpenServiceList.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; Manage &gt; By Vulnerability</td>
<td>assetprofile.ByVulnerabilityList.header</td>
</tr>
<tr>
<td></td>
<td>assetprofile.ByVulnerabilityList.footer</td>
</tr>
<tr>
<td>Vulnerabilities &gt; My Assigned</td>
<td>assetprofile.MyAssignedVulnerabilitiesList.header</td>
</tr>
<tr>
<td></td>
<td>assetprofile.MyAssignedVulnerabilitiesList.footer</td>
</tr>
</tbody>
</table>

Table 42. Custom fragments injection points (continued)

27 Custom fragments injection points 117
<table>
<thead>
<tr>
<th>Tab and page</th>
<th>Injection point</th>
</tr>
</thead>
</table>
| Vulnerabilities > Research > Advisories | QVM.ResearchAdvisoriesList.header  
                      | QVM.ResearchAdvisoriesList.footer                   |
| Vulnerabilities > Research > News  | QVM.ResearchNewsList.header                            
                      | QVM.ResearchNewsList.footer                           |
| Vulnerabilities > Research > Vulnerabilities | assetprofile.ResearchVulnerabilityList.header  
                      | assetprofile.ResearchVulnerabilityList.footer         |
| Vulnerabilities > Scan Results     | QVM.ScanResultsList.header                            
                      | QVM.ScanResultsList.footer                            |
| Vulnerabilities > Vulnerability Assignment | assetprofile.AssetOwner.header  
                       | assetprofile.AssetOwner.footer                        |
| Vulnerabilities > Vulnerability Exception | assetprofile.ExceptionRulesList.header  
                          | assetprofile.ExceptionRulesList.footer                |
28 Custom column injection points

Use custom column injection points to add custom columns in QRadar.

The following table lists the custom column injection points that you can use in your app. The page IDs are used in the `page_id` field in the `custom_columns` block in the app’s manifest file.

*Table 43. Custom column injection point examples*

<table>
<thead>
<tr>
<th>Page ID</th>
<th>Table Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssetDetailsVulnList</td>
<td>Assets &gt; IP Address column to open Asset Details page and Vulnerabilities table</td>
</tr>
<tr>
<td>AssetList</td>
<td>Asset tab list</td>
</tr>
<tr>
<td>AttackerList</td>
<td>Offenses &gt; By Source IP</td>
</tr>
<tr>
<td>ByAssetListForm</td>
<td>Vulnerabilities &gt; Manage vulnerabilities &gt; by Asset</td>
</tr>
<tr>
<td>ByNetworkList</td>
<td>Vulnerabilities &gt; Manage vulnerabilities &gt; by Network</td>
</tr>
<tr>
<td>ByOpenServiceList</td>
<td>Vulnerabilities &gt; Manage vulnerabilities &gt; by Open Service</td>
</tr>
<tr>
<td>ByVulnerabilityInstanceList</td>
<td>Vulnerabilities &gt; Manage vulnerabilities</td>
</tr>
<tr>
<td>ByVulnerabilityList</td>
<td>Vulnerabilities &gt; Manage vulnerabilities &gt; by Vulnerability</td>
</tr>
<tr>
<td>ExceptionRulesList</td>
<td>Vulnerabilities &gt; Vulnerability Exception</td>
</tr>
<tr>
<td>FlowsSourceList</td>
<td>Admin &gt; Flow Sources</td>
</tr>
<tr>
<td>MyAssignedVulnerabilitiesList</td>
<td>Vulnerabilities &gt; My Assigned Vulnerabilities</td>
</tr>
<tr>
<td>NetworkList</td>
<td>Offenses &gt; By Network</td>
</tr>
<tr>
<td>NetworkOffenseList</td>
<td>Offenses &gt; By Network Double-click an offense.</td>
</tr>
<tr>
<td>OffenseList</td>
<td>Offenses tab main page</td>
</tr>
<tr>
<td>ReferenceSetElems</td>
<td>Admin &gt; Reference Set Management. Double-click row to open content window &gt; Content tab.</td>
</tr>
<tr>
<td>ReferenceSets</td>
<td>Admin &gt; Reference Set Management</td>
</tr>
<tr>
<td>SensorDeviceList</td>
<td>Admin &gt; Log Sources</td>
</tr>
<tr>
<td>TargetList</td>
<td>Offenses &gt; By Destination IP</td>
</tr>
<tr>
<td>VaScannerSchedulesList</td>
<td>Admin &gt; Schedule VA Scanners</td>
</tr>
<tr>
<td>VaScannersList</td>
<td>Admin &gt; VA Scanners</td>
</tr>
</tbody>
</table>
29 Custom actions for CRE responses

You can add your own script that runs as a part of a custom action when a custom rules engine (CRE) rule is triggered.

The following scripting languages are supported:

• Bash version 4.1.2
• Perl version 5.10.1
• Python version 2.7.9

You can use base libraries in these languages to do custom operations that use data that is passed directly from the event that triggered the rule.

Create custom actions by using the Define Actions window on the Admin tab. You can also create custom actions by using the /api/analytics/custom_actions REST endpoints. The following sample is an example of a custom action JSON file that the GET /api/analytics/custom_actions/actions endpoint returns.

```
{
  "id": 1004,
  "interpreter": 1,
  "description": "Custom action containing two parameters",
  "name": "custom_action_1",
  "script": 43,
  "parameters": [
    {
      "encrypted": false,
      "name": "fixedParam",
      "value": "Hello World!",
      "parameter_type": "fixed"
    },
    {
      "encrypted": false,
      "name": "dynamicParam",
      "value": "sourceip",
      "parameter_type": "dynamic"
    }
  ]
}
```

The two JSON objects that are contained within the parameters field represent parameters, which are passed to your script when it is run. Two types of parameters are supported:

• **Fixed parameters** represent fixed values that are passed to your script as is. For example, if the `fixedParam` parameter has a value of "Hello World!" when accessed from your script, this parameter returns the value "Hello World!".

• **Dynamic parameters** and their corresponding `value` fields represent properties that are extracted from the event that triggered the CRE rule. For example, if the `dynamicParam` parameter has a value of "sourceip" when passed to your script, this value is replaced with the corresponding source IP address that is contained within the rule that triggers the event.

Parameters are passed to scripts in the order that they are defined within the custom action. These parameters can then be accessed by using the supported methods for each language:

**Bash**

```
param1=$1 # First parameter
param2=$2 # Second parameter
```
Perl
$param1 = $ARGV[0]
$param2 = $ARGV[1]

Python
import sys
param1 = sys.argv[1]
param2 = sys.argv[2]

Example: Making a REST call to an external server

To write a script that makes a REST call to an external server when a rule is triggered, create a script that passes the IP address to the external server. The following JSON file is an example custom action:

```json
{"id": 1004,
  "interpreter": 1,
  "description": "Custom action containing two parameters",
  "name": "custom_action_1",
  "script": 43,
  "parameters": [
    {
      "encrypted": false,
      "name": "serverIP",
      "value": "10.100.78.11",
      "parameter_type": "fixed"
    },
    {
      "encrypted": false,
      "name": "username",
      "value": "admin",
      "parameter_type": "dynamic"
    },
    {
      "encrypted": true,
      "name": "password",
      "value": "ASDB231434DSDK#05DA23SDD1",
      "parameter_type": "dynamic"
    },
    {
      "encrypted": false,
      "name": "offendingIP",
      "value": "sourceip",
      "parameter_type": "dynamic"
    }
  ]
}

The following Bash script uses these parameters to pass the IP address to an external server.

```
#!/bin/bash
# Assign parameters to variables.
serverAddress=$1
username=$2
password=$3
offendingIP=$4
# Call to an external server REST endpoint using the supplied parameters.
curl -u $username: $password -i -H "Accept: application/json" -X POST
   -d "ip= $offendingIP" "https://" $serverAddress/some_service
```

Defining custom actions

You can attach scripts to custom rules that do custom actions in response to network events. Use the Custom Action window to manage custom action scripts.
Use custom actions to select or define the value that is passed to the script and to define the resulting action.

For example, you can write a script to create a firewall rule that blocks a source IP address from your network in response to a rule that is triggered by a defined number of failed login attempts.

The following examples are custom actions that are the outcomes of passing values to a script:
- Block users and domains.
- Initiate work flows and updates in external systems.
- Update TAXI servers with a STIX representation of a threat.

Custom actions work best with a low volume of events and with custom rules that have a low response limiter value.

Take the following steps to define your custom actions:
1. From the Admin tab, click the Define Actions icon.
2. Click Add on the Custom Action window toolbar to open the Define Custom Action dialog where you can upload scripts that define custom actions.
3. Select a programming language version that the product supports from the Interpreter list.
4. Select and name a parameter from the following table to pass to the script that you upload.

Table 44. Custom action parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed property</td>
<td>Values that are passed to the custom action script. Not based on the events or flows, but are based on other defined values that you can use the script to act on. For example, the fixed properties username and password for a third-party system are passed to a script that results in sending an SMS alert, or other defined action. You can encrypt fixed properties, such as passwords, by selecting the Encrypt value check box.</td>
</tr>
<tr>
<td>Network event property</td>
<td>Dynamic Ariel properties that are generated by events. Select from the Property list. For example, the network event property sourceip provides a parameter that matches the source IP address of the triggered event.</td>
</tr>
</tbody>
</table>

In order to ensure the security of your deployment, the product does not support the full range of scripting functionality that is provided by the Python, Perl or Bash languages.

Parameters are passed into your script in the order in which you added them in the Define Custom Action dialog box.

Testing your custom action

You can test whether your script runs successfully before you associate it with a rule. Select a custom action and click Test Execution > Execute to test your script. The Test custom action execution dialog returns the result of the test and any output that is produced by the script.
Custom action scripts are executed inside a sand-boxed environment on your managed hosts. If you need to write to disk from a custom action script, you must use the following directory: /home/customactionuser. Custom action scripts execute on the managed host that runs the event processor that triggered the rule.

After you configure and test your custom action, use the Rule Wizard to create a new event rule and associate the custom action with it.

**Testing your custom action**

Test whether your script runs successfully before you associate it with a rule.

**Procedure**

1. From the Admin tab, click the Define Actions icon.
2. Select a custom action.
3. Click Test Execution > Execute to test your script.

   Custom action scripts are run inside a sandbox environment on your managed hosts. If you write to disk from a custom action script, you must use the /home/customactionuser directory. Custom action scripts run on the managed host that runs the event processor that triggered the rule.

**What to do next**

After you configure and test your custom action, use the Rule Wizard to create a new event rule and associate the custom action with it.

**Adding a custom action script to an event rule**

You use the Rule Wizard to add a custom action script that runs in response to a custom rule event.

**About this task**

To create a new rule, you must have the Offenses > Maintain Custom Rules permission.

You can test rules locally or globally. A local test means that rule is tested on the local Event processor and not shared with the system. A global test means that the rule is shared and tested by any Event processor on the system. Global rules send events to the central Event processor, which might decrease performance on the central Event processor.

**Procedure**

1. Click the Offenses tab.
2. On the navigation menu, click Rules.
3. From the Actions list, select New Event Rule.
4. In the Rule Test Stack Editor page, type a unique name for this rule in the enter rule name here field in the Rule pane.
5. From the list box, select Local or Global.
6. Add one or more tests to a rule:
   a. To filter the options in the Test Group list box, type the text that you want to filter for in the Type to filter field.
   b. From the Test Group list box, select the type of test you want to add to this rule.
   c. For each test you want to add to the rule, select the plus (+) sign beside the test.
   d. To exclude a test, click and at the beginning of the test in the Rule pane. The and is displayed as and not.
e. Click the underlined configurable parameters to customize the variables of the test.
f. From the dialog box, select values for the variable, and then click Submit.

7. To export the configured rule as a building block to use with other rules:
   a. Click Export as Building Block.
   b. Type a unique name for this building block.
   c. Click Save.

8. On the Groups pane, select the check boxes of the groups to which you want to assign this rule.

9. In the Notes field, type a note that you want to include for this rule. Click Next.

10. On the Rule Responses page, click the Execute Custom Action check box and select your script from the Custom Action to execute drop-down list.

11. Click Next.

12. Review the Rule Summary, and then click Finish.

### Custom action REST API endpoints

Custom action endpoints are available in the IBM QRadar API that you can use to aid application development.

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /analytics/custom_actions/actions</td>
<td>fields, range, filter</td>
<td>Retrieves a list of available custom actions.</td>
</tr>
<tr>
<td>POST /analytics/custom_actions/actions</td>
<td>fields, custom_action</td>
<td>Creates a new custom action with the supplied fields.</td>
</tr>
<tr>
<td>GET /analytics/custom_actions/actions/{action_id}</td>
<td>action_id, fields</td>
<td>Retrieves a custom action based on the supplied action_id.</td>
</tr>
<tr>
<td>POST /analytics/custom_actions/actions/{action_id}</td>
<td>action_id, fields, custom_action</td>
<td>Updates an existing custom action.</td>
</tr>
<tr>
<td>DELETE /analytics/custom_actions/actions/{action_id}</td>
<td>action_id</td>
<td>Deletes an existing custom action.</td>
</tr>
<tr>
<td>GET /analytics/custom_actions/interpreters</td>
<td>fields, Range, filter</td>
<td>Retrieves a list of available custom action interpreters.</td>
</tr>
<tr>
<td>GET /analytics/custom_actions/interpreters/{interpreter_id}</td>
<td>interpreter_id, fields</td>
<td>Retrieves a custom action interpreter based on the interpreter ID.</td>
</tr>
<tr>
<td>GET /analytics/custom_actions/scripts</td>
<td>application id</td>
<td>Retrieves a list of meta-data for available custom action script files.</td>
</tr>
<tr>
<td>POST /analytics/custom_actions/scripts</td>
<td>fields, file</td>
<td>Creates a new custom action script file.</td>
</tr>
<tr>
<td>GET /analytics/custom_actions/scripts/{script_id}</td>
<td>script_id, fields</td>
<td>Retrieves meta-data of a custom action script file based on supplied script_id.</td>
</tr>
<tr>
<td>POST /analytics/custom_actions/scripts/{script_id}</td>
<td>script_id, fields, file</td>
<td>Updates an existing custom action script file.</td>
</tr>
<tr>
<td>DELETE /analytics/custom_actions/scripts/{script_id}</td>
<td>script_id</td>
<td>Deletes an existing custom action script file.</td>
</tr>
</tbody>
</table>

For more information, see the API documentation page on your QRadar Console: https://<Console_IP>/api_doc. Alternatively, see the IBM QRadar API Guide.
30 Custom AQL functions

You can create IBM QRadar apps that use custom Ariel Query Language (AQL) functions.

After you upload the app, you can use these custom functions in AQL statements in advanced searches, API calls, and application apps. For more information about AQL, see the IBM QRadar Ariel Query Language Guide.

Coding tips

Before you implement custom AQL functions, consider these items:

- Scripts are not throttled and cannot be canceled. Be careful of infinite loops and resource leaks in your code.
- Deletion of custom functions is possible, but not supported.
- QRadar parses AQL strings much more than it needs to. Expensive `init_function_name` implementations that are combined with the use of your function in a literal context can be expensive. Use with caution.
- The `execute_function_name` implementation must be thread-safe. Use the `Utils.concurrent` library to ensure thread safety.

Custom AQL function management

Custom AQL functions are uploaded and updated by using the QRadar Extension Manager on the Admin tab. For more information about the QRadar Extension Manager, see the IBM QRadar Administration Guide.

The following XML file is an example of a QRadar extension that defines a custom AQL function.

```xml
<content>
  <custom_function>
    <namespace>application</namespace>
    <name>concat</name>
    <return_type>string</return_type>
    <parameter_types>string string</parameter_types>
    <execute_function_name>calculate</execute_function_name>
    <script_engine>javascript</script_engine>
    <varargs>false</varargs>
    <script>
      function calculate(input1, input2) {
        return input1 + input2;
      }
    </script>
    <username>user1</username>
  </custom_function>
</content>
```

You can also use the Content Management Tool, which is a command-line tool, to upload extensions:

```
/opt/qradar/bin/contentManagement.pl -a update -f my_bundle.xml
```

The following code samples provide examples of how you can create QRadar apps that define custom AQL functions for use in AQL statements.
Example: Custom functions in AQL statements

AQL statements that use custom functions use the following basic format:

```
SELECT <CUSTOM FUNCTION NAMESPACE>::<CUSTOM FUNCTION NAME>({<INPUTS>}) FROM <TABLE>
```

SELECT application::concat('This is my IP: ', sourceip) FROM events LIMIT 20

Example: Simple addition

1. Create the following QRadar extension:

```xml
<content>
  <custom_function>
    <namespace>application</namespace>
    <name>add</name>
    <return_type>number</return_type>
    <parameter_types>number number</parameter_types>
    <execute_function_name>execute</execute_function_name>
    <script_engine>javascript</script_engine>
    <script>
      function execute(input1, input2)
      {
        return input1 + input2;
      }
    </script>
  </custom_function>
</content>
```

2. Use the QRadar Extensions Management to upload the extension to your QRadar Console. For more information about the QRadar Extensions Management, see “Extensions management” on page 56.

If you have multiple hosts, wait up to 60 seconds for the function to propagate across your deployment.

You can use this function in AQL statements as follows:

```
SELECT application::add(eventcount, 5) FROM events
```

Example: Variable argument concatenation

1. Create the following QRadar extension:

```xml
<content>
  <custom_function>
    <namespace>application</namespace>
    <name>concat</name>
    <return_type>string</return_type>
    <parameter_types>string</parameter_types>
    <varargs>true</varargs>
    <execute_function_name>execute</execute_function_name>
    <script_engine>javascript</script_engine>
    <script>
      function execute()
      {
        var result = "";
        for(var i=0; i<arguments.length; i++)
          result = result + arguments[i];
        return result;
      }
    </script>
  </custom_function>
</content>
```

2. Use the QRadar Extension Manager to upload the extension to your QRadar Console. For more information about the QRadar Extension Manager, see “Extensions management” on page 56.

If you have multiple hosts, wait up to 60 seconds for the function to propagate across your deployment.
You can use this function in AQL statements:
SELECT application::concat(sourceip, ':', sourceport) FROM events

**Example: Complex initialization that uses a remote API**

1. Create the following QRadar extension:
   ```xml
   <content>
   <custom_function>
   <nameSpace>application</nameSpace>
   <name>isFlaggedIP</name>
   <return_type>boolean</return_type>
   <parameter_types>host</parameter_types>
   <execute_function_name>isFlaggedIP</execute_function_name>
   <init_function_name>onInit</init_function_name>
   <script_engine>javascript</script_engine>
   <script>
   var flaggedIPs;
   function onInit()
   {
   var properties = Utils.config.readNamespacePropertiesFile('my_api.properties');
   var ip = properties.get('ip');
   var url = 'https://' + ip + '/my_api/flagged_ips';
   var username = properties.get('username');
   var token = properties.get('token');
   var headers = {'SecurityToken': token};
   var jsonResponse = JSON.parse(Utils.http.invokeHTTP("GET", url, 200, headers, null, null));
   flaggedIPs = [];
   for(var i in jsonResponse)
   {
   flaggedIPs.push(jsonResponse[i]);
   }
   }
   function isFlaggedIP(ip)
   {
   return (flaggedIPs.indexOf(ip) >= 0);
   }
   </script>
   <username>user1</username>
   </custom_function>
   </content>
   
2. Use the QRadar Extension Manager on the **Admin** tab to upload the extension to your QRadar Console. For more information about the QRadar Extension Manager, see "Extensions management" on page 56.
   If you have multiple hosts, wait up to 60 seconds for the function to propagate across your deployment.

3. Create a properties file with the following content in the /store/custom_functions/namespaces/application/my_api.properties file.
   ```xml
   <ip>ip for remote API server</ip>
   <username>your username for remote API server</username>
   <token>your token for remote API server</token>
   
   You can use this function in AQL statements:
SELECT application::isFlaggedIP(sourceip) AS is_flagged_qradar_ip, sourceip FROM events

**Custom AQL function fields**

Multiple fields are available to custom AQL functions.
**namespace and name**

Required: Yes  
Default:  
Case-sensitive: No  
Whitespace-sensitive: Yes

The following table describes the namespace and name fields for custom AQL functions.

*Table 45. Namespace and name fields*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
<td>The first component that makes up the identifier of a custom AQL function.</td>
</tr>
<tr>
<td>name</td>
<td>The second component that makes up the identifier of a custom AQL function.</td>
</tr>
</tbody>
</table>

Both values are case-insensitive and must be unique from all other custom functions. The Ariel database uses a concatenation of these fields to expose the custom function through AQL. For example:

- namespace: :MyNamespace
- name::MyFunction
- AQL: MyNamespace: MyFunction

When you specify your custom function through AQL, you must employ double quotation marks when white space or special characters are used.

**return_type**

You use the return_type field to declare the script type that your custom function returns.

Required: Yes  
Default:  
Case-sensitive: No  
Whitespace-sensitive: No  
Example: String

The return value is automatically converted to the internal Java type that is interpreted by Ariel (QRadar_type).

The following table describes acceptable script types for the return_type field.

*Table 46. return_type field acceptable script types*

<table>
<thead>
<tr>
<th>Type</th>
<th>Script type</th>
<th>QRadar type</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>String</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Number</td>
<td>java.lang.Number</td>
</tr>
<tr>
<td>LONG</td>
<td>Number</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>HOST</td>
<td>String</td>
<td>com.q1labs.core.dao.util.Host</td>
</tr>
<tr>
<td>PORT</td>
<td>Number</td>
<td>com.q1labs.frameworks.nio.Port</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Boolean</td>
<td>java.lang.Boolean</td>
</tr>
</tbody>
</table>

**parameter_types**

You use the parameter_types field to declare the type of each parameter that your custom function accepts.
Required: Yes
Default: Case-sensitive: No
Whitespace-sensitive: No
Example: String, Number, Boolean

The incoming Java object is automatically converted to your declared script type.

The following table describes acceptable script types for the parameter_types field.

<table>
<thead>
<tr>
<th>Type</th>
<th>Script type</th>
<th>QRadar type</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>String</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Number</td>
<td>java.lang.Number</td>
</tr>
<tr>
<td>LONG</td>
<td>Number</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>HOST</td>
<td>String</td>
<td>com.q1labs.core.dao.util.Host</td>
</tr>
<tr>
<td>PORT</td>
<td>Number</td>
<td>com.q1labs.frameworks.nio.Port</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Boolean</td>
<td>java.lang.Boolean</td>
</tr>
</tbody>
</table>

Generally, when you pass parameters from AQL, most values are interoperable. If you find they are not, you can use AQL casting functions such as LONG and PORT.

Note: You can use an empty string if you accept no parameters.

varargs

If "true", variable arguments are used with your parameter_types value. You can specify the last type in your parameter_types string zero or more times.

If your parameter_types value is empty, this value has no meaning.

Required: No
Default: false
Case-sensitive: Yes
Whitespace-sensitive: Yes
Example: true

The way that you handle variable arguments depends on the script engine that you use. For JavaScript, you can use the built-in arguments variable:

```javascript
function myFunction() {
    for(var i=0; i<arguments.length; i++)
    {
        //Do something with arguments[i]
    }
}
```

execute_function_name

The value of this field maps to the execution function in your custom function script. The Java programming language uses the parameters in your AQL string to call this function. It then uses the value that is returned in the calculations.

The execution function is called concurrently. It must be thread-safe.
init_function_name

The value of this optional field maps to the initialization function in your custom function script. The Java programming language must call this function only once for a search. Place your expensive initialization code in this function so that your execution function is faster. Because this function is called once, it does not need to be thread-safe.

Required: No
Default: No
Case-sensitive: Yes
Whitespace-sensitive: Yes
Example: onInit

finish_function_name

The value of this optional field maps to the finish function in your custom function script. The Java programming language calls this function only once for a search. However, there might be some situations in which it is never called. Close any resources that you hold onto within your script to prevent resource leaks. Because this function is called only once, it does not need to be thread-safe.

Required: No
Default: No
Case-sensitive: Yes
Whitespace-sensitive: Yes
Example: onFinish

script_engine

The value of this field indicates the script engine that is used to interpret and invoke your script. The only supported script engine is JavaScript.

Required: Yes
Default: No
Case-sensitive: No
Whitespace-sensitive: No
Example: javascript

script

The value of this field is the code for your custom function. It is interpreted based on the script_engine that you specified.

You can place any code in here that you like, including other functions. However, the execute_function_name must exist and must be a valid function.

Required: Yes
Default: Yes
Case-sensitive: Yes
Whitespace-sensitive: Yes
Example: function addOne(value) {return value + 1;}

username

The value of this field currently has no implementation. It makes reference to your IBM QRadar username.

Required: Yes
Default:
Case-sensitive: Yes
Whitespace-sensitive: Yes
Example: administrator

Custom AQL function utilities

Use custom AQL function utilities.

Utils.concurrency.readLock
/**
 * Acquire a read lock against the global lock.
 * If the write lock is not held by another thread, this returns immediately.
 * If the write lock is held by another thread then the current thread becomes
 * disabled for thread scheduling purposes and lies dormant until
 * the read lock has been acquired.
 */
void readLock()

JavaScript example:
Utils.concurrency.readLock();
try { /*Do something*/ }
finally { Utils.concurrency.readUnlock(); }

Utils.concurrency.readLock(String name)
/**
 * Acquire a read lock against the given name.
 * If a lock by the given name does not exist yet, it will be created.
 * If the write lock is not held by another thread, this returns immediately.
 * If the write lock is held by another thread then the current thread becomes
 * disabled for threadscheduling purposes and lies dormant until
 * the read lock has been acquired.
 *
 * @param name The name of the lock to acquire a read lock against.
 * @param null, global lock is used.
 */
void readLock(String name)

JavaScript example:
Utils.concurrency.readLock("my_lock");
try { /*Do something*/ }
finally { Utils.concurrency.readUnlock("my_lock"); }

Utils.concurrency.tryReadLock(long timeoutMsec)
/**
 * Attempts to acquire a read lock against the global lock.
 * If the write lock is not held by another thread, this returns immediately.
 */

30 Custom AQL functions
* If the write lock is held by another thread, this will wait
  * a maximum of timeoutMsec milliseconds to acquire the read lock.
  *
  * @param timeoutMsec The maximum time to wait (in milliseconds) to acquire the lock
  * <= 0 can be used to return immediately if the read lock can not be acquired.
  * @return True if the read lock has been acquired, false otherwise.
  */
boolean tryReadLock(long timeoutMsec)

JavaScript example:
if(Utils.concurrency.tryReadLock(1000))
{
   try { /* Lock acquired */}
   finally { Utils.concurrency.readUnlock();}
}
else
   throw "Failed to acquire lock in reasonable time";

Utils.concurrency.tryReadLock(String name, long timeoutMsec)
/**
 * Attempts to acquire a read lock against the given name.
 * If the write lock is not held by another thread, this returns immediately.
 * If the write lock is held by another thread, this will
 * wait a maximum of timeoutMsec milliseconds to acquire the read lock.
 * @param name The name of the lock to acquire a read lock against.
 * If null, global lock is used.
 * @param timeoutMsec The maximum time to wait (in milliseconds)
 * to acquire the lock.
 * <= 0 can be used to return immediately if the read lock can not be acquired.
 * @return True if the read lock has been acquired, false otherwise.
 */
boolean tryReadLock(String name, long timeoutMsec)

JavaScript example:
if(Utils.concurrency.tryReadLock("my_lock", 1000))
{
   try { /* Lock acquired */}
   finally { Utils.concurrency.readUnlock("my_lock");}
}
else
   throw "Failed to acquire lock in reasonable time";

Utils.concurrency.readUnlock()
/**
 * Releases the global read lock held by the current thread.
 * You must currently be holding the global read lock,
 * or an exception occurs.
 * If the number of read locks held by all threads is now 0,
 * then the lock is made available for write lock attempts.
 */
void readUnlock()

JavaScript example:
Utils.concurrency.readLock();
try { /*Do something*/}
finally { Utils.concurrency.readUnlock();}
Utils.concurrency.readUnlock(String name)
/**
 * Releases the read lock against the given name held by the current thread.
 * You must currently be holding the read lock by the given name,
 * or an exception occurs.
 * If the number of read locks held by all threads is now 0,
 * then the lock is made available for write lock attempts.
 * @param name The name of the lock to acquire a read lock against.
 * If null, global lock is used.
 */
void readUnlock(String name)

JavaScript example:
Utils.concurrency.readLock("my_lock");
try {
 /*Do something*/
} finally {
Utils.concurrency.readUnlock("my_lock");
}

Utils.concurrency.writeLock()
/**
 * Acquire a write lock against the global lock.
 * If neither the read nor write lock are held by another thread,
 * this returns immediately.
 * If the write lock is held by another thread,
 * then the current thread becomes disabled for thread
 * scheduling purposes and lies dormant until
 * the read lock has been acquired.
 */
void writeLock()

JavaScript example:
Utils.concurrency.writeLock();
try {
 /*Do something*/
} finally {
Utils.concurrency.writeUnlock();
}

Utils.concurrency.writeLock(String name)
/**
 * Acquire a write lock against the given name.
 * If neither the read nor write lock are held by another thread,
 * this returns immediately.
 * If the write lock is held by another thread then the current thread
 * becomes disabled for thread scheduling purposes and lies dormant
 * until the read lock has been acquired.
 * @param name The name of the lock to acquire a write lock against.
 * If null, global lock is used.
 */
void writeLock(String name)

JavaScript example:
Utils.concurrency.writeLock("my_lock");
try {
 /*Do something*/
} finally {
Utils.concurrency.writeUnlock("my_lock");
}

Utils.concurrency.tryWriteLock(long timeoutMsec)
/**
 * Attempts to acquire a write lock against the global lock.
 * If neither the read nor write lock are held by another thread,
**tryWriteLock(long timeoutMsec)**

JavaScript example:

```javascript
if(Utils.concurrency.tryWriteLock(1000))
{
    try {
        /* Lock acquired */
    } finally { Utils.concurrency.writeUnlock(); }
}
else
    throw "Failed to acquire lock in reasonable time";
```

**tryWriteLock(String name, long timeoutMsec)**

JavaScript example:

```javascript
if(Utils.concurrency.tryWriteLock("my_lock", 1000))
{
    try {
        /* Lock acquired */
    } finally { Utils.concurrency.writeUnlock("my_lock"); }
}
else
    throw "Failed to acquire lock in reasonable time";
```

**writeUnlock()**

JavaScript example:

```javascript
Utils.concurrency.writeUnlock()
```

/*
* this returns immediately.
* If either the read or write lock is held by another thread,
* this will wait a maximum of timeoutMsec milliseconds
* to acquire the write lock.
* @param timeoutMsec The maximum time to wait (in milliseconds)
* to acquire the lock
* <= 0 can be used to return immediately,
* if the write lock can not be acquired.
* @return True if the write lock has been acquired, false otherwise.
*/

boolean tryWriteLock(long timeoutMsec)

**/**
* Attempts to acquire a write lock against the given name.
* If neither the read nor write lock are held by another thread
* this returns immediately.
* If either the read or write lock is held by another thread,
* this will wait a maximum of timeoutMsec milliseconds
* to acquire the write lock.
* @param timeoutMsec The maximum time to wait (in milliseconds)
* to acquire the lock.
* <= 0 can be used to return immediately if the write lock can not be acquired.
* @param name The name of the lock to acquire a write lock against.
* If null, global lock is used.
* @return True if the write lock has been acquired, false otherwise.
*/

boolean tryWriteLock(String name, long timeoutMsec)

**/**
* Releases the global write lock held by the current thread.
* You must currently be holding the global write lock, or an exception occurs.
* If the number of write locks held by all threads is now 0,
* then the lock is made available for write lock attempts.
*/

void writeUnlock()
Utils.concurrency.writeLock("my_lock");
try { /*Do something*/ }
finally { Utils.concurrency.writeUnlock("my_lock"); }

**Utils.concurrency.createAtomicBoolean()**

```java
/**
 * @return A new instance of java.util.concurrent.atomic.AtomicBoolean
 */
java.util.concurrent.atomic.AtomicBoolean createAtomicBoolean()
```

JavaScript example:
```javascript
var atomicBoolean = Utils.concurrency.createAtomicBoolean();
if(atomicBoolean.getAndSet(true))
    return "Value has not changed";
```

**Utils.concurrency.createAtomicInteger()**

```java
/**
 * @return A new instance of java.util.concurrent.atomic.AtomicInteger
 */
java.util.concurrent.atomic.AtomicInteger createAtomicInteger()
```

JavaScript example:
```javascript
var atomicInt =
   Utils.concurrency.createAtomicInteger();atomicInt.set(5);
```

**Utils.concurrency.createAtomicLong()**

```java
/**
 * @return A new instance of java.util.concurrent.atomic.AtomicLong
 */
java.util.concurrent.atomic.AtomicLong createAtomicLong()
```

JavaScript example:
```javascript
var atomicLong =
    Utils.concurrency.createAtomicLong();atomicLong.set(5)
```

**Utils.concurrency.createAtomicDouble()**

```java
/**
 * @return A new instance of java.util.concurrent.atomic.AtomicDouble
 */
java.util.concurrent.atomic.AtomicDouble createAtomicDouble()
```

JavaScript example:
```javascript
var atomicDouble =
    Utils.concurrency.createAtomicDouble();
atomicDouble.set(5);
```

**Utils.concurrency.createAtomicIntegerArray(int size)**

```java
/**
 * @param Size of the array. Must be >= 0
 * @return A new instance of java.util.concurrent.atomic.AtomicIntegerArray
 * with the given size
 */
java.util.concurrent.atomic.AtomicIntegerArray createAtomicIntegerArray(int size)
```

JavaScript example:
```javascript
var atomicIntArray =
    Utils.concurrency.createAtomicIntegerArray(5);
atomicIntArray.set(0, 25);
```
Utils.concurrency.createAtomicLongArray(int size)
/**
 * @param Size of the array. Must be >= 0
 * @return A new instance of java.util.concurrent.atomic.AtomicIntegerArray
 * with the given size
 */
java.util.concurrent.atomic.AtomicIntegerArray createAtomicIntegerArray(int size)

JavaScript example:
var atomicLongArray = Utils.concurrency.createAtomicLongArray(5);
atomicLongArray.set(0, 25);

Utils.concurrency.createAtomicDoubleArray(int size)
/**
 * @param Size of the array. Must be >= 0
 * @return A new instance of java.util.concurrent.atomic.AtomicDoubleArray
 * with the given size
 */
java.util.concurrent.atomic.AtomicDoubleArray createAtomicDoubleArray(int size)

JavaScript example:
var atomicDoubleArray = Utils.concurrency.createAtomicDoubleArray(5);
atomicDoubleArray.set(0, 25);

Utils.concurrency.createConcurrentMap()
/**
 * @return Returns a new instance of
 * java.util.concurrent.ConcurrentHashMap.ConcurrentHashMap,
 * which is a thread safe map that accepts any key or value
 */
java.util.concurrent.ConcurrentHashMap.ConcurrentHashMap createConcurrentMap()

JavaScript example:
var myMap = Utils.concurrency.createConcurrentMap();
myMap.put("some_key", "some_value");
myMap.put("some_other_key", 25);

Utils.concurrency.createConcurrentSet()
/**
 * @return Returns a new thread safe set that accepts any value,
 * backed by java.util.concurrent.ConcurrentHashMap.ConcurrentHashMap
 */
java.util.concurrent.ConcurrentHashMap.ConcurrentHashMap createConcurrentSet()

JavaScript example:
var mySet = Utils.concurrency.createConcurrentSet();
mySet.add("something");
mySet.add(25);

Utils.crypto.aesEncrypt(String data)
/**
 * Given a String, the frameworks AES encryption protocol will be invoked
 * to return an encrypted String.
 * @param data The String that will be encrypted.
 * @return The encrypted String. Null if null data was provided.
 */
String aesEncrypt(String data)

JavaScript example:
var myEncryptedData = Utils.crypto.aesEncrypt
   ("my plaintext data, preferably not stored here as a string");

Utils.crypto.aesDecrypt(String data)
/**
 * Given an String that is encrypted by the frameworks AES encryption protocol,
 * return the decrypted value.
 * @param data The String that was encrypted by the framework's
 * AES encryption protocol.
 * @return The decrypted String. Null if null data was provided.
 */
String aesDecrypt(String data)

JavaScript example:
var myData = Utils.crypto.aesDecrypt("my encrypted data");

Utils.general.base64Decode(String value)
/**
 * Given a String encoded as base64, return the original value
 * @param value The base64 value to decode
 * @return The decoded value. Null if value was null
 */
String base64Decode(String value)

JavaScript example:
var myAuth = Utils.general.base64Decode("my base64 encoded auth");

Utils.config.readNamespaceFile(String relPath)
/**
 * Given the path to a file relative to the namespace configuration
 * directory for the function assigned to this utility,
 * read it and return a String representation,
 * decoded using default character set of this JVM (typically UTF-8).
 * The file must exist, or an exception is thrown.
 * The file must be readable, or an exception is thrown.
 * If the relative path leads to a file that is not a child
 * of the proper configuration directory, an exception is thrown.
 * The full path structure is:
 * ${CUSTOM_FUNCTION_CONFIG_DIR}/namespaces/${namespace}/${relPath}
 * @param relPath The path to the file, relative to the namespace configuration
 * directory for the function assigned to this utility
 * @return A String representing the file's content in the default character set.
 */
String readNamespaceFile(String relPath)

JavaScript example:
var myFileContent = Utils.config.readNamespaceFile("test.txt");

Utils.config.readNamespaceFile(String relPath, boolean forceExist)
/**
 * Given the path to a file relative to the namespace configuration
 * directory for the function assigned to this utility,
 * read it and return a String representation,
 * decoded using default character set of this JVM (typically UTF-8).
 * The file must be readable, or an exception is thrown.
 * If the relative path leads to a file that is not a child
 * of the proper configuration directory, an exception is thrown.
 */
* The full path structure is:
  * /$\{CUSTOM_FUNCTION_CONFIG_DIR\}/namespaces/$\{namespace\}/$\{relPath\}
  *
  * @param relPath The path to the file, relative to the namespace configuration
directory for the function assigned to this utility
  *
  * @param forceExist If true, and the file does not exist, an exception if thrown.
  * @param charset If false, and the file does not exist, an empty Map is returned.
  * @return A String representing the file's content in the default character set.
  * An empty String if forceExist was false and the file does not exist.
  */
String readNamespaceFile(String relPath, boolean forceExist)

JavaScript example:
var myFileContent = Utils.config.readNamespaceFile("test.txt", false);

**Utils.config.readNamespaceFile(String relPath, boolean forceExist, String charset)**

/**
 * Given the path to a file relative to the namespace configuration
directory for the function assigned to this utility,
read it and return a String representation,
decoded using the given character set.
The file must be readable, or an exception is thrown.
If the relative path leads to a file that is not a child
of the proper configuration directory, an exception is thrown.
 *
 * The full path structure is:
 * /$\{CUSTOM_FUNCTION_CONFIG_DIR\}/namespaces/$\{namespace\}/$\{relPath\}
 *
 * @param relPath The path to the file, relative to the
namespace configuration directory for the function assigned to this utility
 *
 * @param forceExist If true, and the file does not exist, an exception if thrown.
 * @param charset The character set (encoding) to use when reading the file.
 * If null, the JVM default is used, usually UTF-8.
 *
 * @return A String representing the file's content in the given character set.
 * An empty String if forceExist was false and the file does not exist.
 */
String readNamespaceFile(String relPath, boolean forceExist, String charset)

JavaScript example:
var myFileContent = Utils.config.readNamespaceFile("test.txt", false, "UTF-8");

**Utils.config.readNamespacePropertiesFile(String relPath)**

/**
 * Given the path to a file relative to the namespace configuration
directory for the function assigned to this utility,
read it and return a Map of properties.
The format is assumed to be the Java compatible key/value pair format.
The file must exist, or an exception is thrown.
The file must be readable, or an exception is thrown.
If the relative path leads to a file that is not a child
of the proper configuration directory, an exception is thrown.
 *
 * The full path structure is:
 * /$\{CUSTOM_FUNCTION_CONFIG_DIR\}/namespaces/$\{namespace\}/$\{relPath\}
 *
 * @param relPath The path to the file, relative to the namespace configuration
directory for the function assigned to this utility.
 */
* @param forceExist If true, and the file does not exist, an exception if thrown.
* If false, and the file does not exist, an empty Map is returned.
* @return A map of properties read from the file
*/
Map<String, String> readNamespacePropertiesFile(String relPath)

JavaScript example:
var myProperties = Utils.config.readNamespacePropertiesFile("test.properties");
var myValue = myProperties.get("my_key");

Utils.config.readNamespacePropertiesFile(String relPath, boolean forceExist)
/**
 * Given the path to a file relative to the namespace configuration
 * directory for the function assigned to this utility,
 * read it and return a Map of properties.
 * The format is assumed to be the Java compatible key/value pair format.
 * The file must be readable, or an exception is thrown.
 * If the relative path leads to a file that is not a child
 * d of the proper configuration directory, an exception is thrown.
 * The full path structure is:
 * ${CUSTOM_FUNCTION_CONFIG_DIR}/namespaces/${namespace}/${relPath}
 * @param relPath The path to the file, relative to the namespace configuration
 * directory for the function assigned to this utility.
 * @param forceExist If true, and the file does not exist, an exception if thrown.
 * If false, and the file does not exist, an empty Map is returned.
 * @return A map of properties read from the file.
 * An empty map if forceExist was false and the file does not exist.
 */
Map<String, String> readNamespacePropertiesFile(String relPath, boolean forceExist)

JavaScript example:
var myProperties =
    Utils.config.readNamespacePropertiesFile("test.properties", false);
var myValue = myProperties.get("my_key");

Utils.config.readNamespacePropertiesFile(String relPath, boolean forceExist,
String charset)
/**
 * Given the path to a file relative to the namespace configuration
 * directory for the function assigned to this utility,
 * read it using the given character set and return a Map of properties.
 * The format is assumed to be the Java compatible key/value pair format.
 * The file must be readable, or an exception is thrown.
 * If the relative path leads to a file that is not a child
 * of the proper configuration directory, an exception is thrown.
 * The full path structure is:
 * ${CUSTOM_FUNCTION_CONFIG_DIR}/namespaces/${namespace}/${relPath}
 * @param relPath The path to the file, relative to the namespace configuration
 * directory for the function assigned to this utility.
 * @param forceExist If true, and the file does not exist, an exception if thrown.
 * If false, and the file does not exist, an empty Map is returned.
 * @param charset The character set (encoding) to use when reading the file.
 * If null, the JVM default is used, usually UTF-8.
Map of properties read from the file. An empty map if forceExist was false and the file does not exist.

```java
Map<String, String> readNamespacePropertiesFile(String relPath, boolean forceExist, String charset)
```

JavaScript example:
```javascript
var myProperties =
    Utils.config.readNamespacePropertiesFile
    ("test.properties", false, "UTF-8");
var myValue = myProperties.get("my_key");
```

**Utils.http.urlEncode(String value)**

URL encodes the given value

```java
String urlEncode(String value)
```

JavaScript example:
```javascript
var myEncodedPathParam
    = Utils.http.urlEncode("my Path parameter");
```

**Utils.http.invokeHTTP(String method, String uriStr, Integer expectedResponseCode, Map<String, String> headers, Map<String, Object> parameters, String body)**

This method is purposely monolithic. It will be deprecated when we provide a proper (and reliable) HttpClient implementation directly

Perform an HTTP request with the given parameters and return the response body as a String.

IMPORTANT: All request content (parameters, and body) is assumed to be UTF-8.

@param method The HTTP method to use:
[GET, PUT, POST, DELETE, OPTIONS, TRACE, HEAD, PATCH].
If null or empty, GET is used.
If not recognized, RuntimeException is thrown.

@param uri The address, which is not already escaped.
Query parameters should only be provided as part of this address if the parameters parameter is not specified.

@param expectedResponseCode The response code you expect,
or null if you don't care.
Throws RuntimeException if response code does not match.

@param headers A map of headers. Note that they are case and whitespace insensitive.

@param parameters A map of parameters.
If the request method does not support a request body,
they will be included in the URI.
If the request method does support a request body,
they will be translated to URLFormEncoded, and use the application/x-www-form-urlencoded content type; the payload will also be ignored.

@param body The raw request body as a String.
Will only be used if the request supports a request body.
A Content-Type header should always be provided,
* but Content-Length is automatically calculated.
* IMPORTANT: This is not compatible with extremely large data sets.
* No streaming implementation is provided either.
* @return The response body as a string.
* The character encoding used to construct it will be that which was
* specified by the response Content-Type header.
* If none is available, it will default to UTF-8
*/
String invokeHTTP(String method, String uriStr, Integer expectedResponseCode,
  Map<String, String> headers, Map<String, Object> parameters, String body)

JavaScript example:
var responseBody = Utils.http.invokeHTTP
  ("GET", "https://myServer/api/my_endpoint", 200, {"Token":"MyToken"}, {}, null);

Utils.concurrency.createConcurrentList()
/**
* @return Returns a new thread safe list implemented by
* java.util.concurrent.CopyOnWriteArrayList
*/
java.util.concurrent.CopyOnWriteArrayList createConcurrentList()

JavaScript example:
var myList = Utils.concurrency.createConcurrentList();
myList.add("something");
myList.add(25);

Utils.log.info()
/**
* Log the given message through QRadar using a log level of INFO.
* @param msg The message to log.
* Will be prefixed with relevant state and script information.
*/
void info(String msg)

JavaScript example:
Utils.log.info("My message");

Utils.log.warn()
/**
* Log the given message through QRadar using a log level of WARN.
* @param msg The message to log.
* Will be prefixed with relevant state and script information.
*/
void warn(String msg)

JavaScript example:
Utils.log.warn("My message");

Utils.log.error()
/**
* Log the given message through QRadar using a log level of ERROR.
* @param msg The message to log.
* Will be prefixed with relevant state and script information.
*/
void error(String msg)
JavaScript example:
Utils.log.error("My message");

**Utils.log.debug()**

```javascript
/**
 * Log the given message through QRadar using a log level of DEBUG.
 * Typically this log will be ignored unless enabled
 * through QRadar logging configuration.
 * @param msg The message to log.
 * Will be prefixed with relevant state and script information.
 */
void debug(String msg)
```

JavaScript example:
Utils.log.debug("My message");

**Note:**

When you use JavaScript, you also have access to standard ECMA utilities such as "JSON". For more information, see the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse).
31 Resources

Use various resources to help you to build apps with the IBM QRadar GUI Application Framework

**QRadar documentation**

- QRadar [SIEM](http://www-01.ibm.com/support/docview.wss?uid=swg27048741) documents in PDF format.
- QRadar [Vulnerability Manager](http://www-01.ibm.com/support/docview.wss?uid=swg27048729) documents in PDF format.
- QRadar [Risk Manager](http://www-01.ibm.com/support/docview.wss?uid=swg27048730) documents in PDF format.
- QRadar [Incident Forensics and Packet Capture](http://www-01.ibm.com/support/docview.wss?uid=swg2704873) documents in PDF format.

**QRadar API resources**

- QRadar API video tutorial: [Learn to use the QRadar API in six minutes](https://www.youtube.com/watch?v=iMom3BRGzXk).
- QRadar API samples on [GitHub](https://github.com/ibm-security-intelligence/api-samples).

**Flask API**

- [http://www.flaskapi.org/](http://www.flaskapi.org/)
- [Flask Tutorials](https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world)

**Jinja2 templates**

- [http://jinja.pocoo.org/docs/dev/](http://jinja.pocoo.org/docs/dev/)
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM’s suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

### Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

### Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

#### Applicability

These terms and conditions are in addition to any terms of use for the IBM website.
Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED “AS-IS” AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user’s session id for purposes of session management and authentication. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.
