VANTIQ Back-To-Work Accelerator Developer Guide

Last Revision: May 14, 2020

Contents

About

VANTIQ Accelerators are working applications that are designed to be imported into the VANTIQ platform as a new project and then customized by users who are looking to create applications with a similar purpose. The benefit of using an accelerator is starting off with an application that has a completed code base to bootstrap a new project being undertaken. Although accelerators are complete working applications, they are not intended to be used *as-is*, rather they are intended to speed up development of a new application by providing a completed and working set of base components and functionality to build upon and customize.

How to Use This Document

This document is intended for VANTIQ developers who want to customize the **Back-To-Work Accelerator** to create new functionality or modify existing features. A pre-requisite of using this document is an understanding of the VANTIQ platform. Recommended skills for VANTIQ developers are Javascript and SQL programming knowledge, but anyone with a background in software programming will be able to use this document.

It is not required but it is recommended to first read *the Back-To-Work Accelerator User Guide* to provide context to how the internal VANTIQ components are implemented and displayed from the end-user point of view.

The full documentation for the VANTIQ platform is available here:
https://dev.vantiq.com/docs/system/ide/index.html

For any readers who are not yet familiar with VANTIQ development, it is recommended that you first run through the Tutorials and then return to this guide to customize the **Back-To-Work Accelerator**. The Tutorials can be found at:
https://dev.vantiq.com/docs/system/tutorials/tutorial/index.html

This document will cover the following major topics:

- Connecting external data sources to your VANTIQ namespace.
- Creating new Client Builder pages to display reports, sensor readings and other visualizations.
- All the Procedures used by the Client Builder that can also be used as API's for external web or mobile clients.
- Setting up new Apps to process and analyze new sensor types.
- Creating your own Collaborations that include interactive user facing workflows.
- Adding plugins for specific third-party hardware vendors.
- A discussion on how to incorporate the Catalog system and why/when you would use it.

Code examples will be provided where relevant and additional information will be available on the accelerators GitHub page.

https://github.com/Vantiq/vantiq-blueprints

Back-To-Work Accelerator Features

The **Back-To-Work Accelerator** allows the user to model an organization's workplace by creating a digital twin of that workplace. A digital twin is a computerized representation of a real-life physical object, in this case, an internal space within a building and the assets contained within those spaces. Digital twins are often connected to sensors, cameras and other data sources to provide real-time displays of the status a particular object or space by capturing streams of data from these sources, analyzing the incoming data and monitoring for anomalies.

The value of the VANTIQ **Back-to-Work Accelerator** is to allow developers to quickly create their own unique models of their workplaces without spending a lot of time designing the user facing elements and visualizations. A complete set of user interfaces is provided along with the ability to create and define custom data schema's for sensors and devices.

Many of the assets in the accelerator such as *Types*, *Procedures*, and *Client Builder* pages will work right away with any data sources without requiring any additional development work. A user will need to define the buildings, rooms, assets and sensors through the Back-To-Work client.

The *Apps* and *Collaborations* will need to be modified to support your data and your user facing workflows.

Data Types

Types in the **Back-To-Work Accelerator** are organized into a hierarchy of data structures. These are arranged from the user perspective in a top-down order as shown in the image, but the Types have schema definitions that work in reverse to start at the bottom and work your way up. The child records will reference the parent record by its unique id property. The default configuration stores references in the following structure.

Sensors → Assets
Assets → Floors
Spaces → Floors
Floors → Buildings
Buildings → Organization

There is a smaller secondary hierarchy for users and groups represented with a one-to-many relationship between Personnel and Roles.

Personnel → Roles

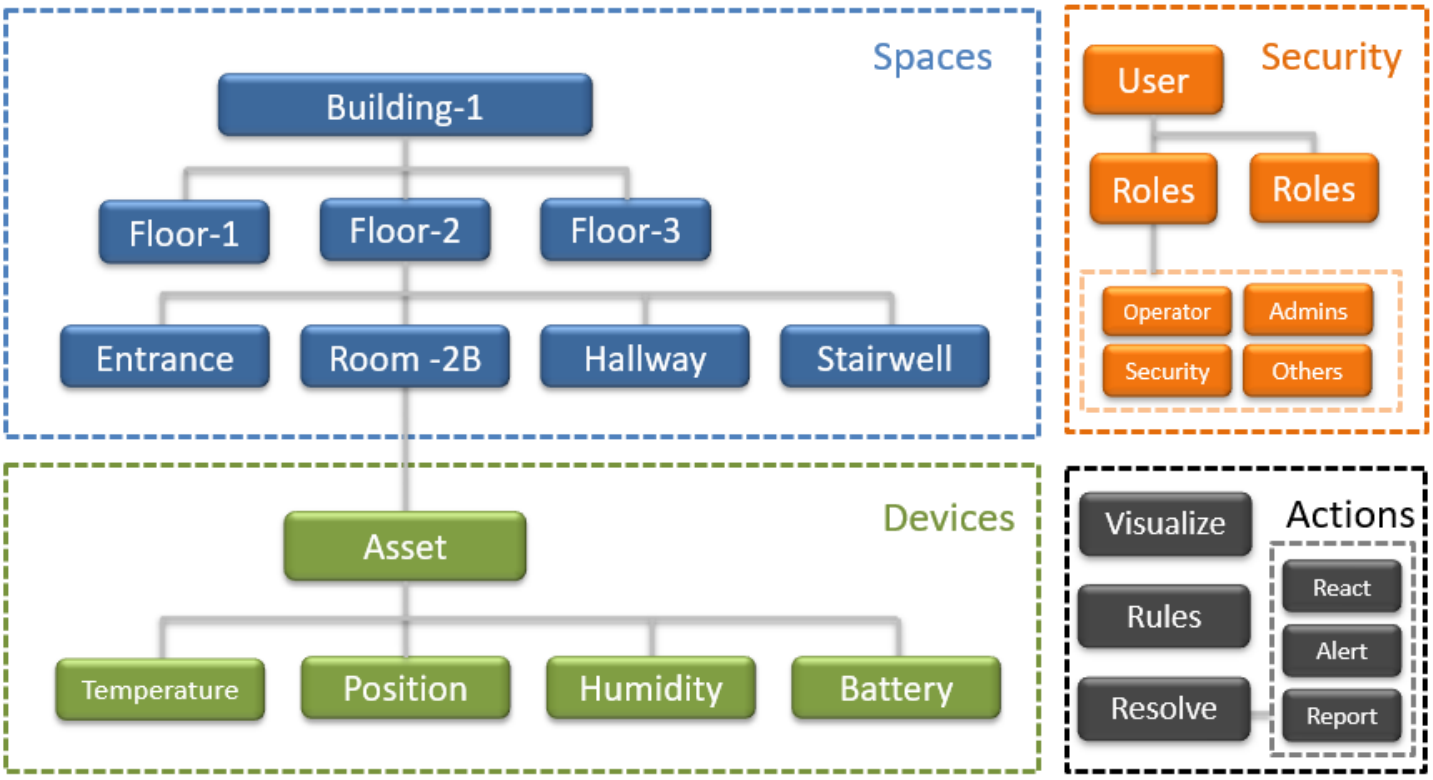


Figure 1 - Back-To-Work data structure hierarchy

An incoming event, such as a sensor reading, provides the initial context for data lookups. The sensor event will have a sensor id. From the sensor reading we can work our way up with a series of queries until reaching the **Organization**.

Here is an example query starting with a simple reading from a temperature sensor.

Figure 2 - Incoming Sensor Reading

```
{
        temp: 99,
        sensorid: “t1001”,
        time: 1588264119
}
```

Using the incoming sensorid as the starting point here is how to query from **Sensor** to **Organization**:

```
//Query the sensor record
SELECT ONE * from sensors AS sensor WHERE id == sensorid

//Get the Asset for that sensor
SELECT ONE * from assets AS asset WHERE id == sensor.assetid

//Get the spaces for that asset
SELECT ONE * from spaces AS space WHERE floorid == asset.floorid

//Get the floors for the asset
SELECT ONE * from floors AS floor WHERE floorid == asset.floorid

//Get the building for the floor
SELECT ONE * from buildings AS building WHERE id == floor.buildingid

//Get the organization for the building
SELECT ONE * from organization AS org WHERE id == building. organizationid
```

Asset to Space Assignment

The relationship between **assets**, **spaces** and **floors** is different because an **asset** is not assumed to be in a fixed location the same way that a **space** is. A **sensor** may move to a different room or floor but your kitchen isn't likely to change its location. An **asset** may be in motion all the time and it may move between floors or even between different buildings. In order to identify the correct space for the asset, the location is always calculated and the closest space is selected based upon distance.

In the above select statements, notice that both assets and floors refer to the floorid instead of an asset referencing a space directly. This is due to how the space is calculated based on the location of the asset and with each sensor reading, the closest space will be identified. For more details on location tracking, please refer to the code example in the Appendix on the procedure **app.updateSensorAndAssetLocation**, which shows how to update an asset's current location from the latest sensor reading.

Connect External Data Sources

One of the first and most important activities to perform when setting up the **Back-To-Work Accelerator** is getting connected to external data sources. External sources may be sensors, webhooks, cameras or extension sources. Extension sources will cover all the external connectors not built directly into the VANTIQ platform and includes examples like JDBC connections, OPC-UA, Object Recognition (streaming video) and others. There are multiple ways to connect sensors and external devices to the VANTIQ platform and this document will not cover each one in detail. Instead please refer to *the Enterprise Connectors Reference Guide*.

Incoming sensors will each have their own formats and schemas. The sensor's *Type* will store the sensor data in an unstructured object called data. This allows any sensor value to be stored regardless of how data is structured. The sensor's *Type* will only store the most recent reading. If there is a need to store the full history of all the sensor readings it is necessary to create a new *Type* to store the timeseries of readings.

Note: If the timeseries data is expected to be very large, it is recommended that the Type be setup to store the sensor history using a recommended format for timeseries data (boxed into chunks based on second/minute/hour/day) or that an external database (S3, InfluxDB, ect) is used to store the full history of the sensor.

Here is an overview of some of the recommended ways to connect data streams to the VANTIQ platform. Not all devices can be configured to connect directly to VANTIQ and in those cases using Webhooks or even pulling the data via REST calls might be necessary.

HTTP/REST/WebSockets

The easiest method to send data into VANTIQ is to have the device, be it a sensor or gateway, publish to a VANTIQ topic over an HTTP connection. To do this, a user will need a URL endpoint and an access token. Nothing is required on the developer side other than generating an access token. No data definitions need to be known or configured in advance as topic names can be used on demand and do not require any setup in advance.

It is also possible to post directly to a Type or even a Procedure, but these will require knowing the sensor structure ahead of time. It is much quicker and easier to use the pub/sub system to receive data streams which then provides the opportunity to process, enrich, filter, or transform the sensor reading before it is stored.

REST

REST calls are very common and will POST the sensor readings to a URI endpoint with an authorization and a JSON formatted body that contains the data payload. Authentication can be provided directly in the URI or through an Authorization header.

Token in URI:
https://dev.vantiq.com/api/v1/resources/topics//MyDataStream?token=<YOUR-ACCESS-TOKEN>

Token in Header:
https://dev.vantiq.com/api/v1/resources/topics//MyDataStream
Authorization: Bearer <YOUR-ACCESS-TOKEN>
Content-Type: application/json

The body of the POST should be in JSON format.

WSS

VANTIQ provides a secure WebSocket interface that can be used to stream data into the VANTIQ platform. WebSockets differ from REST in that they open a connection to the VANTIQ platform that remains active while streaming data while REST opens a new connection to the VANTIQ platform with every reading. This allows data transfer over WebSockets to be very fast and these should be used in any situations where high speed and high volume are expected.

To use WSS connections, you will need an authentication token. Publishing to a topic will require resource parameters that are passed into the WebSocket when it is initially opened that includes the initial authentication resources:

```
{ "op": "validate", "resourceName": "system.credentials", "object": "<YOUR-API-KEY>"};

And the next message sent would include the publishing resources.

{"op": "publish","resourceName": "topics","resourceId": "/topics/<LOCAL-TOPIC-NAME>",
"object":{<DATA-TO-SEND>};
```

Please refer to *the VANTIQ API Reference Guide* for detailed information.

MQTT/AMQP

MQTT is an increasingly popular protocol used by IoT devices to stream information to an end point. AMQP is a similar protocol but is not as common as MQTT. Both are supported in the VANTIQ platform and require a broker that sits between the VANTIQ platform and the device. The VANTIQ platform only provides the client-side subscriber libraries to ingest MQTT/AMQP data streams.

A **Source** has to be added to provide the VANTIQ platform with the details of the brokers location and the topic to which to subscribe. An MQTT connection for example might look like this:

tcp://public.vantiq.com:1883/cmp/gateway

Username and password credentials can be supplied in the source configuration.

Webhooks or Data Pulls

Webhooks and data pulls are less ideal for use with the VANTIQ platform. In these cases, the sensors are streaming to another location first, perhaps to a cloud platform like AWS, then the forwarded to the VANTIQ platform automatically in the case of webhooks or on demand in the case of a data pull.

Webhooks can be configured the same way the REST HTTP/WSS connections are made. Refer above to REST section for details needed to setup your webhooks.

A data pull usually involves creating a **Rule** that is periodically triggered by a **Scheduled Event**. The **Rule** will invoke a **Source**, most commonly this is a REST call to an external system, with some query properties indicating the data to retrieve.

A data pull may require some effort be put into a **Rule** before it will run properly. For example, if you need to query the latest data by date/time stamp, it will be necessary to store the date/time from the last pull attempt. It may be necessary to perform some de-duplication checks on records to avoid importing duplicates.

Remember to setup a **Scheduled Event** that will publish to a **Topic** that triggers the **Rule** to run and perform the data pull. Use a periodic timer that uses an interval that is in line with the expectations and requirements of the end user.

Extension Sources

VANTIQ Extension Sources use WebSocket connections to stream data from external data sources into VANTIQ. A Source is setup that contains all the configuration details for the connection so that when the Extension Source is running the WebSocket will publish the stream of events to that Source. Extension Sources are based on the VANTIQ Java SDK and examples can be found here:

https://github.com/Vantiq/vantiq-extension-sources

Viewing Incoming Data

While not required it is a good idea to setup a Subscription to view the incoming data before its applied to the App Builder. This way you can verify the incoming data is arriving, connections are working and authentication tokens have been correctly generated.

Subscriptions can be setup for Sources and Topics so that any of the connection methods described can be used, including webhooks and data pulls.

Once data is arriving and is visible form the Subscription it will be possible to view the data structures for the incoming sensor data which is used in the sensor configuration and use the Create Data Type button to setup a timeseries database to store the history of that sensors readings.

App Builder

Once data is arriving on the platform a user can log into the run time client and access the **Back-To-Work** client to configure buildings, floors, spaces, assets and sensors.

Data will not appear in the **Back-To-Work** client and rules will not yet be triggered until a VANTIQ App is created that will process the incoming data sources, update the sensors type and trigger collaborations based on the sensors rule configurations.

Here is an example of an App that is monitoring a sensor that indicates volume levels in a hand sanitizer station.

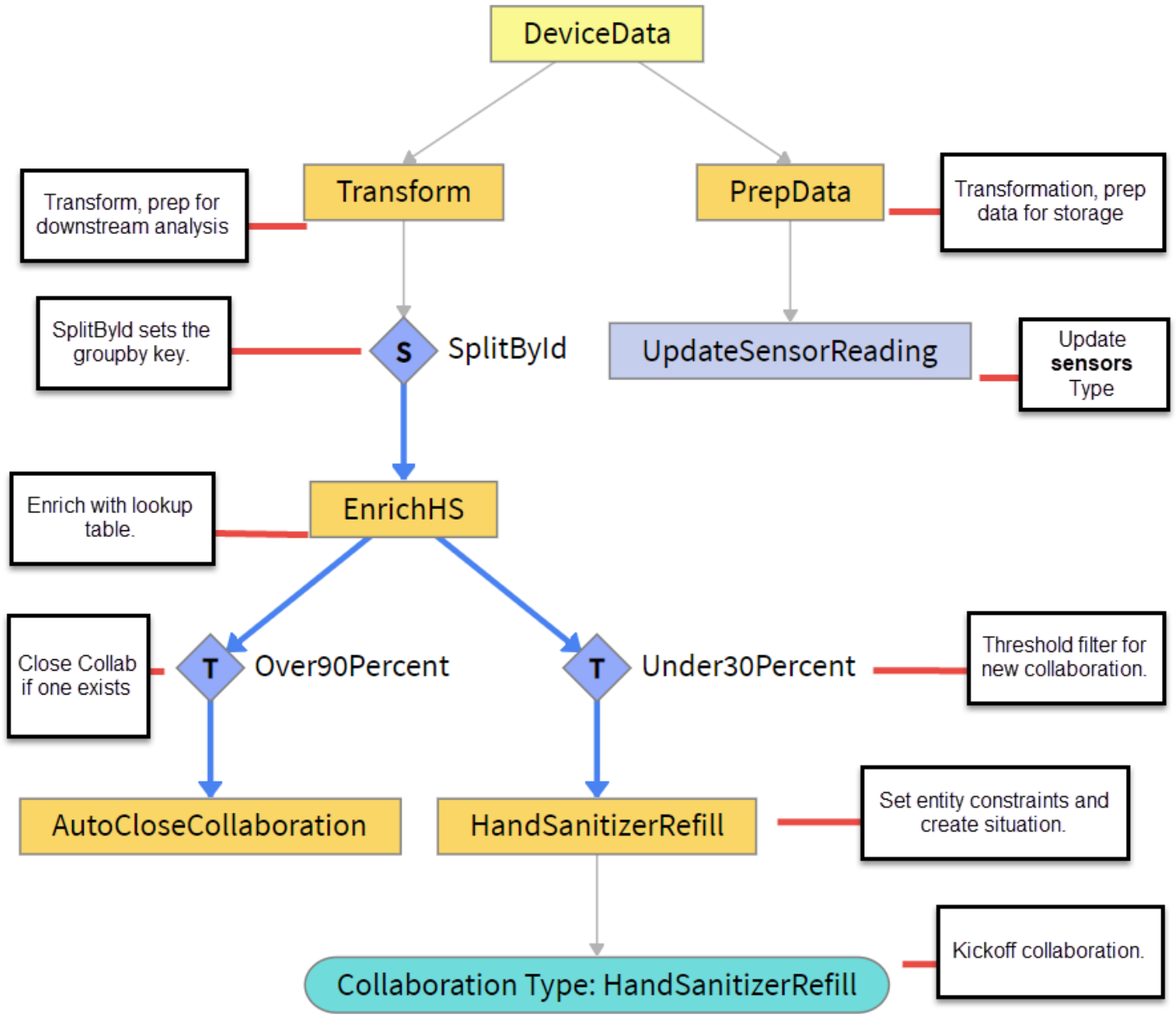


Figure 3 - Example App That Monitors a Volume Sensor in a Hand Sanitizer Station

In this diagram, the initiate step called **DeviceData** is configured for the Source or Topic where the sensors are publishing data. If there is a data pull being performed, then the results of the pull can be published to an internal topic or inserted into a **Type** and that would be used in the DeviceData configuration.

The *SaveToType* activity pattern is required for each sensor or device that has unique requirements. A thermal camera that performs temperature scans would require a second App that processes the thermal camera data stream with separate *dwell* or *threshold* filters to identify situations.

Rules

When a user sets up a sensor in the **Back-To-Work** client, a **rule** can be optionally defined. The rule values can then be applied to the filter activity patterns like *Threshold* or *Dwell*. If the user goes back and changes the values in the rule, those values would be applied automatically in the App when the next reading arrives.

Here is an example of a rule definition a user can input when setting up a sensor.

```
{
  "display": {
    "page": "GaugeDisplay",
    "config": {
      "minimum": 0,
      "maximum": 100,
      "lowZones": "0:30",
      "lowColor": "#ff0000",
      "mediumZones": "30:50",
      "mediumColor": "#ffff00",
      "highZones": "50:100",
      "highColor": "#00cc00",
      "title": "% Full"
    }
  },
  "rules": {
    "low": {
      "lte": 29
    },
    "full": {
      "gte": 90
    }
  }
}
```

The user can define the name of the rule such as “low” or “full” which indicates the volume levels in the hand sanitizer. Then the user can provide the condition that determines the low and full volume levels, in this example the operator *lte* (less than equal) and *gte* (greater than equal) to a numeric value. These values represent the volume percent which is provided by the sensor itself with low being 29 or lower and full being 90 or higher.

When developing the App, you can use *appCondition.evaluteConfig* procedure to access the rule values. Here is an example view of the hand sanitizer app’s **Under30Percent** *Threshold* filter.

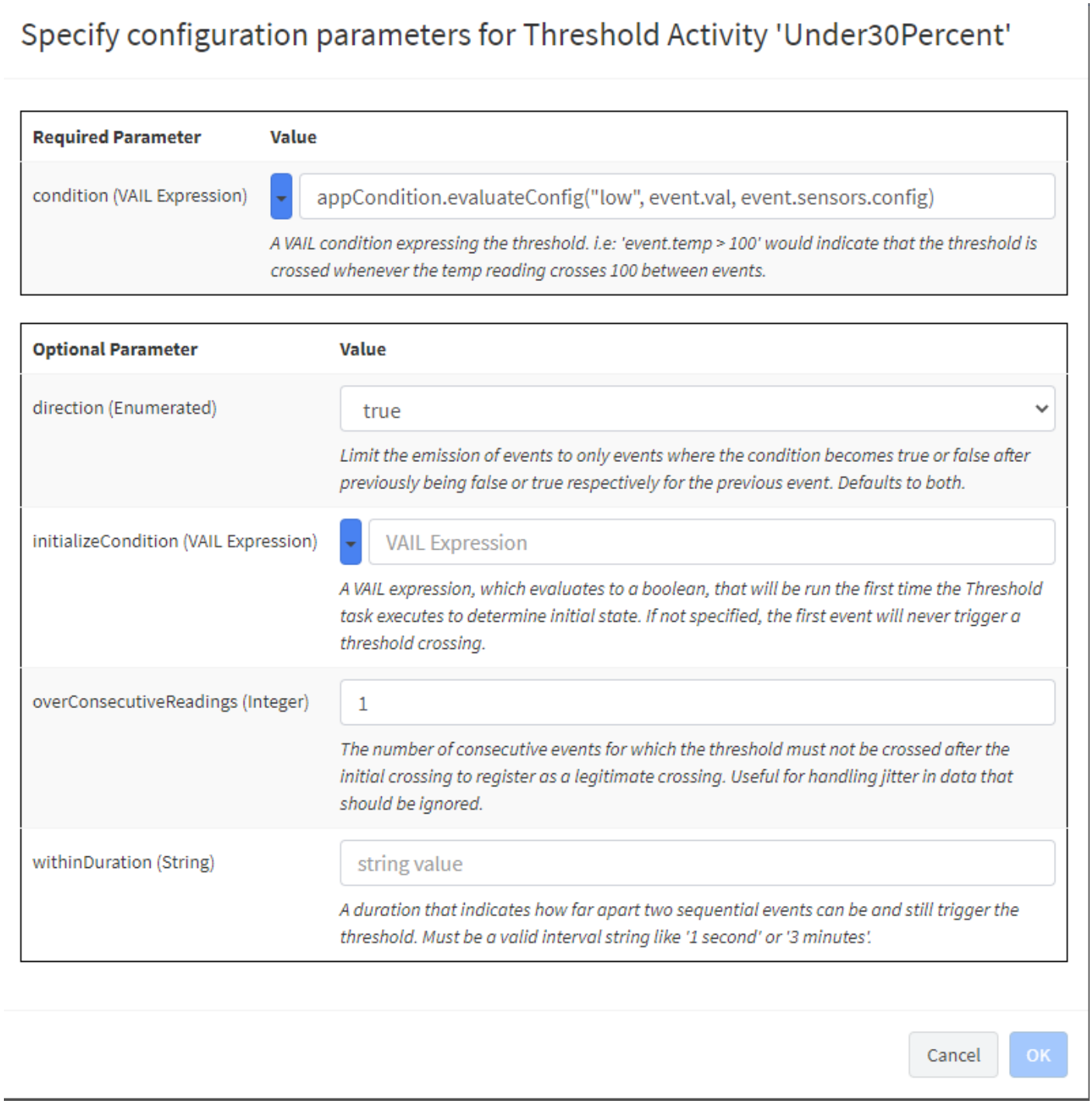


Figure 4 - Setting Threshold Parameters

Condition: **appCondition.evaluateConfig("low", event.val, event.sensors.config)**

The condition looks specifically for the "low" setting in the sensor config. The **Over90Percent** would then look for the full value as shown in this example:
appCondition.evaluateConfig("full",event.val, event.sensors.config).

As a VANTIQ developer, it will be necessary for you to provide the rule details to the end user.

For example, if a new temperature sensor is added and the App will monitor high and low values then develop a sensor configuration that the user can plug into the sensor when setting it up in the **Back-To-Work** client.

```
{
  "display": {
    "page": "",
    "config": {}
  },
  "rules": {
    "low": {
      "lt": 5
    },
    "high": {
      "gt": 190
    }
  }
}
```

Skip down to the section on Visualization Plugins for instructions on setting the display values.

When adding a *Dwell* or a *Threshold* filter in the App builder, use the rule's key name when running the **appCondition** procedure to pull the associated value.

> **High Value: appCondition.evaluateConfig("high",event.val, event.sensors.config)**
> **Low Value: appCondition.evaluateConfig("low",event.val, event.sensors.config)**

The event.val input is the actual value provided by the sensor. Change this value to reflect your actual event object or use a transformation to change it to match.

The **event.sensors.config** is a value provided by the *enrich* or *cached enrich* activity step. The enrichment uses the sensorid to query the sensors **Type**. This contains the config settings for the sensor where the rules are stored.

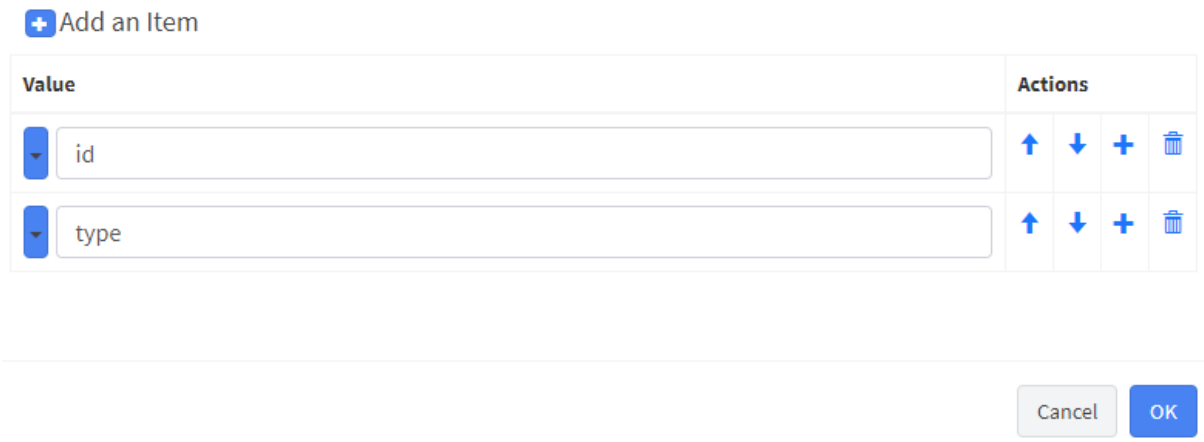


Figure 5 - Enrichment configuration of the sensors table will be the same for all apps. Use transformations to get id and type placed properly.

The **appCondition.evaluateConfig** procedure will return *true* or *false* based on the evaluation of the sensor value and the rule condition. If *true*, the event moves downstream and executes the next step in the App builder.

Procedures

The **procedures** written in the **Back-To-Work Accelerator** are meant to be very reusable from both inside th VANITQ platform and as well as outside. All **procedures** in VANTIQ double as API calls that are accessible outside of VANTIQ. These procedures could be used to create custom front ends, external dashboards and reports, or integrations into other third-party applications or platforms.

Procedures use a common naming convention. The procedure name is prefixed based on where in VANTIQ it is being used. App Builder procedures will begin with *"app.<procedurename>"* and Client Builder procedures begin with *"cb.<procedurename>"*. Procedures that begin with *"dashboard.<procedurename>"* are used to populate the Issues table on the Home/Start page of the **Back-To-Work Accelerator**. Collaborations will begin with *"collab.<procedurename>* and there maybe additional prefixes or potentially no prefix at all for some procedures, which will often indicate it is used commonly across the platform.

The Client Builder procedures *cb.<procedurename>* are the only procedures provided as part of the accelerator. However, examples of *app*, *dashboard* and *collab* procedures will be provided.

Procedure names attempt to be as self-descriptive as possible, for example the **cb.deleteAsset** is used in the Client Builder to delete an asset. The first letter in the procedure name is always lower case followed by upper case for any remaining words.

Client Builder Procedures

cb.adjustTheme	Used by the Organization → Theme page to make changes to the Back-To-Work theme. Adjusts colors and logo.
cb.findSpaceForAsset	Finds the closest space for an asset. This calculation uses the assets.fplocation value, which is updated with each sensor reading that contains a location, and

	compares it to all the spaces on the floor to determine the closest one. **An RTLS system using indoor tracking can also provide the closest space as part of the sensor reading. When that is the case this procedure can also be used in simulations.**
cb.getAssetTypes	Queries all the records in assettypes.
cb.getAssetById	Queries one specific asset by id.
cb.getAssetForSensor	Queries a specific sensor by id and uses the result to query a specific asset by the sensors.assetid field.
cb.getAssetTypes	Queries all the assettypes, sorts results alphabetically by name.
cb.getAssets	Queries all assets.
cb.getAssetsForFloor	Uses a Floor id and a Building id to return a list of every asset associated with a floor.
cb.getBuildingById	Queries a single building by its id.
cb.getBuildingForSpace-BySpaceId	Nested query, first fetches one specific space by id, then one specific floor by spaces.floorid and then one specific building using floors.buildingid.
cb.getBuildings	Queries for all buildings by organization id.
cb.getCollabContext	Queries a specific collaboration id and returns an array of entities associated with the collaboration. Entities provide the context for the collaboration such as space, floor, building, asset id's.
cb.getDashboardData	Provides the number counts on the Home page for buildings, floors, assets, sensors, ect.
cb.getFloorById	Queries for a specific floor by its id
cb.getFloorForSpaceBy-SpaceId	Nested query returns one specific Floor by querying the space first by its id.
cb.getFloorPlansBy-BuildingId	Queries a specific floorplan using a building id.
cb.getFloorplan-ByFloorId	Nested query that uses a floor id to return a specific floor plan.
cb.getFloorsForBuilding	Returns an array of all floors for a specific building id.
cb.getHierarchyForAsset-ByFloorId	Queries a specific building using a floor id.
cb.getHierarchyForAssets-BySpaceId	Nested query that starts with the spaceid and then returns the space, floor and building details in a single object.
cb.getOpenIssues	Procedure used by the Home/Start page to populate the Issues table. This procedure will query the list of procedures for all that begin with the word **dashboard**. The dashboard procedures must be added before any issues can be returned. More details in the Dashboard and Issues section.
cb.getOrganization	Returns the organization name. Assumed just one organization exists.
cb.getPersonnel	Returns a list of all personnel, used to populate the View Users table, sorted by lastname.
cb.getRoles	Returns a list of all Roles sorted by name, used in the View Roles table.
cb.getSensorId	Queries for a specific sensor by its id.
cb.getSensorByIdAndType	Queries for one specific sensor using two conditions id and type.

cb.getSensirConfigById	Queries for exactly one sensorconfig by id. Will generate an error if no result is found.
cb.getSensorConfigs	If a sensortype name is provided will return all sensors of that type, if no type name is provided will return all sensor types sorted by the type name.
cb.getSensors	Returns a list of all sensors sorted by type.
cb.getSensorTypes	Returns a list of all sensortypes sorted by name
cb.getSensorForAsset	Returns a list of all sensors associated with a specific asset, sorted by type.
cb.getSpaceById	Queries a specific space by id.
cb.getSpacesForFloor	Queries a list of spaces by floor id.
cb.getTheme	Parses the Back-To-Work client to determine the current Theme settings, color selections and logo.

App Builder Procedures

These are procedures used in the App Builder to provide customized activities. Since no Apps exist in the default accelerator, no app builder procedures will exist either.

The following example applies to the Hand Sanitizer App which takes as input the event object, looks for open collaborations associated with the device, and closes the collaboration if it exists.

```
PROCEDURE app.closeCollabForHandSanitizer(event Object)

SELECT ONE FROM sensors WHERE id == event.id
SELECT ONE FROM system.collaborations as collab
        WHERE name == "HandSanitizerRefill" AND
        status == "active" AND
        entities.sensor == "/sensors/" + sensors._id

if (collab){
   CollaborationUtils.closeCollaboration(collab.id)
}

return event
```

A procedure like this can be easily modified to support a number of use cases and is useful to automatically close a collaboration when sensor readings return to normal instead of having a user manually close the collaboration.

Dashboards and Issues

The *Home/Start* page in the Back-To-Work client will perform a lookup of open Collaborations using procedures prefixed with the *"dashboard.<procedurename>"* naming convention.

Each sensor type can be used to initiate collaborations that all have different data values and intents. The dashboard procedure will allow each type to be queried separately and formatted to specifically show results in the *Issues* table that are relevant.

The example query in the appendix will return all open Hand Sanitizer collaborations and apply the contents to a template for display in the *Issues* table. Go To the Dashboard for Hand Sanitizer.

The template referenced in the Procedure, which is used to format the results into HTML tags used by the DataTable, was added as a text document under the path *"/templates/dashboardhandsanitizer"*.

```
<p><strong>Hand sanitizer station needs refill</strong></p>
<p><strong>Building</strong>: ${buildingname}</p>
<p><strong>Floor</strong>: ${floorname}<br></p>
<p><strong>Asset</strong></p>
<ul>
  <li>Name: ${assetname}</li>
  <li>Type: ${assettype}</li>
</ul>
<p><strong>Sensor</strong></p>
<ul>
  <li>Name ${sensorname}<br></li>
  <li>Type ${sensortype}</li>
</ul>
```

The dashboard procedure will perform a substitution of all the ${variablenames} using a system procedure called **generateResource**. The input values are the text to scan (the template file) and a JSON document containing the field names and the replacement values.

For example, to replace ${buildingname} with the name of a building, run the following in a procedure:

```
var text = "${buildingname}"
var sub = {buildingname: "Corporate Headquarters"}
var result = generateResource(text, sub)
```

Result will be "Corporate Headquarters" The dashboard example performs the same task, only the template is stored as a text file in **Documents** to make it easier to author.

Other Procedures

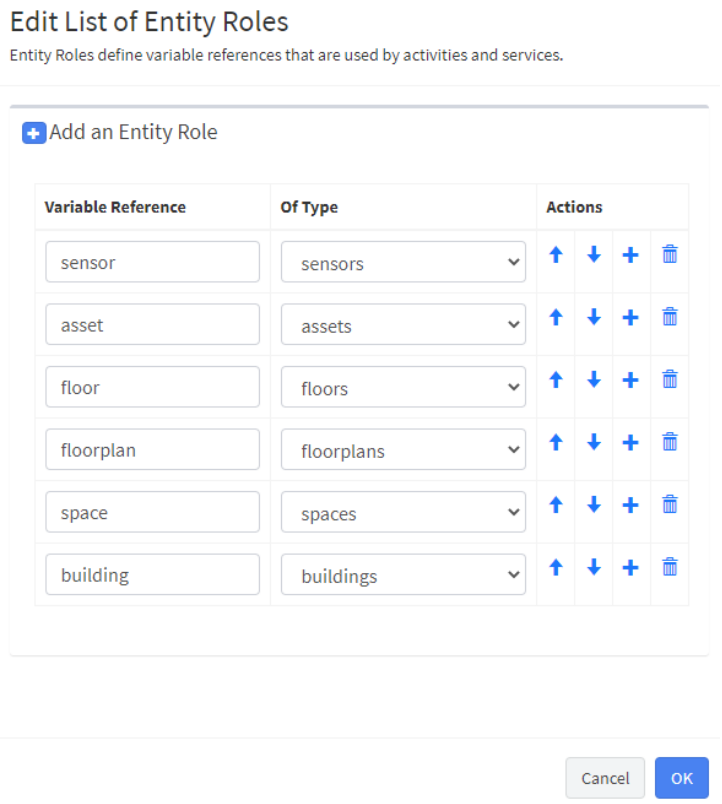
appCondition.evaluateConfig	This procedure is used in App builder filters like Threshold, Dwell, Filter to read the user defined rule conditions for that sensor.
generateResources	System procedure used to perform a find and replace on ${<name>} tags.
utility.removeSensorsWhereTypeEmpty	Deletes all sensors with a null type.

Collaborations

Collaborations can be built to utilize all the available data **Types** and use **Entity Roles** to provide references in the collaboration with values to the building, floor, space and asset. This data will not be included as part of the sensor reading that initiates the collaboration so the **entity roles** will need to be assigned the correct values.

One way to do this is to execute a procedure that returns the entity role values and then assign the value to the entity. Here is how:

1. Setup the list of Entity Roles that correspond to the data types in the Back-To-Work Accelerator.



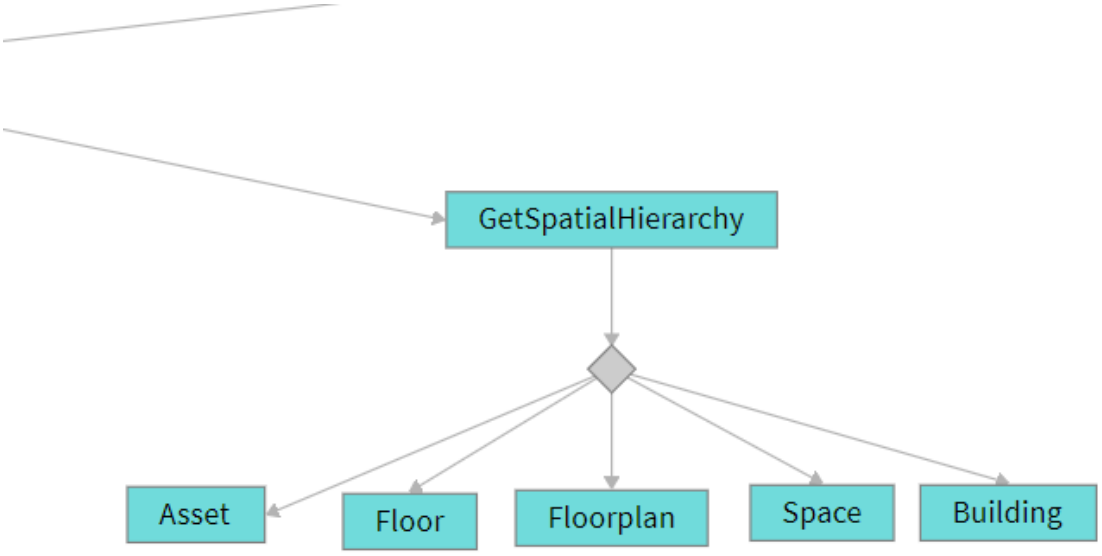
2. Execute a procedure that will use the id of the item that triggers the **collaboration**. This will likely be the sensors id value. The example procedure below will run a query against each data Type:

```
PROCEDURE collab.getHierarchyFromSensorId(id String, type String)
var obj ={}
SELECT ONE FROM sensors WHERE id == id AND type == type
obj.sensorREF = {_id:sensors._id}
SELECT ONE FROM assets WHERE id == sensors.assetid
obj.assetREF = {_id:assets._id}
SELECT ONE FROM floors WHERE id == assets.floorid
obj.floorsREF = {_id:floors._id}
SELECT ONE FROM floorplans WHERE id == floors.floorplan
obj.floorplanREF = {_id:floorplans._id}
var space = findSpaceForAsset(floors.id , assets.fplocation)
obj.spaceREF = {_id:space._id}
SELECT ONE FROM buildings WHERE id == floors.buildingid
obj.buildingsREF = {_id:buildings._id}

return obj
```

3. Tne last step is to create Assignment activities for each of the data **Types**. Use the output of the parent step, in the example shown its **GetSpatialHierarchy**, as the assignment value (expression) in the runtime params: **collaboration.results.GetSpatialHierarchy.result.spaceREF**

   Do this for each type substituting in the correct runtime value such as buildingREF or floorplanREF.



This will provide the collaboration the context it needs, for example, to know its location inside the building. This will be useful when creating collaborations that require the location of the entity. This **collaboration**

might used be to dispatch a field service worker to a broken item, a security guard to an intruder, or a member of the hospital staff to a wheelchair needed by a patient.

As sensor readings continue to arrive, the location of the asset my change. The collaboration's entities should be updated to reflect the latest readings so that a moving wheelchairs current location can always be correctly determined.

This will require running a procedure from the **App** itself. The **collaboration** will not update automatically from a new reading as the entity constraint is designed to prevent original situation details from changing. A new step will need to be added to the App Builder which can update the collaborations entities with the latest readings. Go to Update Collaboration with Latest Location for an example of how to code this.

Visualization Plugins

The **Back-To-Work Accelerator** is meant to be modified and provides the ability to simply and easily create new pages for the **Back-To-Work** client. These pages can be used to display sensor values or produce statistical and historical reports.

Refer back to the sensor type configuration where a display value can be provided.

```
{
  "display": {
    "page": "GaugeDisplay",
    "config": {
      "minimum": 0,
      "maximum": 100,
      "lowZones": "0:30",
      "lowColor": "#ff0000",
      "mediumZones": "30:50",
      "mediumColor": "#ffff00",
      "highZones": "50:100",
      "highColor": "#00cc00",
      "title": "% Full"
    }
  },
  "rules": {
    "low": {
      "lte": 29
    },
    "full": {
      "gte": 90
    }
  }
}
```

The display properties contain a pair of values called “page” and “config”. The page can be used to reference a new page that is added to the **Back-To-Work** client. The config settings are used to specify settings to apply to a widget on the display page.

The example provided uses a page called “GaugeDisplay”. When a user is viewing a sensor and clicks on the “View” button for that sensor to see the latest value, a pop-up page will load this “GaugeDisplay: page.

Figure 6 - Sensor Gauge

In this example the gauge widget is displayed in the popup page and the “config” settings are used to provide custom settings for the gauge widget. This allows a user to modify the *High*, *Med*, *Low* and *Title* values that can then be applied to other types of sensors that measure at different scales.

The sensor **Type** stores this value in the config property field so when the sensors list is loaded, the configuration for the sensor is included. When a user clicks on the **View** button for the sensor, if there is an associated page, it will load that page with the parameters objects for the clicked row. On the popup page you can then modify the widget using the values that are provided in the parameters. Here is sample code that is placed into the pages *On Start* code area using the gauge widget:

```
var gauge = client.getWidget("Gauge1");
gauge.maximum = parameters.config.display.config.maximum;
gauge.minimum = parameters.config.display.config.minimum;
gauge.lowColor = parameters.config.display.config.lowColor;
gauge.lowZones = parameters.config.display.config.lowZones;
gauge.mediumColor = parameters.config.display.config.mediumColor;
gauge.mediumZones = parameters.config.display.config.mediumZones;
gauge.highColor = parameters.config.display.config.highColor;
gauge.highZones = parameters.config.display.config.highZones;
gauge.title = parameters.config.display.config.title;
```

The gauge will be reconfigured based on the user defined configuration parameters. This approach can be applied to other widgets using the same approach.

Be sure to provide your **Back-To-Work** users with the list of fields that can be used in the config section and the page name.

Appendix

Changing Existing Schema's

It is not recommended that the pre-existing **Type** schema's are changed. Renaming a field in the sensors **Type** or another **Type** will likely break functionality in the Client. This will apply to data streams, data objects and the many of the *Client Builder* procedures that may be used.

New properties can be added to the **Type** but may require changes to different Insert or Update's if the expected list of fields used in a *SaveToType* activity do not match.

New **Types** can be easily created and will not impact existing clients, apps, or procedures. When possible, it is recommended to setup new **types** to store additional information which are referenced by the pre-existing ones. A sensorhistory **Type** could be added to store the history of the Hand Sanitizer sensor readings. The id value stored in the sensorhistory Type can match the id value in the sensor Type so that historical queries can be easily executed to populate a graph or chart.

Code Examples

Dashboard for Hand Sanitizer

The following procedure shows how to retrieve any open issues generated by rules associated with a hand sanitizer sensor.

```
PROCEDURE dashboard.getOpenHandSanitizerIssues()
var results = {}
var issueCount = 0
var issues = []
var data = []
SELECT FROM system.collaborations AS collab WHERE status == "active" AND name == "HandSanitizerRefill"{
   var issueObj={}
   if (has (collab.results, "Status") ){
      issueObj.state = collab.results.Status.assignedValue
   } else {
      issueObj.state = "Open"
            }
   issueObj.issuetype = collab.name
   var obj = {}
   var template = getDocument("templates/dashboardhandsanitizer")
            var replace = {}
   SELECT ONE FROM sensors AS sensor WHERE id == collab.results.Initiate.event.id
   if (sensor){
      obj.sensor = sensor
      obj.sensorname = sensor.id
      obj.sensortype = sensor.type
      issueObj.sensortype = sensor.type
      issueObj.d = sensor.id
      issueObj.sensorconfig = sensor.config
      SELECT ONE FROM assets AS asset WHERE id == sensor.assetid
      obj.asset = asset
      if (asset){
         issueObj.assettype = asset.type
         obj.assettype = asset.type
         obj.assetname = asset.name
         SELECT ONE FROM floors AS floor WHERE id == asset.floorid
         obj.floor = floor
         if (floor){
            issueObj.floorname = floor.name
            obj.floorname = floor.name
            SELECT ONE FROM buildings AS building WHERE id == floor.buildingid
            obj.building = building
            obj.buildingname = building.name
            issueObj.buildingname = building.name
            issueObj.geo = building.geo
         }
      }
   }
   issueObj.description = generateResource(template, obj)
   push(issues, issueObj)
   push(data, obj)
   issueCount++
}
results.issueCount = issueCount
results.issues = issues
results.data = data

return results.issues
```

Update Collaboration with Latest Location

```
PROCEDURE app.updateSensorAndAssetLocation(event Object)

var loc = {
            "type": "Point",
            "coordinates": [event.y, event.x]
}

var sensor = SELECT ONE FROM sensors WHERE id == event.id and type == "IndoorLocation"

if (sensor){
   var data = {}
   data.type = "GeoJSON"
   data.value = loc

   UPSERT sensors(id:event.id, type:sensor.type, data:data)

            // assets also have a floorid (floor) and fplocation (floorplanlocation)
            // we'll update these as well

            var asset = SELECT ONE FROM assets WHERE id == sensor.assetid
            if (asset){
      UPSERT assets(id:asset.id, floorid:event.floorid, fplocation:loc )
            }

}

SELECT FROM system.collaborations AS collab WHERE name == "PatientTransport" AND status == "active" AND
entities.tsensor == "/sensors/"+ sensor._id{
   event.collaborationId = collab.id
   PUBLISH event TO TOPIC "/indoortracking"
}

var history = {}
history.id = uuid()
history.sensorid = event.id
history.floorid = event.floorid
history.timestamp = now()
history.x = event.x
history.y = event.y

INSERT trackinghistory(history)

return event
```