# SmartSpace®

## HMIs

From version 3.8.1

Ubisense Limited, St Andrew's House, St Andrew's Road, Cambridge CB4 1DL, United Kingdom.

Telephone: +44 (0)1223 535170. Website: https://www.ubisense.com

Copyright © 2023, Ubisense Limited 2014 - 2023. All Rights Reserved. You may not reproduce this document in whole or in part without permission in writing from Ubisense at the following address:

Ubisense Limited
St Andrew's House
St Andrew's Road
Cambridge CB4 1DL
United Kingdom

Tel: +44 (0)1223 535170

WWW: [https://www.ubisense.com](https://www.ubisense.com)

All contents of this document are subject to change without notice and do not represent a commitment on the part of Ubisense. Reasonable effort is made to ensure the accuracy of the information contained in the document. However, due to on-going product improvements and revisions, Ubisense and its subsidiaries do not warrant the accuracy of this information and cannot accept responsibility for errors or omissions that may be contained in this document.

Information in this document is provided in connection with Ubisense products. No license, express or implied to any intellectual property rights is granted by this document.

Ubisense encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.

UBISENSE®, the Ubisense motif, SmartSpace® and AngleID® are registered trademarks of Ubisense Ltd. DIMENSION4™ and UB-Tag™ are trademarks of Ubisense Ltd.

Windows® is a registered trademark of Microsoft Corporation in the United States and/or other countries. The other names of actual companies and products mentioned herein are the trademarks of their respective owners.

# Contents

**Purpose of this Document** ... *2*
**Introduction to HMIs** ... *3*
Requirements ... *4*
**Installing the HMIs feature** ... *5*
**Configuring HMIs using the HMI Editor** ... *6*
Opening the HMI Editor ... *6*
The HMI Editor ... *7*
Creating a new HMI ... *8*
Saving an HMI ... *9*
Errors in the HMI ... *10*
Reloading an HMI ... *10*
HMI Settings ... *10*
Options ... *10*
Roles settings ... *11*
History settings ... *12*
Assets ... *13*
Publishing an HMI ... *16*
Setting an HMI as the Landing Page for SmartSpace Web ... *16*
Viewing an HMI Version Outside the Editor ... *16*
Exporting and Importing HMIs ... *17*
Exporting HMIs ... *17*
Importing HMIs ... *17*
**Loading a Built Web Application as an HMI** ... *18*
**External Development Workflow** ... *20*
Setting up an External Development Server ... *21*
Project Features for the External Development Workflow ... *21*
Setting an HMI to use the External Development Workflow ... *22*
Production Builds in the External Development Workflow ... *23*

Example of External Development Workflow with Vue CLI ... 24
Exporting/Importing HMIs with the Development Workflow ... 27
**The HMI API ... 29**
Data Binding for Web Searches ... 29
Declarative Web Searches ... 29
Complex Properties in Searches ... 31
Binding Web Search Results ... 31
Search Parameters in HMI Queries ... 32
Refreshing Searches ... 34
Data Binding for Reporting Queries ... 34
Declarative Queries ... 35
Binding Query Results ... 37
Refreshing Queries ... 37
Using the API from JavaScript ... 38
Callbacks ... 38
Using Bound Data in JavaScript ... 38
Setting Object Properties ... 39
Creating New Object Instances ... 40
Error Handling ... 40
Manual Vue Construction ... 41
Tag Management ... 42
Current User and Role ... 44
Getting the Logged-in User and their Roles ... 44
Built-in Components ... 44
Indicator component ... 44
Bar component ... 45
Gauge component ... 47
Map component ... 49
Report component ... 51
API Examples ... 55

Simple Text Example ... *55*
Various Progress Controls ... *56*
Report, Map and Details ... *58*
Styling Bars ... *62*
Progress Rings ... *64*
JavaScript Libraries ... *67*

# Purpose of this Document

This document is a guide to the configuration and use of Ubisense SmartSpace HMIs (Human Machine Interfaces), which is part of the *Visibility*. The intended audience includes users who are:

- Installing HMIs into a SmartSpace system
- Developing custom HMIs for a SmartSpace system
- Users of HMIs

# Introduction to HMIs

The HMIs feature allows custom web interfaces to be developed and deployed within the Ubisense SmartSpace website. The feature provides administrators with an editor interface that can be used to implement HMIs that are hosted by the SmartSpace system. The editor allows interfaces to be tested and developed, with immediate results displayed. Once an interface is working, it can be published to a set of SmartSpace roles.

A simple declarative binding API is provided to allow HMI content to be generated based on web searches defined within SmartSpace. HMIs can interact with and control the business objects and properties, using the same role-based authorization as other parts of the SmartSpace platform. This allows many HMIs to be designed using only attributed HTML. The API includes wrapped versions of the web map and web reports, so these can be embedded into HMIs as required, provided they have been licensed.

From version 3.5, queries from the Reporting component can be used to generate content using data binding in a similar way to web searches, again with role-based authorizations. This can provide a more flexible approach than using the built-in report component.

For more advanced uses, CSS and JavaScript can be added to the interfaces. External assets, such as images and script libraries, can also be hosted in the SmartSpace website for use in HMIs. Interfaces can then be exported and imported, along with any referenced assets.

From version 3.6, SmartSpace HMIs support an external development workflow as an alternative to the built-in HMI Editor allowing the use of more powerful editors and standard version control. Setting up and using an external development workflow is described in *External Development Workflow*.

From version 3.8.1, externally-developed Single Page Applications (SPAs) can be loaded into the HMI framework. They can then be accessed via the SmartSpace website with the same role-based authorizations as other SmartSpace HMIs. An SPA can be rendered in the SmartSpace HMI wrapper making use of the HMI API, or the SmartSpace website can simply serve the index.html of the SPA with no additional JavaScript. Implementing SPAs in SmartSpace is described in *Loading a Built Web Application as an HMI*.

**jQuery and jQuery UI with HMIs**

From the 3.7 SP3 and 3.8 releases of SmartSpace, the version of jQuery UI has been upgraded to 1.13.2.

From the 3.6 SP4 and 3.7 releases of SmartSpace, the versions of **jQuery** and **jQueryUI** have been upgraded to jQuery 3.5.1 and jQuery UI 1.12.1. This could affect sites with existing HMIs. As a result, Ubisense recommend HMIs should be tested to identify and fix deprecated or removed functionality, for example by using the **jquery-migrate** scripts available at: *https://github.com/jquery/jquery-migrate*.

**Browser compatibility**

SmartSpace Visibility features such as HMIs by default work with most recent browsers. However, it is up to the developer of individual web interfaces to ensure that these are compatible with their users' browsers.

## Requirements

The HMIs feature requires a license for Visibility version 3.4 or higher (version 3.5 or higher for data-binding with Reporting queries; version 3.6 or higher for the external development workflow).

If the HMIs are to include reports, then the Reporting component must also be licensed.

# Installing the HMIs feature

To install the HMIs feature:

1. Make sure that the SmartSpace platform includes a license for the correct version of Visibility.
2. Install the HMIs feature using Service Manager.
3. Upgrade the SmartSpace website to the same release version. (Run the **SmartSpaceWeb.msi** from the web folder of your distribution directory.)

For further information on all aspects of installation, see SmartSpace Installation on the Ubisense Documentation Portal or your SmartSpace Installation Guide.

# Configuring HMIs using the HMI Editor

There are three stages to building an HMI and making it available to users:

1. Creating the HMI in the HMI Editor, described in *Creating a new HMI*.
2. Adding it to roles, described in *Roles settings*.
3. Publishing a finished version, described in *Publishing an HMI*.

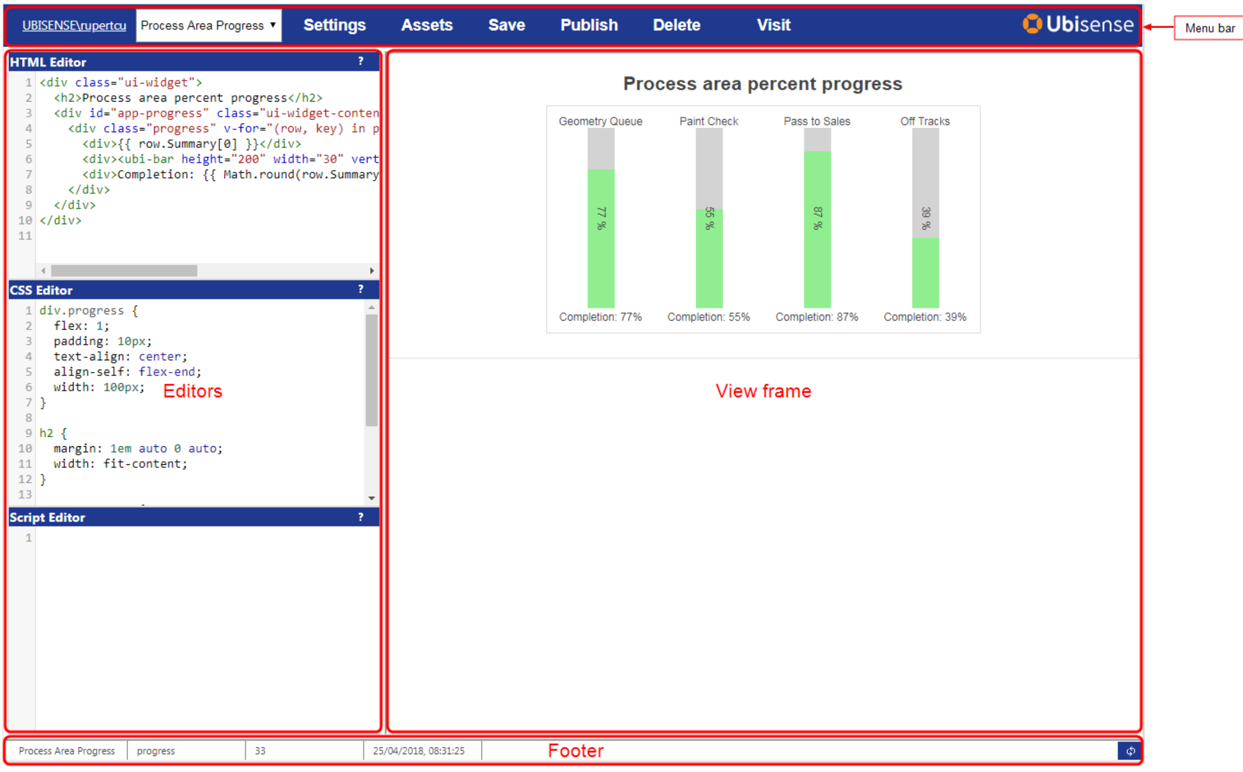
## Opening the HMI Editor

In order to be able to define HMIs, you must be logged in to the website as a user with role "Ubisense.SmartSpace.Administrator". In **SmartSpace Config** use the USERS / ROLES task to add a suitable user to this role. For further information on adding users to roles, see Users and roles on the Ubisense Documentation Portal.

Now open a web browser and point it to the SmartSpace website. You should see a menu link to **HMIs** in the top menu bar.

Click this link, and you should see an **Edit** button, which will take you to the HMI editor.

# The HMI Editor




The editor consists of a menu bar, three editor sections, a view frame, and a footer.

- The menu bar contains a dropdown to select the HMI to be edited; and buttons for managing settings for the interface, the assets hosted by SmartSpace, and buttons to save, publish and delete the interface.
- The editors are for HTML, CSS and JavaScript.
  - Double-click in the title bar of an editor to enlarge it or return it to normal height.
  - Drag the middle separator bar to make the editors wider or narrower.

  - The editors support syntax highlighting and some shortcut keys. Click the **?** button to see a quick help:

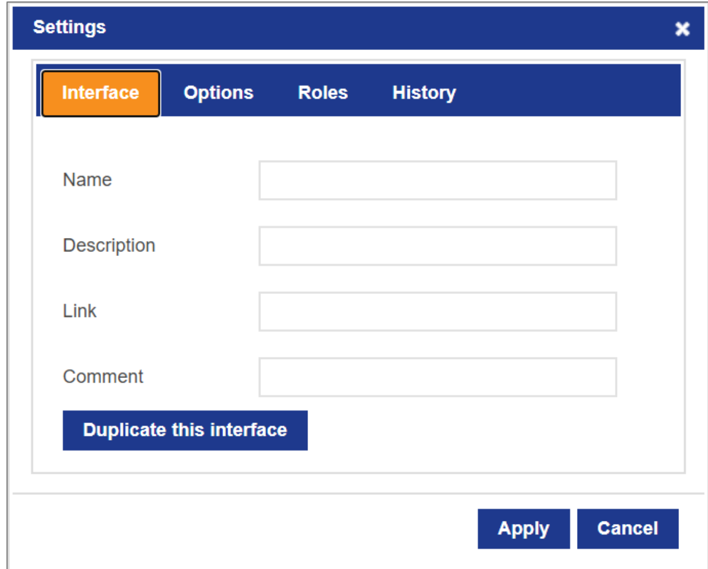
| Key | Action |
| --- | --- |
| Shift+Tab | Auto-indent the current line or selected lines |
| Ctrl+Q | Toggle comment for the current line or selected lines |
| Ctrl+Space | Show completion hints for the current word in context |
| Ctrl+F | Find a string in the editor. Enter the string to search, and press return to show the first match. Use CTRL+G to search for more. |
| Ctrl+G | Find the next instance of the string in the editor |
| Ctrl+S | Save and view the interface – the same as clicking the Save button on the top menu bar |

  - The editors also support undo and redo, and other standard text editing features, using the normal key bindings for your computer (e.g. Ctrl+Z, Ctrl+Y on Windows).
- The footer is only shown after a saved HMI has been shown in the view frame, and includes information about the currently viewed version of the HMI: the name, link, version number, when it was saved, whether it is the published version, and a button to refresh the view.

## Creating a new HMI

To create a new HMI:

1. Choose **<new HMI>** from the dropdown in the menu bar.
   Alternatively you can copy an existing HMI by selecting it in the dropdown, then clicking **Settings**, and then **Duplicate this interface**.
2. In the Settings dialog, fill in the name of the HMI, description, and link.




- The name and description will be visible to the users when the interface has been published.
- The link should be a short word or phrase that can be used in the URL to get to the interface once it has been published. It should contain no spaces or punctuation.

  You can set the link to "_" to make this HMI the landing page for all your SmartSpace Web pages. See *Setting an HMI as the Landing Page for SmartSpace Web*.
- You can also add a comment for internal notes. This will not be visible to users.

3. Click **Apply** to close the Settings dialog, and then click **Save** to write the interface back to the system, and show the interface in the view frame.

## Saving an HMI

To run an HMI in the view frame, you must save it back to the platform. Either click **Save** on the menu bar, or press Ctrl+S. The current HTML, CSS and JavaScript, along with interface settings, are written as a new version of the interface, and this version is loaded into the view frame.

Saved interfaces will be executable only by the administrator until they have been assigned to roles and then published.

The platform keeps the last ten saved versions of each interface, as well as the currently published version. Use the History tab in the Settings dialog to access older versions of the interface. See *HMI Settings* for further information.

## Errors in the HMI

When the HMI is viewed in the editor, any error encountered will cause an error window to be shown in red at the bottom of the screen. Click the arrow button to see a detailed error message.

If the error is in the script, the editor will attempt to convert the line number to match the line number in the script editor. This will only work if the DefaultAPI is used. See *The HMI API*.

You can also use the browser debugger to see more details about any error, insert breakpoints, and step through code, etc. For many browsers, this is opened by pressing F12 while viewing the page.

## Reloading an HMI

If you want to reload the current version of an HMI, click the refresh button in the bottom right of the footer:

This forces the view frame to reload without saving a new version of the interface. If you have made any changes in the editor that have not been saved, they will *not* appear in the view frame.

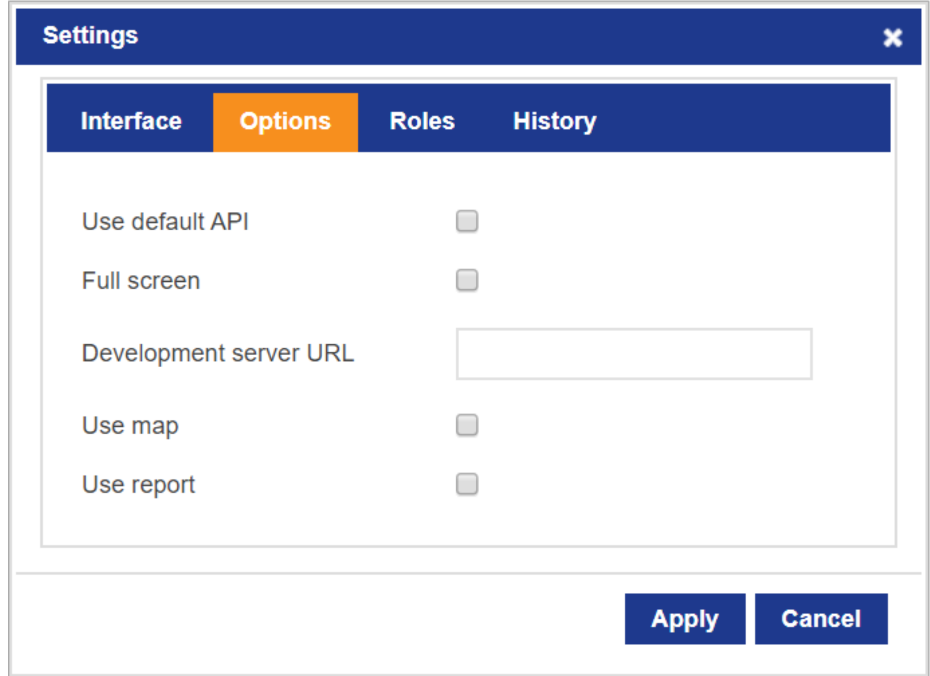
## HMI Settings

The Settings dialog is opened by clicking the **Settings** link on the menu bar. In addition to the interface settings described in *Creating a new HMI*, there are tabs for options, roles and history.

### Options

The Options tab contains the following settings:

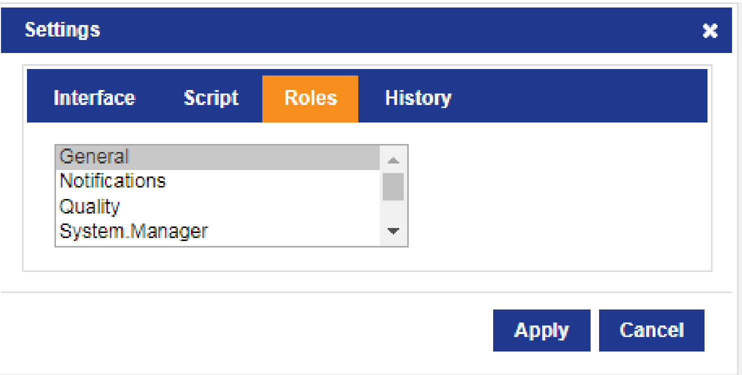
- The Use default API setting determines whether to include the default API for building HMIs. If you need to create a hosted web page that contains no standard SmartSpace HTML wrapper, JavaScript or CSS, clear this check box
- Selecting the Full screen option causes the HMI to occupy the full screen and hides the main menu bar
- To use the external development workflow, enter the local URL of the main HTML snippet as located on the external development web server. See *External Development Workflow* for more information on setting this option
- Select Use map to get the web map component and supporting scripts in the external development workflow
- Select Use report to use the reporting component and supporting scripts in the external development workflow




## Roles settings

The Roles settings tab allows you to specify the roles whose users will be allowed to visit an HMI. To select a single role, click it in the list.

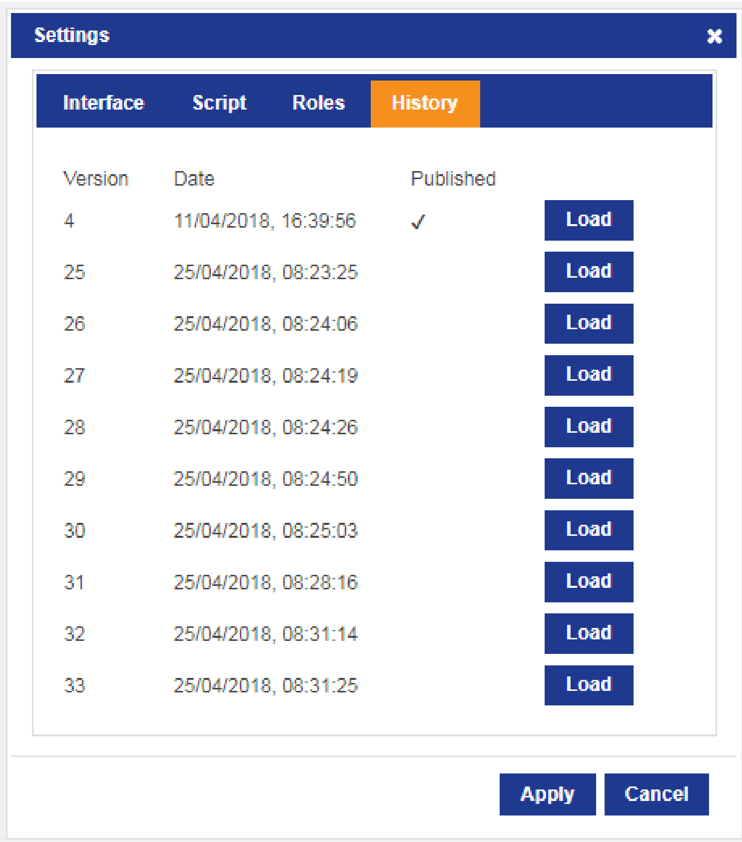
Hold down Ctrl and click to add multiple roles. Hold down Shift and click to add a consecutive range of roles.



In order for users to both access an HMI and view its contents successfully, in addition to specifying the role that enables access to an HMI you must also ensure that any search or query used by the HMI has also been added to the same role. For further information on assigning roles, see Users and roles (for web searches) and Defining Queries (for queries) on the Ubisense Documentation Portal.

## History settings

The History settings tab allows you to see previous versions of the interface that have been retained by the SmartSpace platform. In addition to the currently published interface, the platform retains, by default, ten versions of each HMI. You can go back to a particular version by clicking the **Load** button next to that version. This immediately loads the selected version and shows it in the view frame. You can then edit the loaded HMI definition, and click **Save** to write an updated version, which will be given a new version number.

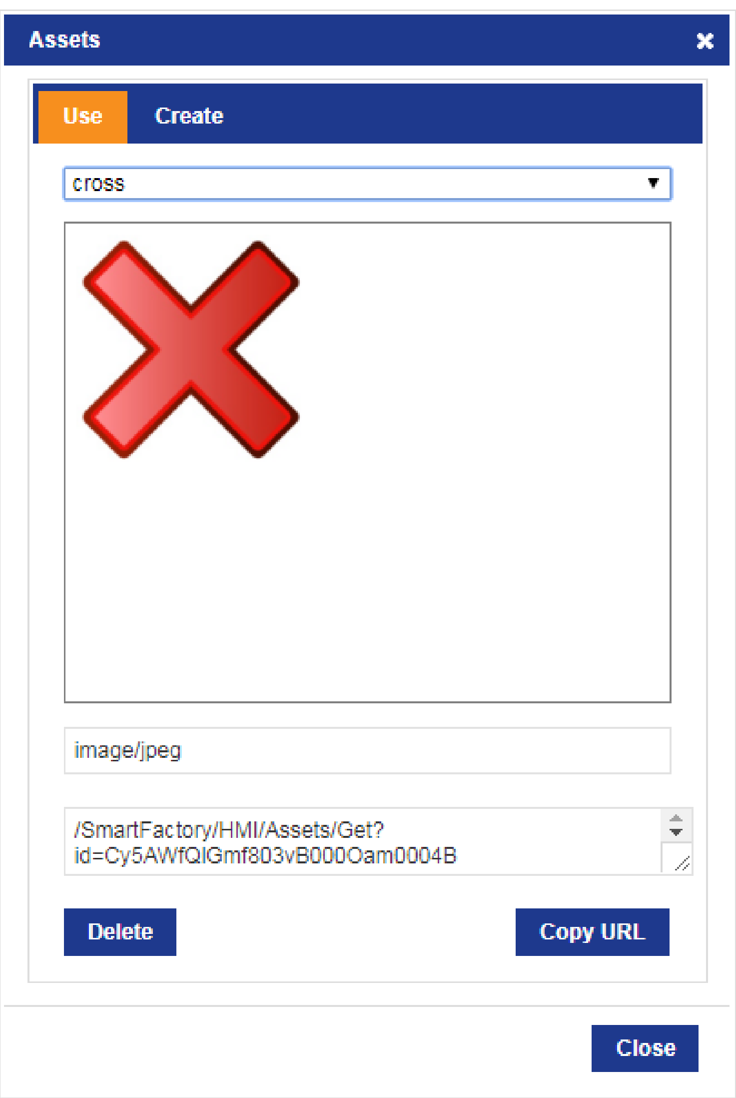



For example, with the above HMI, clicking **Load** next to version 4 will load the currently published version of the interface. This can be edited and then saved, which will create a new unpublished version number 34.

## Assets

The assets dialog allows you to upload files such as images, JavaScript libraries or CSS files so that they are hosted on the SmartSpace website. They can then be accessed via a URL, so you can use them in your HMI definitions.

To manage assets, click the **Assets** button in the menu bar.

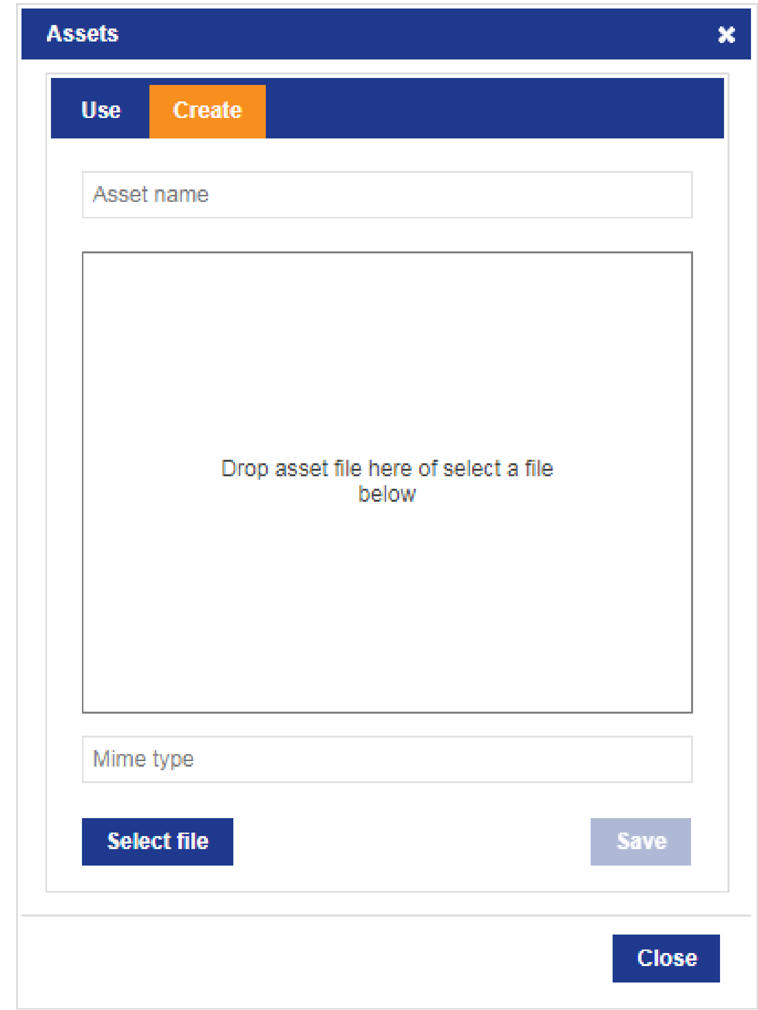



There are two tabs in the dialog. The **Use** tab allows you to select from uploaded assets. A preview of the selected asset is shown, and the URL used to access the asset from within an HMI is also shown.

- Click **Copy URL** to copy the URL text to the clipboard.
- Click the **Delete** button to remove a selected asset.

Note: There is no attempt made to track references to assets within HMIs, so deleting an asset that is used in a published HMI version will cause that HMI to break. Be careful to only delete assets that are no longer required.

The **Create** tab allows you to upload a file from your computer as a new asset definition.




Drag the file into the dialog to select it for upload. A name will be suggested, but can be edited for clarity, and the mime type of the asset will be deduced. If drag doesn't work, click the **Select file** button to show a file dialog. Click **Save** to store the asset, then switch to the **Use** tab to copy the URL for use in an HMI.

NOTE: Because asset data is stored in the SmartSpace platform, it takes up disk space both on the server and cached on the website under IIS control. The platform cleans up asset data when the asset is deleted. Thus when a large asset is no longer required it is recommended that it should be removed to free up disk space. However it is not recommended that you delete assets that are used in unpublished versions of HMIs

that are still retained by the SmartSpace platform as this will break those versions of the HMIs.

## Publishing an HMI

Once you have tested the functionality of an HMI, and the HMI has been assigned to a set of roles, you can publish the HMI. Only one version of a given HMI is published. This is the version of the HMI that is presented to users when they navigate to the HMI. Only an administrator can view unpublished versions of an HMI.

To publish an HMI, load the version of the HMI you want to publish, and then click **Publish** on the menu bar. The footer will change to indicate that the version shown is the published version.

If you have not yet assigned any roles to the HMI, then it will not appear under HMIs in SmartSpace Web.

## Setting an HMI as the Landing Page for SmartSpace Web

You can use an HMI as the landing page for SmartSpace Web. This means that the pages /SmartSpace, and /SmartSpace/Auth will redirect automatically to the published version of this HMI.

When you create (or edit) the HMI, in the Settings tab of the Interface dialog, set the Link property to "_". See *Creating a new HMI* for further information. The HMI should be given roles such that any valid user can access the page. So if unauthenticated users are allowed to visit SmartSpace, this HMI should be visible to unauthenticated users. See *Roles settings* for further information.

You can also specify the title of the HMI in code so it will appear on the browser title bar. For example:

```
window.titleSymbol = "Welcome to SmartSpace for ACME Industries";
```

This property is copied to **document.title** as the HMI API is initialized.

## Viewing an HMI Version Outside the Editor

To see what the currently loaded version of the HMI will look like outside the editor, click the **Visit** link on the menu bar. You can hold down Ctrl to open in a new tab, or Shift to open in a new

window.

## Exporting and Importing HMIs

There is a command-line tool to export and import HMI definitions to/from file. To get the tool, run **Application Manager**, select the **DOWNLOADABLES** task, and select **Visibility/HMI configuration tool**. Click **Download**.

If you have licensed the Rules engine developer, you can use the BUSINESS RULES workspace to load HMIs (and any dependencies) that have been developed in other SmartSpace installations, and you can export HMIs you have created, for use elsewhere. See the *Module import and export* guide on the Ubisense Documentation Portal.

### Exporting HMIs

To export HMIs, use the **export** mode. This writes the definition in JSON format. Use **-h** to specify a particular HMI to export by name, otherwise all defined HMIs will be exported. Use **-o** to write to a file. If you include **-a**, then the tool will look for assets used in the HMIs, and will save the contents of all used assets to files in the current folder.

```
$ ubisense_hmi_config.exe export -a -h "Andon" -o andon.json
export interface Andon
  export asset CF052A0A5AC3331A29AB5BC000004F6400000096.asset
```

### Importing HMIs

The tool can import a saved JSON file containing HMI definitions, and any referenced asset files. Change directory to the folder containing the **.json** file and the **.asset** files, then use the **import** mode. To import from a file rather than stdin, use **-i**. If the tool detects referenced assets in the interface, it will attempt to load them from the current directory.

```
$ ubisense_hmi_config.exe import -i andon.json
  import asset CF052A0A5AC3331A29AB5BC000004F6400000096.asset
import interface Andon
```

If you import an interface that currently exists in the SmartSpace platform, the exported version will be restored. However it may be quickly cleaned up if too many retained versions with higher version numbers have been saved. In this case, edit the JSON file to increase the version number to a version higher than the maximum version number already present, then reimport the HMI.

# Loading a Built Web Application as an HMI

Externally-developed Single Page Applications (SPAs) can be hosted as HMIs on the SmartSpace website. The website can either serve the index.html of an SPA with no additional JavaScript being loaded, or the SPA can be rendered in the HMI wrapper and the HMI API loaded.

To load a built web application into SmartSpace for use as an HMI, use **ubisense_hmi_config** in **load** mode.

To make it easy to use with node builds, load mode takes its parameters from a JSON file. For example:

```
{
  "name": "Vue SPA Test",
  "description": "A test single page application built using Vue without the HMI api",
  "path": "./dist/",
  "html": "index.html",
  "link": "spa",
  "roles": ["General User"]
}
```

The fields in the JSON file have the following meanings:

| Field | Meaning |
|---|---|
| name | The name of the HMI, as it will appear in the list offered to users |
| description | The additional description as it appears on the list offered to users |
| path | Where the built application files are found – usually the output folder of the build tools |
| html | The main HTML file of the application; if not specified this is "index.html" |
| link | The URL path to the HMI within SmartSpace |
| roles | An array of roles, defined using the Users and roles feature, to which the HMI will be accessible |
| embed | A Boolean. If true, then the HTML is rendered within the SmartSpace HMI wrapper, and the HMI API will also be loaded. If false (the default), then the HTML is served exactly as it is, with no additional JavaScript loaded. Because anything under namespace "U.WM" is normally loaded by the wrapper, if embed is false, access to user and roles information will not be available automatically. Instead, you can use the endpoint described in *Getting the Logged-in User and their Roles*. |

The load method finds all files under the given path, and loads them as assets of the HMI. It automatically publishes this version of the HMI, replacing any existing HMI version with the same name. For example:

```
ubisense_hmi_config load smartspace.json
```

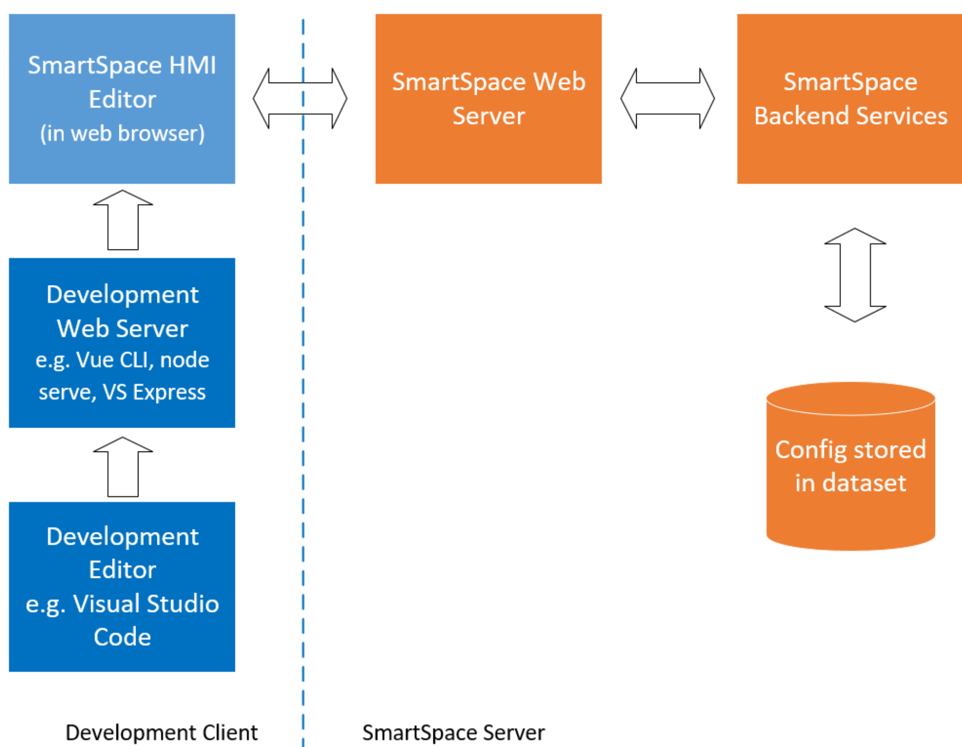
If you are using a node-based toolchain to build the web application, you can set up **package.json** so that the "postbuild" script loads the HMI. For example:

```
"scripts": {
  "serve": "vue-cli-service serve",
  "build": "vue-cli-service build",
  "postbuild": "ubisense_hmi_config load smartspace.json",
  "lint": "vue-cli-service lint",
  "serve:prod": "vue-cli-service serve --mode production"
},
```

# External Development Workflow

From release 3.6, SmartSpace HMIs support an external development workflow. In this workflow, rather than edit HTML, CSS and JavaScript inside the HMI Editor, the developer works using an external development environment, such as Vue CLI, that presents a website containing the current compiled/minified version of an HMI. The external development toolchain can use standard version control, and more powerful editors, to increase productivity.

The HMI Editor within SmartSpace watches this external development website for changes, and automatically incorporates those changes into the current version of the edited HMI. The developer can then publish the HMI when it is ready to be used.




An HMI that uses the external development workflow is configured with the URL from which it should retrieve the main HTML snippet for the interface. The editor will load this snippet and render the resulting HMI within its frame.

The editor now watches for relative assets loaded by the rendered HMI, such as images, scripts, and CSS files. For example, the main HTML might load an image using a relative path:

```
<img alt="Customer logo" src="./assets/logo.png" />
```

The HMI Editor sees this request for a relative path asset, retrieves it from the development server, and stores it in SmartSpace as a local asset for the current HMI version. It subsequently polls the development server looking for changes to the main HTML snippet or any of the local assets, and whenever one changes the whole HMI is refreshed in the editor.

## Setting up an External Development Server

To use the external development workflow, the HMI you are editing must be available via a development web server for SmartSpace to import. The import is performed by the HMI Editor website, and this requires your development server to enable cross-site scripting (XSS) access. This means that the server responses must include the following headers:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: If-Modified-Since, If-None-Match
Access-Control-Expose-Headers: ETag, Last-Modified
```

The first header allows the HMI Editor to access the resources served out by the development web server, the second and third allow it to reliably detect changes to those resources.  These headers are only used by the HMI development workflow; they are not enabled for the HMI when it is in production in SmartSpace.

## Project Features for the External Development Workflow

For the external development workflow to work reliably, the following project features should be used.

- Ensure that your project uses relative paths to access its assets (images, scripts, style sheets), otherwise they will not be detected as local assets and will not be imported and saved in SmartSpace.
- The main HTML file served out should not be wrapped in an “<html>” element. Instead it should be a snippet such as a “<body>”. SmartSpace wraps the snippet you provide inside an outer “<html>” document, with headers, style sheets, and references to the HMI API. Having an “<html>” element nested inside another is not supported in some browsers and is not recommended.
- To avoid JavaScript warnings and compilation errors, use **window.UbiHMI** rather than **UbiHMI** so that the compiler knows this is an external class.

This is because the **UbiHMI** object is not known to the Vue CLI toolchain, because it is in an externally-provided library (included by the SmartSpace server), so various warnings are generated by the syntax and type checkers within the Vue CLI development toolchain. However, **UbiHMI** is a global object, so it is actually equivalent to **window.UbiHMI**, and the window global variable *is* known in Vue CLI. So using **window.UbiHMI** prevents the warnings.

- Use dynamic ("lazy") loading of your main components, so that the HMI API is available when your components are instantiated. Otherwise you might be unable to find key API functions such as **window.UbiHMI.mixin**.
- In your main JavaScript file, instead of importing "Vue", use the one already loaded as **window.Vue**. The HMI API ensures that this version of the Vue library has all the API components, such as **ubi-map** and **ubi-report**, imported and ready to use.
- Be careful with "hot reload" development servers, such as Vue CLI in development mode. While the development mode Vue CLI server does work, it is not recommended for this workflow because the HMI created will have all this development mode hot reloading code. It is recommended to use production mode for this workflow, or to turn off the "hot reload" in your project.

See *Example of External Development Workflow with Vue CLI* for a complete example of how to set up a Vue CLI.

## Setting an HMI to use the External Development Workflow

To use the external development workflow, create an HMI in the HMI Editor as normal (see *Creating a new HMI*. Open the Settings dialog and go to Options to edit the **Development server URL** option. Enter the local URL of main HTML snippet as located on the external development web server. For example, using a Vue CLI project, where the HMI HTML snippet is in **index.html**, this might be "http://localhost:8080/index". Click **Apply** to save the setting, and then click **Save** to write this back to SmartSpace.

The layout of the HMI Editor will now change. The HTML, CSS and Script editor panes will be hidden, and the preview pane will fill the main editor window. If the URL entered was correct, and the development server correctly configured, then the HMI will now appear within this preview pane. The editor will now refresh the page and its assets if any of them change at the development web server. The refreshing of assets can be seen in the bottom right pane of the editor footer.

There are a few things to be aware of when using the external development workflow:

1. SmartSpace attempts to reduce the download size of an HMI by not including map and report scripts and style sheets unless they are required. Since the external development workflow can dynamically load components that might use "<ubi-map>" or "<ubi-report>" components, it is not possible for the editor to work out statically whether to include these libraries. The settings page of the HMI Editor now includes options for explicitly enabling map and report. See *Options*.
2. Because the HMI Editor can update the local assets for the current version of an HMI, the "Publish" action now also saves a new version of the HMI after it has been published. This protects the published version from being modified by mistake.
3. If your development web server is not running when you visit the HMI in the HMI Editor, then the interface will not be displayed, and no changes will be loaded. This is not the same as visiting the HMI outside the HMI Editor, where all the assets are served by the SmartSpace website directly.

## Production Builds in the External Development Workflow

Before you run production builds, ensure you have added the following to **package.json** under the scripts key (you will see the other startup scripts there, such as serve and build):

```
"serve:prod": "vue-cli-service serve --mode production"
```

To perform a production build, stop the whole project and run the following command:

```
npm run serve:prod
```

This will build the production html files and serve them so that SmartSpace can pick them up. It will also prune the size of the deployment.

You can further reduce the size of production files by purging unused content from Tailwind CSS. With Vue, add the following to your **tailwind.config.js**:

```
purge: {
  enabled: true,
  content: [
     './src/**/*.vue',
    './public/**/*.html',
  ]
},
```

```
purge: {
```

```
    enabled: true,
    content: [
      './src/**/*.vue',
     './public/**/*.html',
    ]
  },
```

## Example of External Development Workflow with Vue CLI

We will now go through a basic example using Vue CLI for the external development workflow.

- Set up a new Vue CLI project – run **vue ui** and in your browser visit the project manager. Click "Create" and give the project a name. We will use "hmi" as the project name.
- Go to your project "Configuration" page, and
  - Set "Public Path" to the empty string '', so that assets are linked with relative paths. Click "Save Changes".
  - Click "Open vue config" at the top of the page, and in your editor, add the required headers for the development server:

```
module.exports = {
 configureWebpack: {
  devServer: {
   headers: {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Headers': 'If-Modified-Since, If-None-Match',
    'Access-Control-Expose-Headers': 'ETag, Last-Modified'
   }
  }
 },
 publicPath: ''
};
```

```
module.exports = {
 configureWebpack: {
  devServer: {
   headers: {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Headers': 'If-Modified-Since, If-None-Match',
```

```
        'Access-Control-Expose-Headers': 'ETag, Last-Modified'
      }
    }
  },
  publicPath: ''
};
```

- In your editor, under the public folder, edit **index.html**. This should be an HTML snippet rather than a full web page, so remove all but the body:

```
<body>
  <div id="app"></div>
  <!-- built files will be auto injected -->
</body>
```

```
<body>
  <div id="app"></div>
  <!-- built files will be auto injected -->
</body>
```

- In your editor, open the **src** folder, and edit **HelloWorld.vue**. Remove the default generated markup, and Insert a map component:

```
<template>
 <div class="hello">
  <h1>{{ msg }}</h1>
  <ul>
   <li>
    <ubi-map ref="map" width="500px"></ ubi-map>
   </li>
  </ul>
 </div>
</template>
```

```
<template>
 <div class="hello">
  <h1>{{ msg }}</h1>
  <ul>
   <li>
    <ubi-map ref="map" width="500px"></ ubi-map>
```

```
    </li>
  </ul>
 </div>
</template>
```

- In your editor, open the **src** folder and edit **App.vue**. Change the import to use dynamic loading. This ensures that the Ubisense HMI will be available when your HMI module is loaded:

```
…
<script>
const HelloWorld = () => import("./components/HelloWorld.vue");
…
```

```
…
<script>
const HelloWorld = () => import("./components/HelloWorld.vue");
…
```

- Edit the **main.js** file, and switch to using the preloaded Vue library, so that the Ubisense API components are available.

```
…
// Import the Vue from SmartSpace to get the components.
var Vue = window.Vue;
import App from './App.vue'
…
```

```
…
// Import the Vue from SmartSpace to get the components.
var Vue = window.Vue;
import App from './App.vue'
…
```

- Go to your project “Tasks” page, and select “serve”:
    - Click “Parameters”, and under Specify env mode select “production”. It is probably wise to specify host “localhost” and port “8080”, so that the web server doesn’t

change port. Click Save.
    - Now click “Run task” to start the development server.
- In SmartSpace, open the HMI Editor.
    - Create a new HMI called “VueTest”. In the Interface tab of the Settings dialog, set the link to “vuetest”.
    - In the Options tab, enter the URL of the running development server main snippet: **http://localho**
    - Select **Use map** to get the web map component and supporting scripts.
    - Click **Apply**, then Save the HMI.

You should see an HMI including a map in your HMI Editor. Editing the project in Vue, and clicking save, will cause the changes to be imported into the HMI Editor automatically.

If you do need to run the Vue CLI server in development mode, turn on “disableHostCheck” in your Vue config:

```
module.exports = {  configureWebpack: {
    devServer: {
      …
      disableHostCheck: true
…
    }
  }
};
```


```
module.exports = {  configureWebpack: {
    devServer: {
      …
      disableHostCheck: true
…
    }
  }
};
```


Also make sure you have set the hostname and port for the development server in its “Properties” page.

## Exporting/Importing HMIs with the Development Workflow

HMIs developed using the external development workflow can be loaded and saved, and copied between platform datasets, using the standard tools.

- The current releases of the **ubisense_hmi_config** tool know about local assets for an interface, and will save and load them if the **-a** flag is given.
- In SmartSpace Config, the Business rules Load and Save features include local assets when saving and loading HMIs. See Module import and export for further information.

# The HMI API

In this section we describe the default API that is provided to make it easy to generate HMIs that interact with the SmartSpace business objects and properties. The HMI API is based around the **Vue.js** library, and provides a binding between HTML and the results of SmartSpace web searches and, if the Reporting component is licensed, queries. The API also provides some built-in Vue components that are useful for creating HMIs, and wraps the SmartSpace web map and SmartSpace reports as components.

## Data Binding for Web Searches

Data binding supports the creation of reactive data corresponding to the results of selected SmartSpace web searches. It allows HTML elements, including the built-in components, to be instantiated and have their attributes, properties, and contents bound based on the results of those searches.

### Declarative Web Searches

The simplest way to access web searches is to use the **udm** attribute on an element in your HTML. The **udm** attribute takes a comma-separated list of data names and corresponding web searches:

```
<div udm="dataName: {search:'Web Search'}, dataName: {search:'Web Search'}, …"
```

The data names can be any legal JavaScript identifiers. The web searches must be valid web searches defined in the SmartSpace platform. For information on how to define web searches, see Web searches on the Ubisense Documentation Portal.

Within contents of this element, each **dataName** will be available as reactive data. The following properties will be available to be bound – see *Binding Web Search Results* for how to use these properties.

| Property | Contents |
| --- | --- |
| **dataName.rows** | The rows returned by the search. These are updated at the refresh interval configured for the web search. The keys of rows are the object identifiers, and the values are:<br>• Summary – an array with one element for each summary property returned by the web search<br>• R – internal representation data used by the map to render the object – reserved for future use |
| **dataName.selected** | When this property is set to a valid object identifier, such as one of the keys from the rows above, this causes the details property to be set with the details results for the web search. |
| **dataName.details** | When **dataName.selected** has been set, this property receives the details results for that object from the web search. The results are an array of value structures. See below for the format of value structures. |

So, declaring a **udm** attribute causes the API to fetch the results of the specified search, and store them under "rows" in the specified data name, and also detect when selected is set and update the details for the selected object. You can use this data by binding as described below, or directly within your JavaScript.

Note that the search is run as the role under which the HMI page was loaded – it must be available to that role or it will fail to return any results. For further information on adding searches to roles, see Users and roles on the Ubisense Documentation Portal. Searches are executed for all views, so even objects with no location will be returned.

The details property of a search contains an array of value structures. The fields in these value structures depend on the type of the property returned, which is indicated by the Type field. This is an integer with the following definitions:

| Type Field Value | Contents of Value Structure |
|---|---|
| 0 | string property in **value.String** |
| 1 | date/time property in **value.Date**, which is UTC seconds since 1/1/1970, so you can create a JavaScript using new Date(value.Date) |
| 2 | number in **value.Number** |
| 3 | object identifier in **value.Id** |
| 4 | boolean in **value.Boolean** |
| 5 | html/css color string in **value.Colour** |

## Complex Properties in Searches

When search details include a complex property, multiple rows can be returned–one for each row in the complex property that contains the selected object. The first row returned has the “Title” set to the property name, or the title, as configured in the search. Subsequent rows will have an empty Title.

The following example shows two complex properties returned in details, with three rows each:

```
{ "String": "Building 5", "Title": "Current Vehicle Areas", "Type": 0 },
{ "String": "Offtracks", "Title": "", "Type": 0 },
{ "String": "Paint Inspection Bay 6", "Title": "", "Type": 0 },
{ "String": "P199761A", "Title": "Vehicle Quality Tickets", "Type": 0 },
{ "String": "P199761B", "Title": "", "Type": 0 },
{ "String": "E002136", "Title": "", "Type": 0 }
```

## Binding Web Search Results

You can use standard Vue binding to use the data from web searches within the contents of the element on which the **udm** attribute is set. Typically you will use:

- **v-for** to create an element for each row returned by the search
- **v-bind** (or “**:**”) to bind element attributes
- the “Mustache” syntax (double curly braces): **{{ }}** to bind properties into element content
- **v-model** to use a two-way binding between data and the value of a control

Vue.js binding is very powerful, and we do not cover all the possibilities here. See the online documentation of **Vue.js** template syntax for more details.

For example, assuming there is a “Products” search defined, and the first summary result of the search is the name of the product object, then the following HTML will result in a div for each product containing the product name:

```
<div id="app" v-cloak udm="products:'Products'">
  <div v-for="(row, key) in products.rows">
    {{ row.Summary[0] }}
  </div>
</div>
```

As another example with the same search, the following HTML creates a select dropdown control where the text of each option is the name of the product, and the value is the product object identifier:

```
<div id="app" v-cloak udm="products:'Products'">
   <select>
      <option disabled value="" >select a product </option>
      <option v-for="(row, key) in products.rows" :value="key">
        {{ row.Summary[0]  }}
      </option>
   </select>
</div>
```

Note the use of the **v-cloak** attribute, which hides the contents of the div until the data has been bound. This prevents the ugly template syntax from being shown to the user.

## Search Parameters in HMI Queries

You can programmatically specify search parameters. There are a range of reasons why this would be useful:

- It eliminates the need to write JavaScript to filter the results which can sometimes be challenging
- If the search returns a very large number of results, filtering the results on the client side can lead to performance issues if the client device has very limited resources. By filtering the results on the server using parameters this load can be reduced
- If the search involves a large number of joins, using a parameter may also reduce server load

There is a **.params** property on each bound search, which is an array of parameters to pass. Setting a parameter causes the search to execute again.

The items in the array correspond to each search parameter in the order they are defined. Values accepted depend on the type of the parameter:

- String: pass a substring to search, or a comma or space separated set of values if the parameter takes multiple value. For example ["BN123"].
- Time: each date can be either a Date() instance, or a string that can be parsed as a date. There are three supported formats:
  - A single value: this is used as the "from" part of the date range. For example ["2026-03-21"]
  - A struct with "from" and/or "to" fields. For example [{from:"2026-03-21",to:"2026-03-27"}], or [\{to: "2026-03-27"}]
  - An array with the first value being from and the second to. For example [["2026-03-21","2026-03-27"]], or [[undefined,"2026-03-27"]]
- Number: as Time, but each value is either a number, or a string than can be parsed as a number. The same three formats can be used.
- Boolean: any of the usual JavaScript true/false values. For example any of the following:

| Value | Boolean |
|---|---|
| true | true |
| false | false |
| 1 | true |
| 0 | false |
| "" | false |
| "anything" | true |
| null | parameter not specified, so true and false |

- Object: a string object ID. For example ["04007zWKXLF9Q4sM000KF00000y:UserDataModel::[Custom]Product"], where the search parameter must be defined as choosable (see Making properties editable by roles)

For choice parameters, only exact values are matched, so the "from" and "to" formats are not supported. However there is no requirement to pass one of the "allowed" parameters defined by the Web Search configuration.

The initial parameters can be specified in the binding object too, and this can be used to avoid returning all rows of a search by default. For example:

```
"
{ search: "Products", params: ["-------"] }
"
```

The following examples show how to define an array of four parameters whose types are [String Bool Int Time] for the Products web search in a UDM attribute and as a UbiHMI.mixin method.

#### UDM attribute

```
<div udm='{"products":{ "search": "Products", "params": ["4961", false, 591, "2017-01-
01"] } }'>
```

#### UbiHMI.mixin method

```
mixins: [ UbiHMI.mixin({ products: { search: 'Products' , params: ["4961", false, 591,
"2017-01-01"]} }) ]
```

### Refreshing Searches

Searches automatically refresh at the same refresh interval as is configured in the web search (in SmartSpace Config/WEB SEARCHES). They also refresh immediately after you call **setProperty** (see *Setting Object Properties*). If you need to force a refresh otherwise, the Vue object also includes a **refreshSearches** method that forces an update of the results of all bound searches.

Search details, returned if you have set the selected property on a bound search, will be refreshed once per second.

## Data Binding for Reporting Queries

Data binding also supports the creation of reactive data corresponding to the results of selected Reporting queries. It allows HTML elements, including the built-in components, to be instantiated and have their attributes, properties, and contents bound based on the results of those queries.

## Declarative Queries

The simplest way to access queries is to use the **uquery** attribute on an element in your HTML with the query. The **uquery** attribute takes a comma-separated list of data names and corresponding queries:

```
<div uquery="dataName: 'Query Name', …"
```

The query names can be any legal JavaScript identifiers. The queries must be valid queries defined in the SmartSpace platform. For information on creating queries, see Defining Queries on the Ubisense Documentation Portal.

Within contents of this element, each **dataName** will be available as reactive data. The following properties will be available to be bound – see *Binding Query Results* for how to use these properties.

| Property | In/Out | Contents |
|---|---|---|
| **dataName.rows** | out | An array of results returned from the query. Each row is an object with properties for each result column. |
| **dataName.count** | out | The total number of results from the query (ignoring paging, applying filters/values). |
| **dataName.progress** | out | The percentage progress of executing the query. Can be used for a progress bar or indicator. |
| **dataName.page** | in | { Size: A, N: B }, defines the page size and page number to return. For example { Size: 100, N: 3 } will return result rows 301-400. The default Size is 100000, to avoid massive query results, but this can be manually adjusted if necessary. |
| **dataName.values** | in | Parameters to pass to the report. See *Parameters and Filters*, below. |
| **dataName.filters** | in | Filters to pass to the report. See *Parameters and Filters*, below. |
| **dataName.disabled** | in | Set true to prevent execution of the query. |

So, declaring a **uquery** attribute causes the API to fetch the results of the specified query, and store them under "rows" in the specified data name. You can use this data by binding as described below, or directly within your JavaScript.

Note that the query is run as the role under which the HMI page was loaded – it must be available to that role or it will fail to return any results. For further information on adding queries to roles, see Defining Queries on the Ubisense Documentation Portal.

#### Parameters and Filters

To set parameters, the values attribute is used. The object can have one property for each parameter of the report. The value of each property is an object { Type: T, Value: V }. For example, with a dataName "steps", you can set a string query parameter "BuildNo" as follows:

```
this.steps.values["BuildNo"] = { Type: "String", Value: 'AMN' }
```

The types supported are:

- String: Value is just a string
- Date: Value is the number of seconds UTC since 1$^{st}$ Jan 1970
- Double: Value is a number
- Bool: Value is true or false

To set filters, the filters attribute is used. The keys of the object are query names. The value for each source is another object with keys corresponding to the column names of that source, and values specifying the filter values. For example:

```
this.steps.filters["ProductProcessHistory"] = {
                        ProductName:'V12817',from:'one_week'}
```

For string columns, the value is just a substring. For date columns, the filter value can be one of the predefined date range strings, or a custom date range object.

The predefined date range strings are:

| | | | | | |
|---|---|---|---|---|---|
| this_year | this_quarter | this_month | this_week | today | last_year |
| last_quarter | last_month | last_week | yesterday | forever | two_years |
| one_year | one_quarter | one_month | one_week | one_day | |

A custom date range looks like this:

```
{
  From: { Type: 'Time',  Value: <sec since 1/1/1970 UTC> },
  To: { Type: 'Time',  Value: <secs since 1/1/1970 UTC> }
}
```

## Binding Query Results

You can use standard Vue binding to use the data from queries within the contents of the element on which the **uquery** attribute is set. Typically you will use:

- **v-for** to create an element for each row returned by the query
- **v-bind** (or "**:**") to bind element attributes
- the "Mustache" syntax (double curly braces): **{{ }}** to bind properties into element content
- **v-model** to use a two-way binding between data and the value of a control

Vue.js binding is very powerful, and we do not cover all the possibilities here. See the online documentation of **Vue.js** template syntax for more details.

For example, assuming there is a "Vehicle in Process Step" query defined, with results BuildNo and ProcessStep, then the following HTML will result in a table row for each row containing the build number and the process step:

```
<div id="wrapper" uquery="steps: 'Vehicle in Process Step'">
  <div v-cloak>
    <table>
      <tr v-for="row in steps.rows" v-if="row.BuildNo">
        <td>{{ row.BuildNo }}</td>
        <td>{{ row.ProcessStep }}</td>
      </tr>
    </table>
  </div>
</div>
```

Note the use of the **v-cloak** attribute, which hides the contents of the div until the data has been bound. This prevents any ugly template syntax from being shown to the user.

## Refreshing Queries

Queries automatically refresh if you have specified a refresh interval when defining the binding (see *Manual Vue Construction*). They also refresh if you change any of the input properties on the bound data name, such as filters or values. If you need to force a refresh otherwise, the Vue object also includes a **refreshQueries** method that forces an update of the results of all bound queries that are not disabled.

# Using the API from JavaScript

## Callbacks

There are two standard callbacks you can use in JavaScript to do things once the search data has been set up.

The normal method is to use the **udm-bound** attribute on the same element you give a **udm** attribute. This attribute takes a JavaScript function that will be called when the **Vue.js** data has been created (but not yet retrieved or updated). The function receives an argument which is the Vue instance that has been constructed for the element.

There is also a global function called when all Vue instances have been constructed and bound. This is **UbiHMI.loaded**. Set this to a function that should be called, which takes a single argument that is an array of Vue instances, one for each element with the udm attribute. For example:

```
UbiHMI.loaded = function (vms)
{
  // Do something with the instances in array vms
  console.log(vms);
}
```

Alternatively, for more advanced cases, you can switch to manual Vue construction (see *Manual Vue Construction*) which gives full control over lifecycle and data.

## Using Bound Data in JavaScript

In order to use the bound data in JavaScript, you typically need to receive a notification when the data values change. You can use one of the callbacks to register to watch when the bound data changes. For example:

```
<div id="app" v-cloak udm="products:'Products'" udm-bound="onBound">
 ...
</div>
```

Then in JavaScript define the **onBound** function:

```
function onBound(vm)
{
  vm.$watch("products.rows",function (newVal, oldVal) {
    // do something whenever rows change
    console.log(newVal);
  });
}
```

Once you have got hold of the Vue instance, you can directly access the data on that instance. For example:

```
// Get the current product rows.
var rows = vm.products.rows;
// Iterate over them.
for (var id in rows)
{
   // Executes once for each product returned by the search,
   // with id as the object identifier.
   if (!rows.hasOwnProperty(id)) continue;
   var cols = rows[id].Summary;
   // do something with the summary columns.
}
```

## Setting Object Properties

The Vue binding also provides support for setting properties of objects back into the SmartSpace business objects and properties, just like the web map can set properties. The properties must be settable by the role under which the HMI is running. For information on configuring properties as settable see Making properties editable by roles on the Ubisense Documentation Portal.

To allow this, each Vue instance has a setProperty method. For simple properties, this takes the form:

```
setProperty: function (obj, prop, val, onSuccess, onFailure)
```

For simple properties, the arguments are:

- obj: the object identifier, such as one of the keys from search rows
- prop: the name of the property to set, including namespaces
- val: the value to assign – this must be compatible with the type of the property
- onSuccess: optional callback when the property has been successfully written back to the SmartSpace platform
- onFailure: optional callback on failure to set the property value – usually this will be because the wrong property name was given, the object identifier is invalid, or the current user is not a member of a role that can set the property. The function is called with three arguments:
    - operation: ‘set’
    - property: the name of the property being set
    - error: the response from the server

From SmartSpace release 3.8, the setProperty function supports complex properties. For complex properties, the method supports a JavaScript array as the first argument:

```
setProperty: function (args, prop, val, onSuccess, onFailure)
```

The arguments are the same as those described for simple properties with the exception of 'args'. When 'args' is convertible to a legal key for the property 'prop', the value of prop(args) is set to 'val'.

## Creating New Object Instances

The Vue binding includes the createObject function that simplifies creation of new object instances. It is entirely a client-side function,

and makes no change to the user data. Use *setProperty* to make the object known in user data back at the server.

The function takes the format:

```
createObject: function (type) {..}
```

It takes a type with namespaces.

For example:

```
var created = this.createObject("UserDataModel::[Custom]Product");
this.setProperty(created,"[Custom]name<[Custom]Product>",this.newProductName);
```

It should be noted that normally the first thing to do with a created object would be to set the name property. Objects of a named type that do not currently have a name will periodically be cleaned up at the server.

## Error Handling

A global callback is provided for errors encountered when invoking searches, getting details, or setting properties. To be notified of these errors, set **UbiHMI.error** to be a function that takes the following three arguments:

- error: one of
  - ‘searches’: could not get the list of searches available to the current role
  - ‘auth’: a search is not available for the current role

    - 'search': an error was encountered executing a search
    - 'details': an error was encountered getting details for the selected object
    - 'set': an error setting a property value
- property: the name of the property if any
- error: the response from the server if any
    - status: HTTP return code, such as 400
    - data: the data returned by the server, if any

## Manual Vue Construction

For more advanced control of data, methods, watches, etc., you can take control of constructing the Vue instances yourself, instead of using the declarative udm or uquery attributes. To make this easy, the API provides a method to get a Vue "mixin" that provides all the web search and query data binding functionality. This method is **UbiHMI.mixin**.

For example, the following uses JQuery **document.ready** mechanism to create a Vue once all the DOM has been loaded, and bind to the "Products" search and "ProductHistory" query:

```
$(function () {

   var vm = new Vue({
     // The HTML element for this vue instance.
     el: '#app-progress',
     // Mix in the web search 'Products' as data field products, and the query
'ProductHistory' as data field history with a 60 second refresh interval.
     mixins: [ UbiHMI.mixin({ products: { search: 'Products'}, history: { query:
'ProductHistory', refresh: 60} }) ],
     data: {
        // Declare some extra data in addition to products
       size: '',
       message: ''
     },
     methods: { // your methods here }
   });
 ...
 });
```

Within the methods of this view class, you will have access to the bound dynamic data in `this.products` and `this.history`.

For backward compatibility with previous syntax, the default binding is a web search, so the following binds to a single web search called "Products":

```
...
     mixins: [ UbiHMI.mixin({ products: 'Products' }) ],
...
```

## Tag Management

The Vue mixin also adds a method tagManager() that returns an interface used to check, associate and disassociate tags with objects.

The interface methods all take optional success and failure methods. The success method takes a single argument, and is called with the result of the method if it succeeds. The failure method is called with an error message on failure. The methods also return a promise, so this can be used to handle the results instead of the success/failure methods.

For an example of tagManager() methods, see Example Multi-tag in an HMI.

The tagManager() interface has the following methods:

### getPositions

```
getPositions: function (type, success, failure)
```

- Gets the list of defined tag positions for the given type.
- The type is a raw platform type name, so for example, if the user data model type is "Product", the type string would be "UserDataModel:[Custom]Product".
- The result on success is a list of position objects, each of which has a Name, Offset and Type parameter. Positions are returned for the given type and all its children. For example:

```
[
   { "Name": "Cab", "Offset": { "X": -1, "Y": 0, "Z": 0 }, "Type":
"UserDataModel::[Custom]Forklift" },
   { "Name": "Front", "Offset": { "X": 0.6, "Y": 0, "Z": 1.5 }, "Type":
"UserDataModel::[Custom]Forklift" },
   { "Name": "Back", "Offset": { "X": -0.6, "Y": 0, "Z": 1.5 }, "Type":
"UserDataModel::[Custom]Forklift" }
]
```

### getStatus

```
getStatus: function (tagid, success, failure)
```

Gets the current tag status and association.

- The tagid is a string tag id, e.g. "00:11:CE:00:00:10:83:B1".
- The result on success is an object with fields "Activity", "Battery", "Owner", "Tag", "TagType". For example:

```
{
  "Activity":"Inactive",
  "Battery":"OK",
  "Owner":{"Id":"4cbKlPY7ogK0000017XMA000002:UserDataModel::
[Custom]Forklift","Name":"WH0004","Offset":{"X":-0.6,"Y":0,"Z":1.5}},
  "Tag":"00:11:CE:00:00:00:00:13",
  "TagType":"Industrial tag (C cell)"}
}
```

#### getOwnerTags

```
getOwnerTags: function (owner, success, failure)
```

Gets the tags associated with a given object.

- The owner is the object id of the owner to check.
- The result on success is an array of tag status objects of the same form as returned by getStatus above.

#### associate

```
associate: function (tagid, owner, p, tagType, success, failure)
```

Associate a tag with an object.

- The tagid is a string tag id, e.g. "00:11:CE:00:00:10:83:B1".
- The owner is the object id of the owner to associate.
- Parameter p is either a position name, which must be valid for the owner type, or an offset {X:,Y:,Z:}.
- Optional parameter tagType is a defined tag type name, such as "Industrial tag (C cell)".

#### disassociate

```
disassociate: function (tagid, success, failure)
```

Disassociate just this tag from its object, if it is associated.

- The tagid is a string tag id, e.g. "00:11:CE:00:00:10:83:B1".

#### disassociateOwner

```
disassociateOwner: function (owner, success, failure)
```

Disassociate all tags from an owner.

- The owner is the object id of the owner to disassociate.

### Current User and Role

The Vue object also includes data bound to the logged in user and the role.

- vm.user: the user currently viewing the HMI.
- vm.role: the role under which the HMI was visited, if it was visited via the HMI chooser. This may be undefined if the HMI was visited directly via its shortcut URL, or when the HMI is loaded in the Editor.

For example, the following will print the user and role within an HMI:

```
<div v-cloak udm="{}">
  User is {{ user }}
  <br>
  Role is {{ role }}
</div>
```

## Getting the Logged-in User and their Roles

To allow non-embedded HMIs, such as the single page applications described in *Loading a Built Web Application as an HMI*, to get the current logged in user and their roles, from SmartSpace version 3.8.1 there is an HMI endpoint:

```
fetch('/SmartSpace/hmiapi/hmipublic/get_user_groups')
   .then(response => response.json())
   .then(data => { myUser = data.User;  myGroups = data.Groups });
```

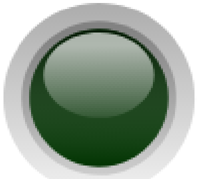
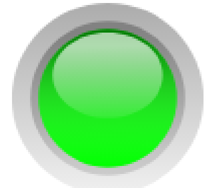
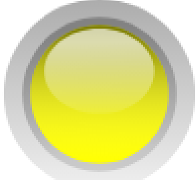
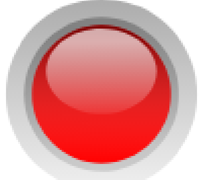
## Built-in Components

The API includes built-in Vue components. All of these must be used inside a Vue element, so either one with a **udm** attribute, or one for which you have constructed a Vue instance manually.

### Indicator component

An indicator is a circular lamp that can be set to a given color and turned on or off, based on bound data.

```
<ubi-indicator size="100"
               :hue="row.Summary[3]"
               :on="row.Summary[4]">
</ubi-indicator>
```

The indicator has the following attributes:

| Attribute | Type | Default | Description |
|---|---|---|---|
| on | Boolean | false | The indicator is on when true. |
| hue | Number | 120 | The hue of the indicator, in HSL color space. For example:<br>0 = red<br>60 = yellow<br>120 = green<br>180 = cyan<br>240 = blue<br>300 = magenta |
| size | String | 32px | The width and height of the indicator. |

## Bar component

A bar is a partly filled horizontal or vertical bar, such as a progress indicator or line, optionally with a label.

```
<ubi-bar width="100" height="30" units="%"
          :ranges="[{from:80,colour:'#faa'}]"
          :value="row.Summary[1]">
</ubi-bar>
```

The bar has the following attributes:

| Attribute | Type | Default | Description |
|---|---|---|---|
| value | Number | 0 | The bar position, relative to min and max. |
| min | Number | 0 | The value corresponding to an empty bar. |
| max | Number | 100 | The value corresponding to a full bar. |
| units | String | | If set, add a label in the center of the bar with the value followed by this units string. |
| width | String | 200px | The width of the bar including background. |
| height | String | 40px | The height of the bar including background. |
| show-label | Boolean | false | Set this attribute to show a label even if units is empty. In JavaScript this is property showLabel. |
| ranges | Array | [ ] | Used to set the color of the bar based on the value. This is an array of ranges, with the first matching range used as the color of the bar. A range has a color (or colour) and an optional from and to value. For example, the following are all valid:<br>{from: 20, to: 40, color: 'blue'}<br>{from: 80, color: '#f00'}<br>{colour: 'yellow'}<br>The latter matches regardless of the value, so only really makes sense as the last item in ranges.<br>A range matches if (from <= value < to). |
| vertical | Boolean | undefined | Normally the bar displays vertically if the width is less than the height. This attribute can be used to override the bar direction regardless of the dimensions. |

You can use CSS to style elements of the bar such as the background color, font size, and whether to draw borders. To style specifically the inner progress bar, target class “UbiHMI-bar”. For example, to make the top of a vertical bar a solid white gap:

**HTML:**

```
<ubi-bar class="bar" width="30" height="100" show-label
         value="61">
</ubi-bar>
```

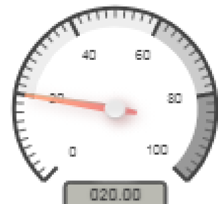
**CSS:**

```
.bar .UbiHMI-bar {
  border-top: 3px solid white;
}
```

## Gauge component

A gauge is a radial or linear control with a needle, scale, and other optional elements such as labels, numerical values, and a background panel rendering. The gauge component wraps the canvas gauges library, and supports all the same options. It has some built-in layouts for easy configuration.

```
<ubi-gauge :value="row.Summary[1]"
           width="130" height="130" radial>
</ubi-gauge>
```



The gauge has the following attributes:

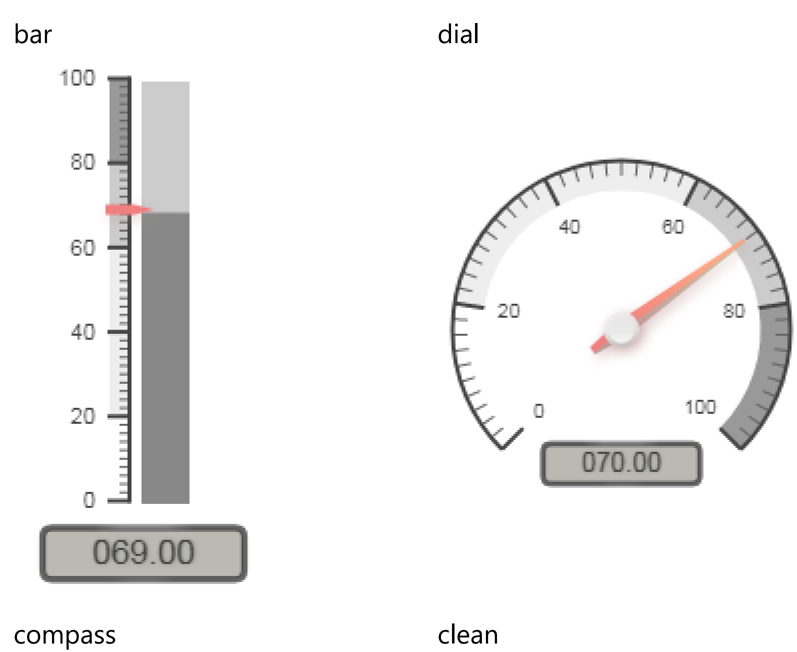
| Attribute | Type | Default | Description |
|---|---|---|---|
| value | Number | 0 | The needle position, relative to min and max. |
| radial | Boolean | false | True to draw a radial gauge, false for linear. |
| width | Number | 200 | The width of the gauge in pixels. |
| height | Number | 200 | The height of the gauge in pixels. |
| layout | String | undefined | Which of the built-in layouts to use – see below. |
| options | Object | { } | Used to set canvas gauge options directly. This gives full control of the gauge drawing. See the complete list of configuration options for canvas gauges online (at time of writing, *https://canvas-gauges.com/documentation/user-guide/configuration*) |

Note: If you set options **maxValue** and **minValue** then you probably also need to set **majorTicks** and **highlights** to match them.

The following layouts are provided:

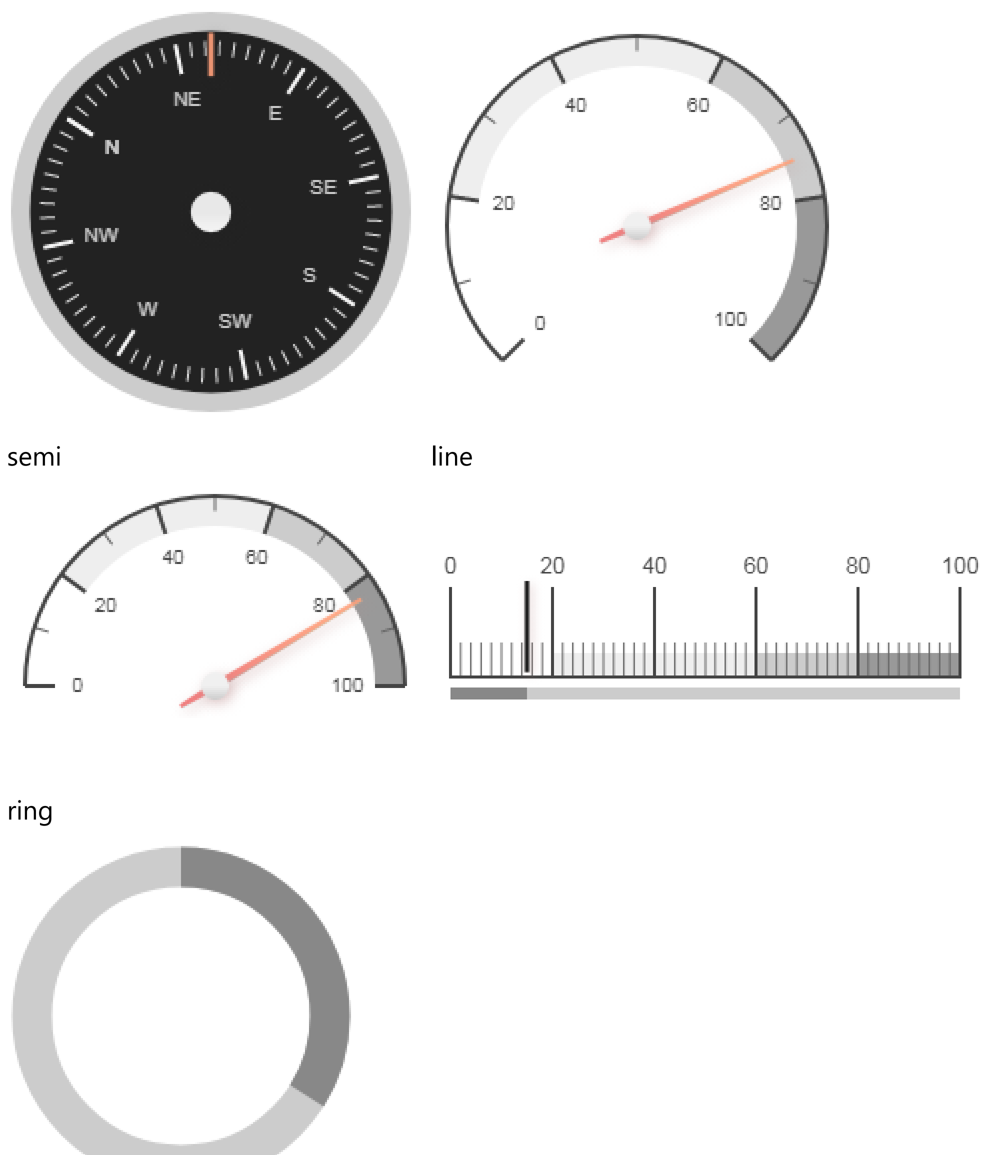
bar

dial




compass

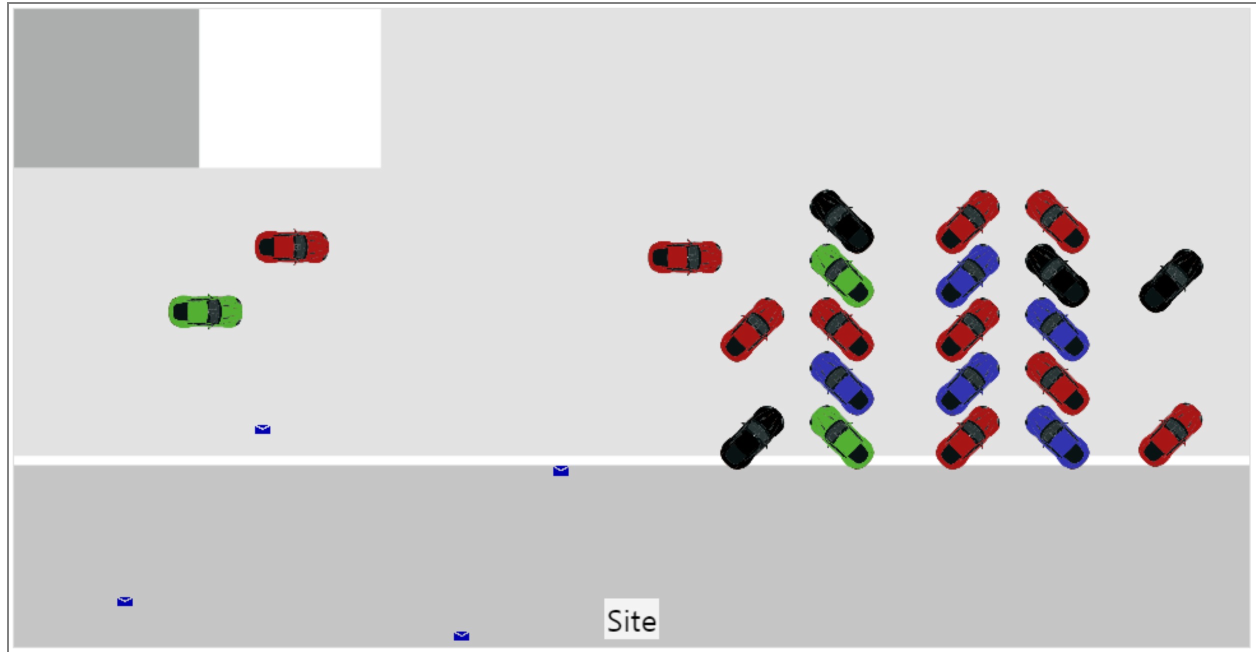
clean




## Map component

The map component encapsulates the SmartSpace web map, with control of displayed elements. It allows the execution of searches, and generates an event when an object is selected.

```
<ubi-map ref="map" width="100%" height="500px"
         search="Products" :value="271"
         @change="selectedObjectsChanged">
</ubi-map>
```




The map has the following attributes:

| Attribute | Type | Default | Description |
| --- | --- | --- | --- |
| search | String | | The name of the search to use. |
| value | String | | The parameter to use for the search input. Only searches with a single input parameter are supported in this release. |
| width | String | 100% | The width of the map. |
| height | String | 500px | The height of the map. |
| details | Boolean | false | If set then show the details panel in the map. |
| alerts | Boolean | false | If set then show the alerts panel in the map. |
| searches | Boolean | false | If set then show the searches panel, allowing other searches to be used. |

| Attribute | Type | Default | Description |
|---|---|---|---|
| details-container | String | | When set to a CSS selector string, the map will render its details panel into the first element matching that string, instead of using an internal panel. Implies details: true. |
| alerts-container | String | | When set to a CSS selector string, the map will render its alerts panel into the first element matching that string, instead of using an internal panel. Implies alerts: true. |
| default-view | Object | | When set to an object, defines the default view to use when no objects are selected. The object format is:<br>{ cx: <center x>, cy: <center y>, w: <width>, h: <height>, theta: <rotation in degrees }<br>For example: <ubi-map :default-view="{cx: 10, cy: 12, w: 20, h:20, theta: 0}"></ubi-map> |
| zoom-scale | Number | 6 | When an object is selected for an auto-zoom search, the map zooms in to the object. This scale determines how close the map zooms, with smaller numbers meaning closer zooms. |

#### The Change Event

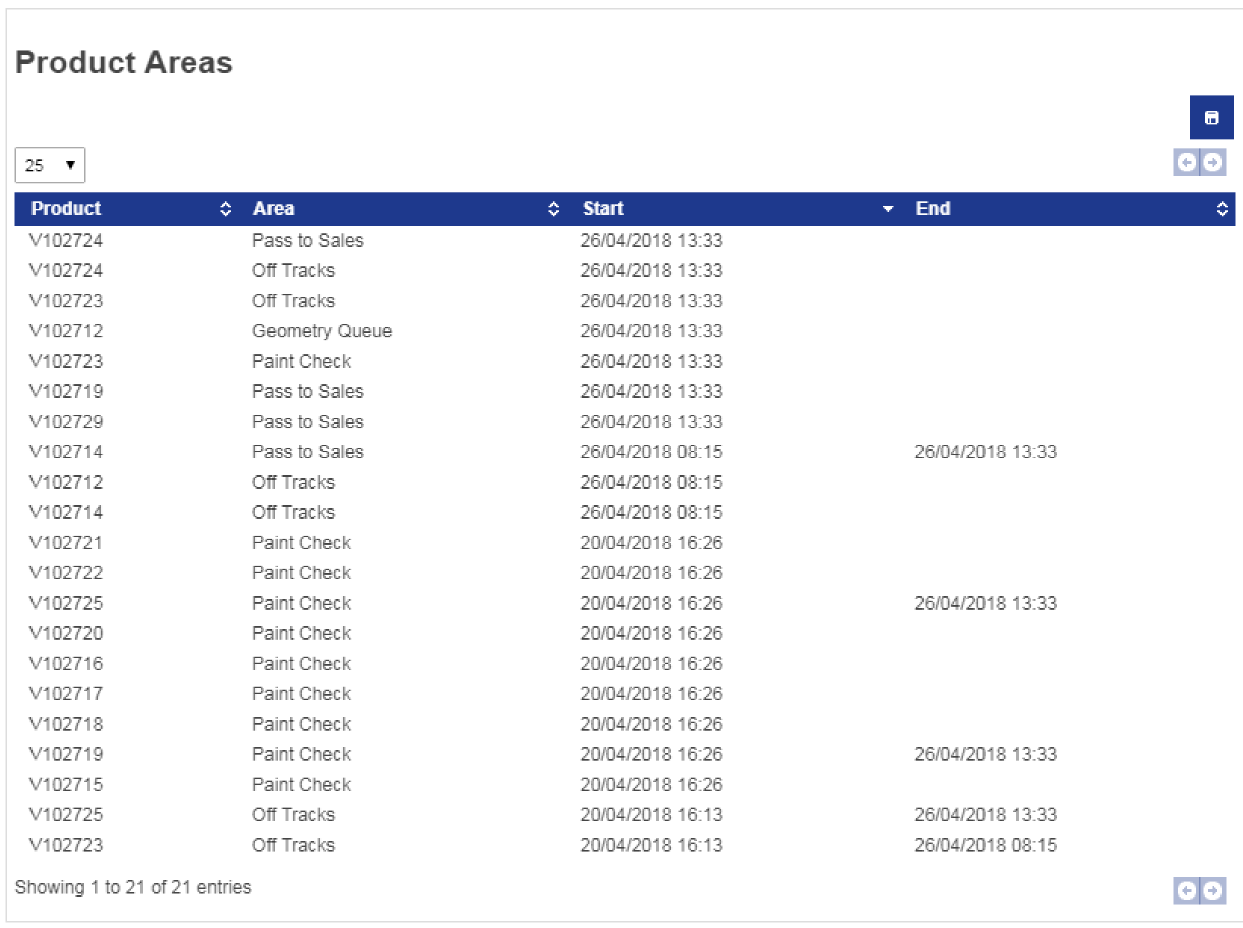
The map emits the "change" event when the user selects an object, either by running a search that returns a single object, or by clicking on one object from those returned by a search. This is a Vue event so doesn't "bubble" up the DOM from the map, and must be handled with **v-on:change** or **@change** on the map component. The event argument is the selected object id, and the event is also passed the map Vue component as a second argument. When the object is deselected, the event is called with the object id undefined.

## Report component

The report component allows a SmartSpace web report to be embedded in an HMI. See Purpose of this guideSmartSpace Reporting where you can find information on configuring reports, and also how to specify that a report is to be used only in an HMI and not within the Reports screen in SmartSpace Web.

Usng the reports component, the report parameters and filters can be set, and an event is generated when a returned row is clicked.

```
<ubi-report ref="report" class="reportClass"
            report="History" no-header
            :filters="{ProductProcessHistory:{
                     ProductName:search,from:'one_week'}}"
            @change="selectedObjectsChanged">
</ubi-report>
```




| Product | Area | Start | End |
|---|---|---|---|
| V102724 | Pass to Sales | 26/04/2018 13:33 | |
| V102724 | Off Tracks | 26/04/2018 13:33 | |
| V102723 | Off Tracks | 26/04/2018 13:33 | |
| V102712 | Geometry Queue | 26/04/2018 13:33 | |
| V102723 | Paint Check | 26/04/2018 13:33 | |
| V102719 | Pass to Sales | 26/04/2018 13:33 | |
| V102729 | Pass to Sales | 26/04/2018 13:33 | |
| V102714 | Pass to Sales | 26/04/2018 08:15 | 26/04/2018 13:33 |
| V102712 | Off Tracks | 26/04/2018 08:15 | |
| V102714 | Off Tracks | 26/04/2018 08:15 | |
| V102721 | Paint Check | 20/04/2018 16:26 | |
| V102722 | Paint Check | 20/04/2018 16:26 | |
| V102725 | Paint Check | 20/04/2018 16:26 | 26/04/2018 13:33 |
| V102720 | Paint Check | 20/04/2018 16:26 | |
| V102716 | Paint Check | 20/04/2018 16:26 | |
| V102717 | Paint Check | 20/04/2018 16:26 | |
| V102718 | Paint Check | 20/04/2018 16:26 | |
| V102719 | Paint Check | 20/04/2018 16:26 | 26/04/2018 13:33 |
| V102715 | Paint Check | 20/04/2018 16:26 | |
| V102725 | Off Tracks | 20/04/2018 16:13 | 26/04/2018 13:33 |
| V102723 | Off Tracks | 20/04/2018 16:13 | 26/04/2018 08:15 |

The report has the following attributes:

| Attribute | Type | Default | Description |
|---|---|---|---|
| report | String | | The name of the report to execute. |
| values | Object | {} | The parameters to apply when executing the report. See below. |
| filters | Object | {} | The filters to apply when executing the report. See below. |
| width | String | 100% | The width of the map. |
| height | String | 500px | The height of the map. |
| no-header | Boolean | false | If set then hide the header of the report, including the title, description, and parameter/filter controls. |

#### Parameters and Filters

The parameters and filters are specified using an object passed as the values and filters attributes.

For parameters, the values object is used. The object can have one property for each parameter of the report. The value of each property is an object { Type: T, Value: V }. For example:

```
<ubi-report ref="report" report="Parts Shortages and Defects" :values="{'BuildNo': {
                      Type: 'String', Value: '225473'}}"></ubi-report>
```

The types supported are:

- String: Value is just a string
- Date: Value is the number of seconds UTC since 1st Jan 1970
- Double: Value is a number
- Bool: Value is true or false

For filters, the filters attribute is used. The keys of the object are source query names used in generating the report. The value for each source is another object with keys corresponding to the column names of that source, and values specifying the filter values. For example:

```
<ubi-report ref="report" report="Vehicle trail" :filters="{'Vehicle Location History':
{
                      BuildNo: '496182', from: 'one_week'}}"></ubi-report>
```

For string columns, the value is just a substring. For date columns, the filter value can be one of the predefined date range strings, or a custom date range object.

The predefined date range strings are:

| | | | | | |
|---|---|---|---|---|---|
| `this_year` | `this_quarter` | `this_month` | `this_week` | `today` | `last_year` |
| `last_quarter` | `last_month` | `last_week` | `yesterday` | `forever` | `two_years` |
| `one_year` | `one_quarter` | `one_month` | `one_week` | `one_day` | |

A custom date range looks like this:

```
{
  From: { Type: 'Time',  Value: <sec since 1/1/1970 UTC> },
  To: { Type: 'Time',  Value: <secs since 1/1/1970 UTC> }
}
```

#### The Change Event

The report emits the "change" event when the user clicks a result row in a report table. This is a Vue event so doesn't "bubble" up the DOM from the map, and must be handled with "v-on:change" or "@change" on the map component.

The event has two arguments – the data for the clicked row, and the report Vue component. The first argument is the row data, and includes all the results from the query used for the table, including results that were not bound to a table column. Thus it may be useful to include the relevant objects in the query results even if they are not displayed to the user as raw identifiers, so that the event handler can easily retrieve the object id for use elsewhere in the HMI when a row is clicked.

As of this release there is no event generated when a report chart is clicked.

# API Examples

In this section we will show some simple examples of HMI definitions, and walk through what is happening in each line.

## Simple Text Example

**HTML:**

This simple example uses only HTML.

```
<div udm='{"products": "Products"}'>
  <div v-cloak v-for="(row, key) in products.rows">
    {{ row.Summary[0] }} is in step {{ row.Summary[1].Name }}
  </div>
</div>
```

The "Products" search is used, and for each resulting row, we create a div containing the product name, and the name of the step the product is in. Note that the second result of the search is an object type, so it has an Id and a Name field – here we use the Name.

The result looks like this:

V102713 is in step
V102719 is in step Pass to Sales
V102715 is in step Paint Check
V102716 is in step Paint Check
V102717 is in step Paint Check
V102718 is in step Paint Check
V102710 is in step Off Tracks
V102711 is in step Geometry Queue
V102712 is in step Geometry Queue
V102714 is in step Off Tracks
V102725 is in step Paint Check
V102726 is in step Geometry Queue
V102727 is in step Off Tracks
V102728 is in step Off Tracks
V102729 is in step Pass to Sales
V102720 is in step Paint Check

## Various Progress Controls

**HTML:**

The HTML uses the “Process Progress” web search, which returns an area name, a progress in percent, the color of an indicator, and whether the indicator should be on or off.

```
<div class="outer">
  <h2>Process area percent progress:</h2>
  <div class="ui-widget-content" id="app-progress"
       udm='{"progress": "Process Progress"}'>
    <div v-cloak class="progress"
         v-for="(row, key) in progress.rows">
      <div class="area-label">{{ row.Summary[0] }}</div>
      <div>
        <ubi-gauge :value="row.Summary[1]" layout="clean">
        </ubi-gauge>
      </div>
      <div>Completion: {{ row.Summary[1] }}%</div>
      <ubi-indicator size="50px"
                     :hue="row.Summary[3]"
                     :on="row.Summary[4]">
      </ubi-indicator>
      <ubi-bar class="bar" width="30" :height="100" show-label
               :ranges="[{from:80,colour:'#faa'}]"
               :value="row.Summary[1]">
      </ubi-bar>
    </div>
  </div>
</div>
```

We use **v-for** to create a div for each area returned which we cloak until bound.

Inside each div we insert:

- The process area name as text
- A gauge bound to the progress percent using the built-in “clean” layout
- A text version of the progress percent
- An indicator based on the hue and on/off value
- A simple bar also based on the progress percent

We also give these elements classes which will be used to lay out and style the HMI in the style sheet.

**CSS:**

```
.outer {
  margin: 10px;
  font-size: 1.2em;
}

.outer > * {
  margin: 10px;
}
```

We style the outer div and the main Vue div to add some margin around the elements.

```
#app-progress {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
}
```

We use the flex layout method (display: flex) to lay out the process area divs in a row, and use flex-wrap: wrap to make the row wrap to another row if the display is not wide enough.

```
.progress {
  text-align: center;
  width: 230px;
}

.progress > * {
  margin: 10px;
}
```

Within each process area ".progress" div, we set a width, and also align text with the center of the div.

```
.area-label {
  font-weight: bold;
}
```

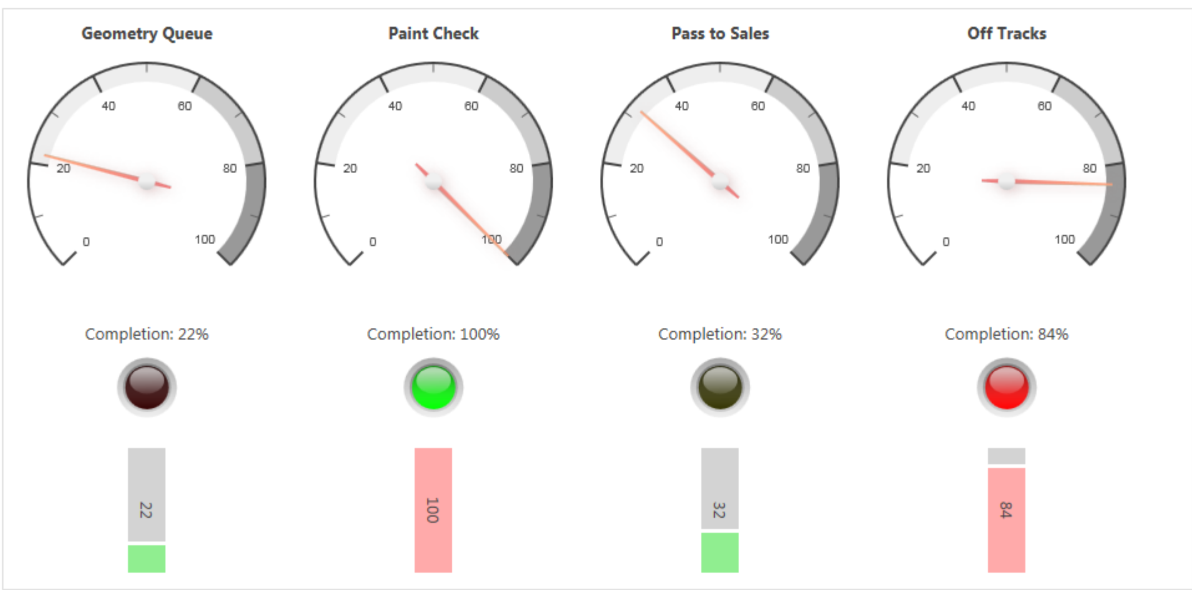
We set the area label to use bold font.

```
.bar .UbiHMI-bar {
  border-top: 3px solid white;
}
```

We style the top margin of the bar component's internal bar to be a white border, giving a gap between the bar and background.

The result looks like this:

**Process area percent progress:**




## Report, Map and Details

This example shows a manual Vue creation, with a report, a map and a details table. We use extra data and the v-model support in Vue, along with event handlers, to hook them together.

**HTML:**

```
<div class="outer">
  <div class="ui-widget-content" id="app-report" v-cloak>
```

We create a div for the app and cloak it.

```
    <input id="searchName"
           placeholder="Search for a product"
           v-model.lazy="search">
```

Inside, we have a text input with a model bound using the lazy modifier to the data “search”. The lazy modifier means that the model is only going to be set when the user presses Return or Tab or otherwise moves focus away from the input control.

```
    <div class="row">
      <ubi-report ref="report" class="reportClass"
                  report="History" no-header
                  :filters="{ProductProcessHistory:{
                     ProductName:search,from:'one_week'}}"
                  @change="selectedObjectChanged">
      </ubi-report>
```

We create a row container, and in it put a report using the “History” report with a filter bound to use the data “search” as the ProductName, and to use a fixed duration of one week. Change events are handled by the method **selectedObjectChanged**.

```
      <div>
        <ubi-map class="map" ref="map" height="400" width="400"
                 search="Products" :value="selectedName">
        </ubi-map>

        <div class="details">
          <h2>Details</h2>
          <table>
            <tr v-for="row in products.details"
                :data-title="row.Title">
              <td> {{ row.Title }} </td>
              <td> {{ convertValue(row)  }} </td>
            </tr>
          </table>
        </div>
      </div>
    </div>
  </div>
</div>
```

We then add a new div containing a map running the “Products” search with value bound to date “selectedName”. The div also contains a table with a row for each detailed result in the products web search. The rows show the title of the row, and use a converter function ”convertValue” to turn the row value into a suitable string.

**Script:**

```
$(function () {
  var vm = new Vue({
    el: '#app-report',
    mixins: [ UbiHMI.mixin({ products: "Products" }) ],
    data: { search: '', selectedName: '' },
    methods: {
      selectedObjectChanged: function (selected, vm)
      {
        var s = undefined;
        if (selected) {
            s = selected.Product;
            this.selectedName = selected.ProductName;
        }
        this.products.selected = s;
      }
    }
  });
});
```

The script uses the jquery **document.ready** shortcut **$(function () { … })** to execute the contents when the DOM has been fully loaded, so all the class libraries and APIs are available.

A Vue object is created and attached to the app-report element. We use the **UbiHMI.mixin** to generate a web search data binding for the "Products" web search, as data "products". We then add the two extra data properties used in the HTML: "search" and "selectedName".

We implement the method **selectedObjectChanged** that handles the change event in the report. If the selected row data is defined, we get the result "Product" which is the project object identifier in this report's query. We also set the **selectedName** data property to be the "ProductName" result from the query. This causes the map to execute its search using this product name.

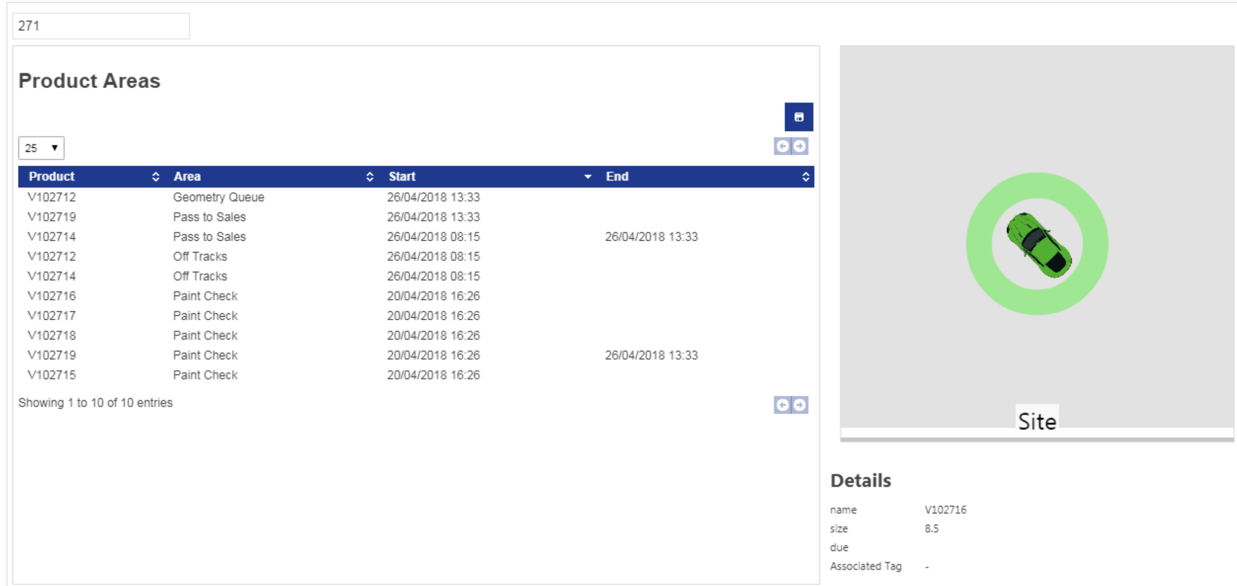
We then set **products.selected** to be the object identifier that was

clicked. This causes the products detail search to be executed, which in turn will populate the details table.

Finally we implement the **convertValue** function used in the details table. Here we decode the type of each detail result and return something appropriate to render to a string in the table.

```
function convertValue(v)
{
  if (v.hasOwnProperty('String'))
    return v.String;
  else if (v.hasOwnProperty('Number'))
    return (v.Number === null) ? '' : new Number(v.Number);
  else if (v.hasOwnProperty('Boolean'))
    return v.Boolean ? '✓': '✗';
  else if (v.hasOwnProperty('Date'))
    return v.Date ? (new Date(v.Date)).toLocaleString() : '';
  else
    return "-";
}
```

The result is a reactive report that can be searched from the input box, and when a row is selected the corresponding product is shown in the map, and its details displayed below.




## Styling Bars

This example shows some advanced styling for bars using CSS.

```
<div class="app" id="app-progress"
     udm='{"progress": "Process Progress"}'>
  <div v-cloak class="progress"
       v-for="(row, key) in progress.rows">
    <div class="area-label">{{ row.Summary[0] }}</div>
    <ubi-bar class="bar" width="200" height="30" show-label
             :ranges="[{from:80,colour:'#b44'}, {colour:'#4b4'}]"
             :value="row.Summary[1]">
    </ubi-bar>
  </div>
</div>
```

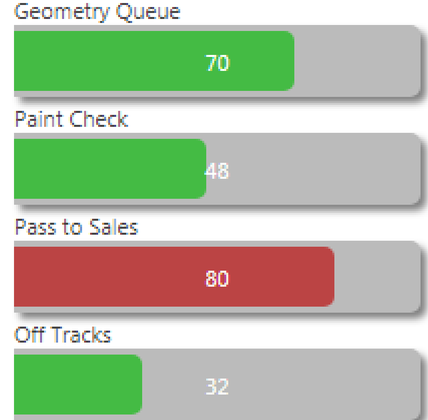
We create some ubi-bars based on a web search, give them a class “bar”, and set them to use darker colors than normal.

```
.bar {
  border-radius: 0px 5px 5px 0px;
  padding: 3px 3px 3px 0;
  box-shadow: 3px 3px 5px gray;
  background: #bbb;
  color: white;
}

.bar .UbiHMI-bar {
  border-radius: 0px 5px 5px 0px;
}
```

Then we use the following style to give a rounded right corner, add some shadow, padding, and make the text label white to stand out against the darker colors. Note the CSS for borders and padding supports separate values for “top right bottom left”.

The result looks like this:



## Progress Rings

The ring layout of the ubi-gauge provides a very clean and simple indicator of progress. The following example shows some layout options, including how to place a simple label over the middle of the gauge, and how to bind the color of the ring depending on the value. The HTML looks like this:

```
<div class="app" id="app-progress"
     udm='{"progress": "Process Progress"}'>
  <div v-cloak class="progress"
       v-for="(row, key) in progress.rows">
    <div class="area-label">{{ row.Summary[0] }}</div>
    <div>
      <ubi-gauge class="dial" width="200" height="200" radial
                 layout="ring"
                 :options="{ colorBarProgress: (row.Summary[1]<80 ?
'green': 'red')}"
                 :value="row.Summary[1]">
      </ubi-gauge>
      <div class="progress-label">{{ row.Summary[1] }}</div>
    </div>
  </div>
</div>
```

Here we use the **colorBarProgress** option from canvas gauges library to set the bar color to green if the value is less the 80, and red otherwise. We wrap the gauge and the progress label in a single div, as we intend to put the label on top of the gauge using CSS.

The CSS is relatively complex:

```
.app {
  display: flex;
  flex-wrap: wrap;
  background: #f5f2f0;
  justify-content: center;
  padding: 10px;
}
```

The outer application div is displayed using the flex layout so that the arrangement wraps according to available screen width. The content of each row is justified by centering. Some padding is added around the screen edges, and a subtle gray is used to make the progress panels themselves stand out.

```
.progress
{
  background: white;
  margin: 15px;
  padding: 10px;
  box-shadow: 2px 2px 3px grey;
}
```

The progress panel background is white, and some space is added around and inside to avoid the layout appearing too cramped. A narrow box shadow is added so they stand out from the background.

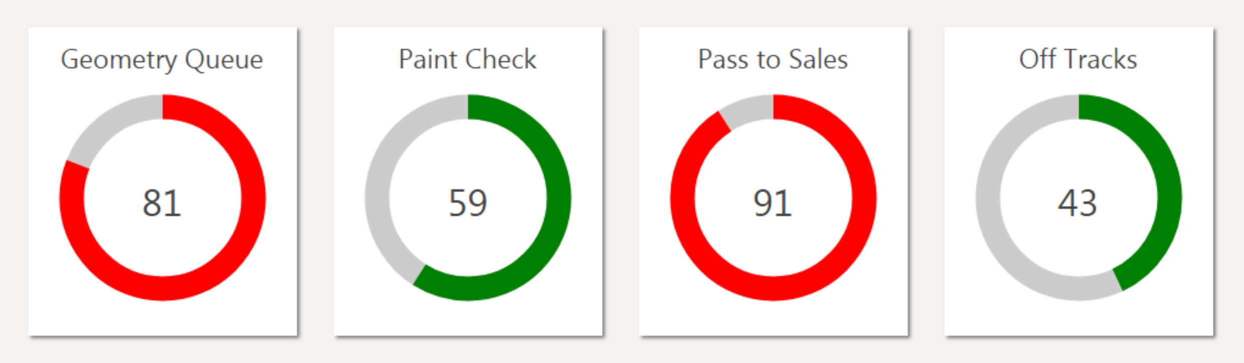
```
.area-label {
  font-size: 2em;
  text-align: center;
}
```

The area label uses a large font and is centered within the progress panel.

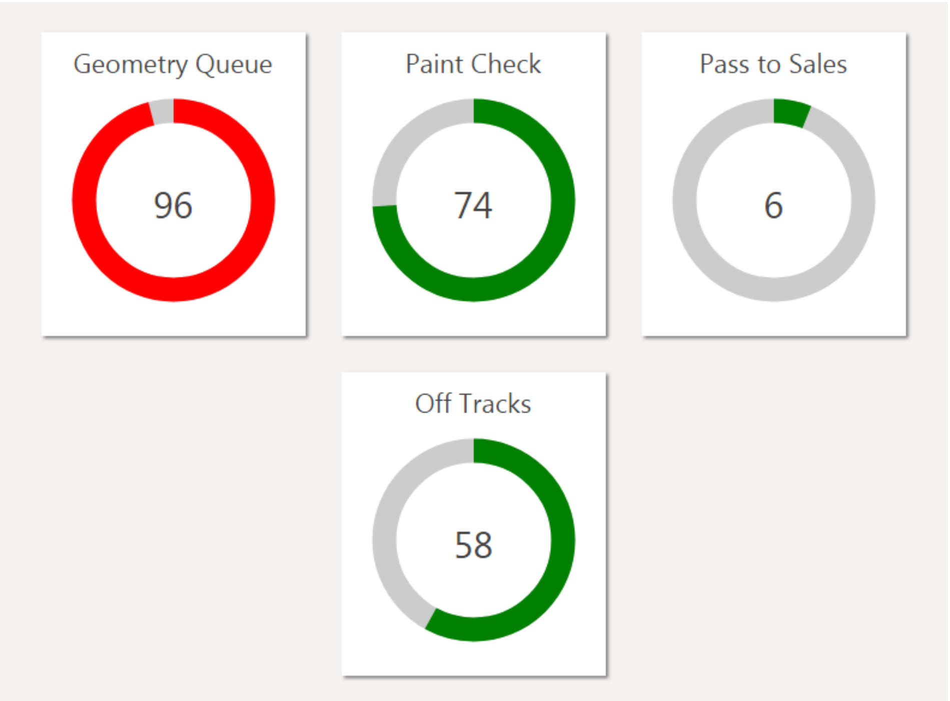
```
.progress-label {
  position: relative;
  font-size: 2.8em;
  width: 5em;
  text-align: center;
  margin: 0 auto;
  line-height: 0;
  top: -100px;
}
```

The progress label uses relative positioning, so it is placed relative to the previous object (the end of the gauge). We give it a width and specify that the text should be center aligned in that width. We then give it an automatic left margin, which moves it into the center of its parent div horizontally. To place it vertically, we set a line-height of zero, and offset the top by half the height of the previous gauge component. This causes the vertical center of the text to be on the vertical center of the gauge.

The result looks like this:




If the width of the browser is not enough to show four progress panels across, it wraps and centers like this:



## JavaScript Libraries

The default API includes the following JavaScript libraries. The versions selected may not be the most recent, but are known to work across the set of modern browsers supported by the rest of SmartSpace Web.

Supported browsers for use with SmartSpace's web-enabled features are recent versions of:

- Microsoft Edge
- Chrome
- Chrome for Android
- iOS Safari

Additionally, recent versions of the following browsers work but are not explicitly supported:

- Firefox
- Opera
- Safari

Internet Explorer 11 is deprecated and SmartSpace's web-enabled features are not guaranteed to work with this browser.

The versions of these libraries are subject to upgrade with subsequent releases of SmartSpace:

- Vue.js: 2.6.11
- Vue-resource.js: 1.5.1
- jquery: 3.5.1
- jquery-ui: 1.13.2
- datatables: 1.9.4