Developing Storefront Classic Widgets for Oracle Commerce

ORACLE®

Contents

10   Manage Storefront Event Notification

Index

1
About This Guide

This guide is intended for developers who want to use the Storefront Classic framework to create widgets for Oracle Commerce storefronts.

It describes how to customize extensions, widgets, elements, and other site settings, and provides guidance on topics such as directory and file structure, configuration file syntax, and localization techniques.

For information about developing widgets for specific features, such as for integrating with an external pricing system or a web checkout system, see Extending Oracle Commerce. For information about using the Open Storefront Framework to create widgets and other storefront components, see Developing Open Storefront Framework Applications for Oracle Commerce.

2
Create an Extension

An extension encapsulates entities and assets that can enhance your Commerce implementation with additional, custom functionality.

This section describes how to create and upload extensions.

Understand extensions

Extensions are pieces of code that you can configure to extend your environment.

Extensions can contain one or more of the following:

- Widgets
  A widget provides a unit of UI functionality that can be deployed on one or more pages of your web store. Widgets are able to display content to visitors or execute specific functions. They provide custom HTML and/or JavaScript code and, optionally, several other types of auxiliary data for styling, localization, and component re-use.

- Elements
  To provide more control over its constituent parts, widgets can be broken up into elements. Each element represents one part of the overall structure of the widget and they can be configured as drag-and-drop sub-components, allowing for finer control over their location in a page layout. For example, the Header widget contains the following elements: Cart Link, Language, Links, Login/registration, Logo, Rich Text and Search. An element can be defined as part of a specific widget or as a stand-alone element that may be used by multiple widgets.

- Stacks
  Page layouts are made up of regions, into which widgets are placed. All widgets placed within a region take up the full widget of the region and are rendered vertically within the region, one after the other. Stacks allow you exercise further control over the placement of widgets within a given region, making it possible to create a stepped progression through the widgets contained in the region.

- Payment Gateways
  Commerce provides support for a number of payment gateways as built-in integrations. In addition, you can use extensions to create custom integrations with other payment gateways. The integrations you create appear as options on the Payment Gateways tab of the Payment Processing page in the administrative console. See Configure Payment Processing for more information.

- Site Settings
  Site settings allow you to define a set of global configuration parameters. These parameters are made available to page layout designers in the administration interface and all widgets have access to their settings. Site settings allow you to define a single setting that controls a feature across multiple widgets.

- Application-level JavaScript modules
  Application-level JavaScript are loaded as part of the main module, before any endpoints have been fired and before any widgets have been loaded. As such, they can be

referenced as a dependency in any widget, allowing you to build reusable modules that can be shared among widgets and elements.

Create an extension ID

You must create an extension ID that you use when you define its meta-data.

In order to upload an extension into Commerce, you must generate an ID for the extension and then use that ID when you define the extension's meta-data. To create an extension ID perform the following steps:

1. Log into Commerce.

2. Click the **Settings** icon.

3. Click **Extensions**, then click the **Developer** tab.

4. Click the **Generate ID** button to generate an extension ID.

   You are prompted to name the extension.

5. Enter a name for the extension and click **Save**.

6. Your extension ID is now generated and must be used in the extension's `ext.json` file. For further details, refer to the next section, Create the extension structure.

Create the extension structure

Extensions require a specific structure configuration.

Once you have a new unique ID for your extension, you can start to develop it by creating the folder structure, as shown in the example below. Make sure your extension uses a name that is unique within your Commerce environment; for example, do not give your extension the same name as a default widget.

Note: If you do not include JavaScript, Less styles, or Locales, you can omit those directories.

```
<extension-name> : The root folder of your extension, must be unique
          ext.json
          <additional directories to define widgets, elements,
           payment gateways, site settings>/
```

Define extension meta-data in ext.json

Each extension provides meta-data describing the author/developer of the extension and other information related to its creation. This meta-data is contained in a manifest file called `ext.json`. Without the manifest file, the extension cannot be loaded. The `ext.json` file should include the following:

- `extensionID`: Unique ID of the extension. Note that this ID must match the value generated on the Extensions page in the administrative console. See Use developer tools to customize your store for more information.

- `developerID`: The extension author's unique developer ID.

- `createdBy`: The name of creator (company or individual).

- `version`: The version of the extension, expressed as a single whole number.

- `timeCreated`: When the extension was created. iso-8601 is the recommended format; however, an informal time stamp is also acceptable.

- `translations`: An array that provides translations for both the extension name and description. These properties are displayed on the Extensions page that is accessible from the Settings page in Commerce. For example, the English translation for the name of the extension in the illustration below is "Visit Counter Widgets" and the description is "Provides visitor counting related functionality".

An example of an `ext.json` file that includes translations for English and French is shown below. Note that the translations property has three sub-properties, language, name, and description. The language property can be either a two-letter language code (for example, `en`) or a two-letter language code and a two-letter country code with an underscore in between (for example, `en_US`). `ISO 639-1` defines the two-letter language codes. `ISO 3166-1 alpha-2` defines the two-letter country codes.

```
{ "extensionID": "f3acef8e-375a-11e4-9c85-ebc5b52923a8",
          "developerID": "987654",
          "createdBy": "Company name",
          "version": 1,
          "timeCreated": "2016-09-08",
          "translations" : [
            {
              "language" : "en_US",
              "name":"Extension label in English",
              "description":"Extension description in English"
            },
            {
              "language" : "fr_CA",
              "name":"Extension label in French",
              "description":" Extension description in French"
            }
          ]
        }
```

Add functionality to the extension

The extension directory contains additional directories, at the same level as the `ext.json` file, that define the functionality you want to upload. For more information on creating these directories and their contents, see the following:

- For information on creating widgets, elements, and site settings, refer to the remainder of this guide. Also refer to this guide for information on uploading application-level JavaScript modules.

- For information on creating payment gateways,see Configure Payment Processing.

Create and load the extension bundle

You must create a bundle that allows you to upload the extension.

When you have finished developing or changing your extension contents, zip up all the files within your *extension-name* directory. This is the file you upload to Commerce to make the

extension available for use. On a Mac or Linux-based machine, you can do this with the following command inside the *extension-name* directory:

```
zip -r extensionName.zip ./*
```

After creating the ZIP file, load it using the procedure below.

To load an extension, perform the following steps:

1. Log into Commerce.

2. Click the **Settings** menu icon.

3. Click **Extensions**.

4. Click the **Upload Extension** button and select the extension zip file from your local file system.

The system now starts the upload and validation process. Any problems identified during the validation checks are displayed in a warning message.

3

Create a Widget

A widget provides a unit of UI functionality that can be deployed on one or more pages of your web store.

This chapter introduces widgets and the files that configure them.

Understand widgets

Widgets are made up of a set of source files and resources.

A number of things provide the widget with its functionality and include the following:

- Templates: Display templates for showing content using knockout.js data bindings.
- JavaScript: View model per widget (optional).
- CSS: Styles for the widget. Can be pure CSS or Less (see http://www.lesscss.org).
- Locale resources: Translation resources per locale for the widget.
- Images: Images used for the widget, referenced via the widget asset mappings.

Widgets also consist of auxiliary files that contain help, configuration, and meta-data describing the widget. They include the following:

- Meta-data: The meta-data that describes the widget.
- Elements: Small pieces or fragments of functionality for a widget, including a display template snippet and a view model.
- Configurable widget settings: A mechanism for providing configurable widget settings that the page layout designer can use to customize the widget's behavior on the Design page, for example, limiting the number of products to display on a Related Products widget.
- Configuration: Default configuration can be delivered with the widget.

Download widget source code

The following sections provide details on how to create a new widget type from scratch. However, you can also download the source code for an existing widget type and use it as a starting point.

To download widget source code perform the following steps:

1. On the **Design** page, select a widget from the **Widget Templates** list.
2. Click the **Download Source** button and, when prompted, save the widget's ZIP file.

Create the widget structure

Widgets are added to an extension's structure in a `/widget` directory that is at the same level as the extension's `ext.json` file.

When you create a new widget, it is important to ensure that the name is less than 50 characters. An error will occur if the name contains more than 50 characters.

Each widget should have its own child directory in the /widget directory. The following example shows the directories and files that can be included for a widget:

```
<extension-name> : the root folder of your extension
          ext.json
          widget/
              <widget-type>/
                widget.json
                templates/
                  display.template
                js/
                  <widget-type>.js
                less
                  widget.less
                locales/
                  <locale code, for example, en or en_US>/
                      ns.<widget-type>.json
                  <other locale codes>/
                      ns.<widget-type>.json
                images/
```

If your widget does not involve the creation of user interface elements or require custom JavaScript, the widget structure can omit a number of directories and files used for these purposes. The following shows the minimum set of files and directories needed for a widget to pass validation on upload:

```
<extension-name>
      ext.json
      widget/
          <widget-type>/
              widget.json
```

Define widget meta-data in widget.json

Much like the ext.json file defines meta-data for an extension, a widget.json file defines meta-data for a widget.

An example of a widget.json file is provided below:

```
{
   "version": 1,
   "global": false,
   "javascript": "<widget-type-js>",
   "i18nresources": "<widget-type>",
   "availableToAllPages": true,
   "jsEditable": true,
   "config": {
   }
   "translations" : [
     {
```

```
      "language": "en",
      "name": "Name in English"
    },
    {
      "language": "de",
      "name": "Name in German"
    }
  ]
}
```

The following list describes all the properties that you might choose to include in a `widget.json` file.

- `javascript`: The name of the widget's main JavaScript file, to which .js is appended, to load in the storefront. The convention is to use the widget-type as the JavaScript module name without the .js suffix. This property is required if the widget includes one or more JavaScript files and should refer to the main JavaScript file for the widget. Other JavaScript files can be defined as dependencies in the main JavaScript file. See Include multiple JavaScript files for more details.

- `jsEditable`: A flag that determines whether it is possible to edit the widget's JavaScript code within the Design page. Defaults to false.

- `i18nresources`: The namespace name for the resources of this widget, to which `ns` is prepended and `.json` is appended. The convention is to use the widget-type for this property, creating a file with a name like `ns.widget-type.json`. This property is required if the widget has resource files. See Localize a widget for more information.

- `imports`: By default, widgets have access to data and functions contained in the common view models (user, cart, order and so on). However, in order for a widget to have access to page-specific view models, those view models must be explicitly defined in the `imports` property. The possible values that are acceptable for the imports property depends on the type of page the widget will be placed on. See Acceptable values for the imports property for detailed information.

- `availableToAllPages`: Set this property to true to allow the widget to be placed on all page types; the widget will appear in the Component library for all pages. Omit this property altogether to restrict placement of the widget to the page types defined in the `pageTypes` property. Also, omit this property for global widgets (see the global property below). Note that this property and the `pageTypes` property are mutually exclusive but one of them must be set.

- `pageTypes`: Defines which page types the widget can be placed on; the widget will appear in the Component library for the specified pages. If you use this property, you must omit the `availableToAllPages` property. Note that this property and the `availableToAllPages` property are mutually exclusive but one of them must be set.
  Available page types include the following:

    - product

    - category

    - home

    - cart

    - checkout

    - confirmation

- article

- error

- searchResults

- noSearchResults

The widget will appear in the Component library for the page types you specify.

- `translations`: An array that provides translations for the widget name. This name is displayed in the Components library panel that you use to add a widget to a page layout. Note that either the translations property or the name property, described below, is required.
The translations property has two sub-properties, language and name. The language property can be either a two-letter language code (for example, en) or a two-letter language code and a two-letter country code with an underscore in between (for example, en_US). ISO 639-1 defines the two-letter language codes. ISO 3166-1 alpha-2 defines the two-letter country codes.

- `version`: Specifies the version of the widget, used to ensure the right version of a widget is used in the storefront. See Understand widget versioning for more information. Defaults to 1.

- `name`: If your storefront only uses one language, meaning you do not need multiple translations for the name of the component, you can choose to use the name property instead of the translations property. Note that one of the two properties is required.

- `global`: Defines the widget as a global widget. A global widget does not include a display template. It is automatically added to all pages but, due to the lack of template, it is excluded from template rendering. Global widgets are useful for tasks like logging web analytics or loading JavaScript libraries. Defaults to false.

- `minWidth`: The minimum width that this widget will fit into it. The system will check if the widget will fit in the region in Grid View.

- `hiddenFromSiteStudio`: A Boolean that determines whether the widget's initial state is to be hidden or not in the Component library. Set to false by default. See Customize your store layouts for details on showing or hiding widgets in the Component library.

Acceptable values for the imports property

When defining a widget, one of the properties you may choose to set in the `widget.json` file is the imports property.

By default, widgets have access to data and functions contained in the common view models (user, cart, site and so on). However, in order for a widget to have access to page-specific view models, those view models must be explicitly defined in the imports property. The possible values that are acceptable for the imports property depends on the type of page the widget will be placed on. To determine which page-specific view models your widget can have access to, determine the page type the widget will be placed on and then review the acceptable values for that page type in the following sections.

Note that some page types have no page-specific view models. These include home, order history, article, error, shopper wish list profile, search results, and no search results.

Category

Acceptable imports values include the following:

- `category`
- `categoryId`
- `dimensionId`
- `productTypes`

Product

Acceptable imports values include the following:

- `product`
- `productTypes`
- `productVariantOptions`

Cart

Acceptable imports values include the following:

- `defaultShippingCountry`
- `order`
- `payment`
- `shippingCountries`
- `shippingCountriesPriceListGroup`
- `shippingmethods`

Checkout

Acceptable imports values include the following:

- `billingCountries`
- `defaultBillingCountry`
- `defaultShippingCountry`
- `order`
- `payment`
- `paymentauthorization`
- `shippingCountries`
- `shippingCountriesPriceListGroup`
- `shippingmethods`

Confirmation

Acceptable imports values include the following:

- `confirmation`

- `defaultShippingCountry`

- `shippingCountries`

- `shippingCountriesPriceListGroup`

Order Details

Acceptable imports values include the following:

- `defaultShippingCountry`

- `orderDetails`

- `shippingCountries`

- `shippingCountriesPriceListGroup`

Shopper Profile

Acceptable imports values include the following:

- `defaultShippingCountry`

- `shippingCountries`

- `shippingCountriesPriceListGroup`

Wish List

Acceptable imports values include the following:

- `space`

Create the widget template file

With the exception of global widgets, all widgets require a template file, called `display.template`, in the `/widget/<widget-type>/templates` directory.

The file takes the following structure:

```
<extension-name> : the root folder of your extension
    ext.json
    widget/
        <widget-type>/
            widget.json
            templates/
                display.template
```

The template is rendered within the context of the widget and should be written as straight HTML with no surrounding script tag. All knockout bindings and behavior are available in the HTML template code.

Note: Additional templates are required if you want to fragment your widget into elements. See Fragment a widget into elements for more details.

The following is an example of the Loyalty Payment widget's template file:

```
<!-- ko if: ($data.paymentsContainer().isLoyaltyEnabled() &&
            ((CCRestClient.profileType == CCConstants.PROFILE_TYPE_AGENT &&
            $data.user().selectedPriceListGroup().currency &&
            $data.user().selectedPriceListGroup().currency.currencyType ==
            CCConstants.LOYALTY_POINTS_PAYMENT_TYPE)
            || (CCRestClient.profileType != CCConstants.PROFILE_TYPE_AGENT
&&
            $data.cart().currency && $data.cart().currency.currencyType ==
            CCConstants.LOYALTY_POINTS_PAYMENT_TYPE))) -->
  <div id="loyaltyPayment">
    <!-- oc layout: panels -->
      <div class="oc-panel" data-oc-id="panel-1">
        <!-- oc section: select-redeem-points -->
          <div data-bind="element: 'select-redeem-points'"></div>
        <!-- /oc -->
      </div>
    <!-- /oc -->
  </div>
<!-- /ko -->
```

Create custom widget JavaScript

You can customize JavaScript to add functionality for your widgets.

To add custom JavaScript to a widget you must create a JavaScript file under the *extension-name*/widget/*widget-type*/js directory. The name of the JavaScript file must match the value of the JavaScript property in the widget.json file, minus the .js extension. The convention is to use the widget-type as the JavaScript file name, without the .js suffix.

Note: As an easier coding option, there is JavaScript Code Layering feature that lets you extend the JavaScript of an Oracle Commerce provided widget with your own custom JavaScript. With the JavaScript Code Layering User Interface feature you can open an additional user interface that lets you layer custom JavaScript on top of the provided widget and which then has the benefit of staying on the provided widget. For more details on this feature, see Modify Your Storefront Using Code Editing Tools and Use the JavaScript Code Layering User Interface feature.

Custom JavaScript for a widget assumes that the file will perform some custom logic and return an object with extensions to the widget's view model. The JavaScript file should implement the following format using RequireJS

Note: The module must be defined anonymously, in other words, have no package name defined in the module, as shown below.

```
define(
   // Dependencies
   ['jquery', 'knockout'],
   // Module Implementation
   function($,ko) {
              // We recommend enabling strict checking mode
        'use strict';
```

```
        // Private variables and functions can be defined here...
        var SOME_CONSTANT = 1024;
        var privateMethod = function () {
          // ...
        };
        return {
         // Widget JS
                // Some member variables...
                textInput: ko.observable(),
                // Some public methods...
                doSomethingWithInput: function () {
           //...

                }
     }
 });
```

The define statement above can be modified to include widget-specific libraries or other JavaScript files, if required. When a widget is instantiated all properties returned from the JavaScript file specified will be copied into that instance of the widget's view model. This allows you to define properties, make Web API calls, or handle UI events.

Understand `this`

Using this in the custom JavaScript is suitable when you refer to the widget itself, but be careful of any callback methods where this may refer to a different context.

Include multiple JavaScript files

If your widget requires multiple JavaScript files, then any additional JavaScript files can be loaded through the dependencies in the widget module's define statement.

To derive the path of the dependency, use the path `js/` to reference the widget's JavaScript folder followed by the path to the dependency but omitting the .js extension. The following example is of widget called `myWidget` includes the following JavaScript files:

```
myWidget/
        js/
            file1.js
            file2.js
            file3.js
            myWidget.js
```

And `myWidget.js` is the main JavaScript file for the widget and it is dependent on the other three JavaScript files, then the required dependencies in `myWidget.js` would look like the following example:

```
// Dependencies
 ['js/file1','js/file2','js/file3'],
```

Running custom logic upon widget instantiation

If you must run custom logic when a widget is instantiated, then add an `onLoad()` function to the widget JavaScript's return object. The following is an example:

```
onLoad: function(widget) {
   //onLoad code here.
 }
```

`onLoad()` will run once the widget has finished loading and is populated with the necessary data. This is the main access point to configure data for the widget after its properties have been loaded and the system is ready to display the widget. As the `onLoad()` function is only called the first time the widget is instantiated, when returning to the same "page", the widget does not need to be re-instantiated, so `onLoad()` is not called again.

Running custom logic each time a widget appears on a page

If you require some logic to run each time the widget appears on the page, add the `beforeAppear()` function to the widget JavaScript's return object. The following is an example:

```
beforeAppear: function(page) {
          // Code to run before showing the widget here. }
```

`beforeAppear()` is run once any re-population of mapping data has occurred. This can be useful when a Web API call is required every time the widget is shown, or, some other functionality required every time the widget is shown.

Specify a function runs when an HTML element is being rendered

`onRender` is a custom Commerce binding that tells a function to run when an HTML element is being rendered on the page. The function is called in the current knockout context (typically bound to the widget, but certain knockout constructs, such as for each, may alter the binding context). For example, when the following div tag is rendered, the `addEventHandlersForAnchorClick` function is called:

```
<div id="CC-customerProfile" data-bind="onRender:
addEventHandlersForAnchorClick">
          <!-- ... -->
 </div>
```

Rely on mapping for a property vs. initializing it via JavaScript

Most of a custom widget's data will come from the JSON. This data does not need to be explicitly defined in the widget's view model; knockout mapping will automatically create it.

For any data that doesn't come through JSON, the observable should be explicitly defined in custom JavaScript. Otherwise, an error will be thrown if a template tries to render the property while it's undefined. For example, imagine a `productName` widget that is configured with a `productId` and uses that `productId` to look up a product in order to display its name. In this case, the `productId` can be a property defined via `koMapping` as it should be returned by the JSON data. The product's name, on the other hand, is expected to be populated when the product look up completes, so it needs to be defined in the JavaScript as `productName`:

`ko.observable()`. Otherwise, when the template is rendered, an error will be thrown because `productName` is not a valid binding (it would be undefined).

Configure a widget's style

You can customize your widgets by configuring the styles they use.

CSS specific to a widget is contained in the `/widget/`*`widget-type`*`/less` directory. The following is an example:

```
<extension-name> : the root folder of your extension
          ext.json
          widget/
            <widget-type>/
                  widget.json
                  less/
                       widget.less
```

These files are always named `widget.less` and any CSS can be added here. Styling across the storefront is built using Less. A Less file, like `widget.less`, can define style using the Less language or native CSS. Less files are compiled to make one CSS file.

For the storefront, there are some Bootstrap Less files, common Less files, and widget-specific Less files, which are compiled together into `storefront.css`. The overall styling of the storefront is known as a Theme. Commerce provides tools to manage Themes and allows merchants to change the storefront styling. You can use the Theme Manager, or the Theme CSS to customize variables. See Customize your theme, or Modify theme code. For information on working with Bootstrap variables, refer to your Bootstrap documentation.

It is important that any style overridden within a widget Less file only applies to that widget, and does not change the style across the storefront. One way to achieve this is to format the widget's styles using the Less nested format, for example:

```
.myWidgetClass {
    .myClass {
      color: red;
    }
    .otherClass {
      color: blue;
    }
 }
```

In this case `myWidgetClass` is a class applied at the top-level of the widget's template, `myClass` is a class created for this widget and `otherClass` is a class that already exists but the styles need to be modified slightly for this widget.

When this Less file is compiled, it produces CSS in the format shown below, which ensures the changes to the styles for `otherClass` are only applied within this widget. The following is an example:

```
.myWidgetClass .myClass {
    color: red;
```

```
  }
  .myWidgetClass .otherClass {
    color: blue;
  }
```

> **Note:** If you design a custom style as part of an extension, the Less style may not be compiled immediately after uploading the extension package. If this happens, open the widget's style in the code editor (on the Design page), make a superficial edit, and then save. This process forces compilation of the style.

Localize a widget

You can customize widgets to recognize the language of the browser that your customer is using.

Any text in a widget that does not come from a remote API call can be defined in a resource bundle so that it can be localized. **Note:** You can include localized resources in your widgets/ elements by including a `/locales` directory. However, this is not necessary. You can just as easily hardcode strings inside the HTML templates or JavaScript source files.

Use resource files

The resources reside in the widget `/locales` directory, as shown in the following example:

```
<extension-name> : the root folder of your extension
      ext.json
      widget/
          <widget-type>/
              widget.json
              locales/
                <locale code, for example, en or en_US>/
                            ns.<widget-type>.json
                <other locale codes>/
                            ns.<widget-type>.json
```

A child directory exists in the `/locales` directory for each locale you want to provide resources for and their names can be either a two-letter language code (for example, `en`) or a two-letter language code and a two-letter country code with an underscore in between (for example, `en_US`). `ISO 639-1` defines the two-letter language codes. `ISO 3166-1 alpha-2` defines the two-letter country codes.

The naming convention for resource files contained in these directories is `ns.<widget-type>.json`, for example, `ns.mywidget.json`. The `ns` prefix stands for "namespace" and widget-type corresponds to the `/widget/widget-type` directory. Resource files are in JSON format. Refer to the Resource loading section for more information on this format.

Localizable resources are defined using JSON objects composed of string keys mapped to string values. The keys represent the resource names and the values represent the localized version of each resource. The `i18next` JavaScript library is used to perform the client-side translations. An example locale file is shown below:

```
{
   "resources": {
       "siteNavigationFooter" : "Site Navigation Footer",
```

```
      "editFooterHeader" : "Select Header Links (multi-select allowed)"
    }
 }
```

Use resources in widgets

Primarily, widget resources are used when text in a widget display template is translated. A Knockout custom binding named `widgetLocaleText` is available on the storefront to invoke the translation of the resource using the store's current locale. This will ultimately call the `i18next` library, but that is invisible to the widget templates. The following example shows using the `widgetLocaleText` binding in its simplest form, passing in the resource key:

```
<h2 data-bind="widgetLocaleText: 'cartHeader'"></h2>
```

The resource files defined for a widget for the current locale are used to replace the key with the right resource. The locale is defined for the storefront when the page is loaded. In the current release of Commerce, the locale is defined for the Site. The resources for the current locale are returned with the widget data when a page is loaded. These resources are mapped onto a widget and also as a `namespace` for il8next.

If you need to translate text within a widget's JavaScript, use the translate function in the widget view model. This would be required, for example, when sending a message for display on the notification bar.

```
notifier.sendSuccess(widget.WIDGET_ID,
widget.translate('loginSuccessText'));
```

Use variable replacement

Rather than manually concatenating variables to localized strings, the il8next library has support for variable replacement.

For example, a welcome message using the user's first name can be defined in a resource, such as, `Welcome __userName__`. Then, in the display template, the translation can be invoked using a knockout object to set the `userName` variable, such as the following:

```
<span data-bind="widgetLocaleText : {value:'welcome',
attr:'innerText', params:
        {userName: firstName()}}"></span>
```

Note that in the JSON resource, a `__doubleUnderscore__` notation marks the variables but, when the resource is invoked in HTML or JavaScript, the underscores are omitted. The `i18Next` library provides other mechanisms such as support for plurals. Refer to the i18next section for more information.

Resource loading

By following the structure defined above for a widget, and putting the widget resources under the `/locales` directory for a widget, the resources will be loaded for the widget by the framework. The data to build a page in the storefront is retrieved from the Pages Web API Endpoint. For the current page, this endpoint will return both context

data and the data related to the layout of the page, such as the regions and widgets to load.

Included with the data about each widget are the resources. These resources are then mapped onto the widget and loaded when the widget is loaded. The i18next `namespace` used to load the resources is defined as part of the widget definition within the server-side Page Repository. When a new widget is created, its `widget.json` file defines a property called `i18nresources`. The following is an example:

```
{
    "name": "Widget Name",
    "version": 1,
    "javascript": "widget-type-js",
    "i18nresources": "widget-type",
    "availableToAllPages": true,
    "jsEditable": true,
 }
```

The `i18nresources` property for a widget is used to determine the file name for each locale resource file, in the format `ns.<i18nresources property>.json`.

So, for the widget definition shown above, the resources file name would be `ns.widget-type.json`. This `i18nresources` property is used in the storefront framework when loading the resources and expects the format to be as defined here. Since the resources are no longer loaded directly via a URL, the file name itself is less important, but following the convention allows for consistency.

Use common resources: i18next

Currently there are two sets of expected common resources, as described in the following:

- `ns.common`: Common text used across the store such as OK, Cancel, Close, and so on.
- `ns.ccformats`: Defines a format for a number.

Use common resources: moment

The `moment.js` library (see http://momentjs.com) is used in the storefront to format dates. This requires a resource file per language. The `en` version comes with moment but other languages require a separate JavaScript file.

Bundle images or other assets within widget

The simplest way to use custom resources (for example, images) in a widget is to host them on an external server. The visitor's browser then accesses them from that location.

It is also possible to bundle custom images within the file structure of a widget. When the extension is uploaded, the images are copied to a directory on the VFS, and you can access them via the Widget Asset Mappings. If you have an image '`image1.png`' stored in the `images/` directory of your widget, you can reference that resource in JavaScript with the following snippet:

```
widget.assetMappings["/images/image1.png"]();
```

The asset mapping is an observable, so use parentheses in JavaScript code to extract the value.

You can also access the asset mapping from a knockout template, and be aware of the current binding context (for example, if inside a loop, you may need to use `$parent` to get back to the widget context). The sample below shows the HTML for an asset mapping:

```
<img class="product-item-img" data-bind="attr: {src:
assetMappings[ productImage ] }">
```

Use ccLink binding for quicker page loading

Using the standard `href` link syntax, for example, `<a href="/aboutUs">About Us<a/>`, within a widget causes a full page load to occur.

For more efficiency, you should use the `ccLink` custom binding instead. When the `ccLink` binding is used, widgets that are shared between the current page and the linked page are maintained; in other words, they are not re-loaded and re-initialized. Instead, when a `ccLink` binding is invoked, only the widget deltas are loaded; that is, widgets that exist on the linked page but not on the current page.

The `ccLink` syntax for a hyperlink looks similar to the following:

```
<a data-bind="ccLink: 'aboutUs'"></a>
```

To facilitate the `ccLink` binding, each widget has a links observable that contains all of the data required to link to each of the page types in the storefront. The name specified in the `ccLink` binding (`'aboutUs'` in the example above) is used to retrieve data from that `widget.links()` object. For example, the following illustration shows the properties that the `widget.links()` object contains for the `aboutUs` name:

The `ccLink` binding uses two of these properties, `route` and `displayName` to generate the code for the link. Specifically, it uses the `route` property for the link URL ("/aboutUs" in the example). It uses the `displayName` property for the link text if there is no existing text in the anchor tag ("About Us" in the example). Therefore, the generated code for the `aboutUs` link would look like the following:

```
<a data-bind="ccLink: 'aboutUs' href="/aboutUs">About Us</a>
```

As part of the code generation process, the `ccLink` binding adds a click event handler to the anchor tag element. This event handler invokes internal Commerce code that requests only the widget deltas for the linked page.

It is also possible to pass `ccLink` an object that contains all of the required data, rather than using the `widget.links()` object data, for example:

```
<a data-bind="ccLink: {route: '/aboutUs', displayName:'Our History'} "></a>
```

Passing an object with all the required data is useful when working with the `product` and `collection` view models. In this case, you can pass the full view model object to the `ccLink` binding, for example:

```
<a data-bind="ccLink: product"></a>
```

Within the `product` view model are properties like the following (in addition to many others):

```
"displayName":"Block Table"
        "route":"/block-table/product/xprod2125"
```

With this usage, the generated code follows a similar approach, using the `route` property for the link URL and the `displayName` property for the link text. The following is an example:

```
<a data-bind="ccLink :product" href="/block-table/product/xprod2125">
        Block Table</a>
```

The same click event handler is added to the anchor tag for invoking the code that requests only the widget deltas for the linked page.

Understand widget versioning

Widgets can be identified by their version.

If a version other than 1, which is the default, is specified for the version property in the `widget.json` file, then Commerce will use that version number as part of the path for loading widget files. For example, the JavaScript file for the first version of a widget (either explicitly stated as `"version" : "1"` or no version specified) is loaded using the URI. The following is an example:

```
/file/widget/myWidget/js/myWidget.js
```

If v1 is later replaced on the system with v2 of the same widget, then the JavaScript file is loaded using the URI, as shown in the following:

```
/file/widget/v2/myWidget/js/myWidget.js
```

In other words, the root folder for the widget has changed from `/file/widget/myWidget` to `/file/widget/v2/myWidget`. This can provide a form of cache-busting, ensuring that the correct files are loaded for the widget.

However, Commerce does not allow multiple versions of a widget to be uploaded at the same time. That is, if a version of a widget (v1) is uploaded in an extension and, at a later time, a new version (v2) is to be uploaded, the extension containing v1 must first be deactivated before the extension with v2 can be uploaded.

The `version` property in `widget.json` also serves as a useful reference for the widget developer.

Add customizable widget settings

You can add settings to a widget that provide a finer level control over the widget's behavior when it is added to a layout.

For example, the Related Products widget has settings that allow the page designer to specify the number of related products to show and whether to display the name or price for related products. Any customizable settings that have been configured for a widget are available via the widget's Settings tab.

To view a widget's Settings tab, perform the following steps:

1. From the Design menu, select the **Layout** tab.

2. From the **Layout** tab menu, select the layout that contains the widget whose settings you want to view by clicking the grid view.
   The layout shows all of the widgets used in the layout.

3. Click on the widget's **setting** icon to view the widget information. Depending on the widget you select, you may see different tabs available, such as **Layout**, **Settings** and **About**.

4. When you have finished making your changes, click **Save**.

Any configurable widget settings you create are added to the widget's view model and can be referenced from the widget's HTML template using a `data-bind` attribute. Examples for creating the data bind are provided later in this section.

Define a widget's configurable settings

Widget settings are defined using a JSON-based schema. To add configurable settings to your widget, add the following files to your directory structure:

```
<extension-name> : the root folder of your extension
    ext.json
    widget/
        <widget-type>/
            widget.json
            config/
                config.json
                locales/
                    en_US.json
                    fr_FR.json
```

The resource files for configurable widget settings are stored in locale files within the `/<widget-type>/config/locales` directory and are not read from the widget's localization resources. However, the structure of these locale files is identical to those for widget localization resources; please refer to Localize a widget for examples.

Note that defining the locales in the format `language_country` (`en_US`) may cause an error indicating that the locale file cannot be found. The locale folder should be named using the language only, and if country-specific strings are preferred, you can optionally include the `locale_country` folder.

The structure of a `config.json` file looks similar to the following:

```
{
    "widgetDescriptorName": "QuoteWidget",
    "properties": [
        {
            "id": "quoteWidgetTitle",
            "type": "sectionTitleType",
            "helpTextResourceId": "quoteWidgetTitle1HelpText",
            "labelResourceId": "quoteWidgetTitle1Label"
        },
        {
            "id": "quoteText",
            "type": "stringType",
            "helpTextResourceId": "quoteTextHelpText",
            "labelResourceId": "quoteTextLabel",
            "defaultValue": "",
            "required": true,
            "maxLength": 50,
            "minLength": 3,
            "pattern": "regex"
        },
        {
            "id": "quoteSource",
            "type": "stringType",
            "helpTextResourceId": "quoteSourceHelpText",
            "labelResourceId": "quoteSourceLabel",
            "defaultValue": ""
        },
        {
            "id": "quoteStyle",
            "type": "booleanType",
            "helpTextResourceId": "quoteStyleHelpText",
            "labelResourceId": "quoteStyleLabel",
            "defaultValue": true
        },
        {
            "id": "quoteSize",
            "type": "optionType",
            "helpTextResourceId": "quoteSizeHelpText",
            "labelResourceId": "quoteSizeLabel",
            "defaultValue": "medium",
            "options": [
                {
                    "id": "quoteSizeSmall",
                    "value": "small",
                    "labelResourceId": "quoteSizeSmallLabel"
                }, {
                    "id": "quoteSizeMedium",
                    "value": "medium",
```

```
                                    "labelResourceId": "quoteSizeMediumLabel"
                            },
                            {

                                    "id": "quoteSizeLarge",
                                    "value": "large",
                                    "labelResourceId": "quoteSizeLargeLabel"
                            }
                      ]
                }
          ]
 }
```

The `widgetDescriptorName` property names the widget for which these settings are defined and it must match the `name` property in the widget's `widget.json` file. The properties array defines the configurable settings that should be added to the widget's Settings tab. For each property, the following standard keys are supported:

- `id`: A unique ID for the property. You use this ID in the widget's HTML template to create a data-bind to the property.

- `name`: A display name for the property that appears on the Settings tab. Note that while this property is still available for backwards compatibility, it is preferable to use the `labelResourceId` property, described below, to set the label for a property.

- `type`: The data type of the property. Refer below for supported data types.

- `helpTextResourceId`: The name of the key in the resource files whose value provides help text for the property.

- `labelResourceId`: The name of the key in the resource files whose value provides a label for the property on the Settings tab.

- `defaultValue`: The property's default value, which must be a valid value for the property's data type. See Use supported data types for configuration for more information on data types.

- `required`: A Boolean flag that determine if the property requires a value.

Use supported data types for configuration

There are a number of data types that are supported for widget settings. To specify the data type for a setting, you set the `type` key to one of the following values:

- `stringType`: Produces a text entry field that allows the page designer to specify a free form text value.

- `optionType`: Produces a drop-down list of preset values. The values are specified using an options array, shown in the example above.

- `booleanType`: Produces a checkbox that allows the property to be enabled or disabled.

- `mediaType`: Produces a menu that allows you to select a media item (e.g. Image) from your library, or by uploading a new file.

- `sectionTitleType`: Produces a read-only Section Title, defined by the `labelResourceId`, and an optional block of descriptive help text, defined by the `helpTextResourceId`, to allow you to group widget settings together. For example:

```
{
          "id": "quoteWidgetTitle",
          "type": "sectionTitleType",
          "helpTextResourceId": "quoteWidgetTitle1HelpText",
          "labelResourceId": "quoteWidgetTitle1Label"
  },
```

- `collectionType`: Produces a picker that allows you to select from the collections defined in your catalog. You define the maximum number of collections that can be chosen using the `maxLength` property. For example:

```
{
          "id": "collectionItem",    "type": "collectionType",
          "name": "collectionPickerValue",
          "helpTextResourceId": "collectionPickerValueHelpText",
          "labelResourceId": "collectionPickerValueLabel",
          "maxLength": 5
}
```

- `multiSelectOptionType`: Produces a list of preset values, as does `optionType`; however, you can select multiple values from this list. By default, the control created for this data type is a drop-down list; however, you can add the `displayAsCheckboxes` property to the setting definition and set it to true to display a set of checkboxes instead. For example:

```
{
          "id": "paymentMethodTypes",
          "type": "multiSelectOptionType",
          "name": "paymentMethodTypes",
          "required": true,
          "helpTextResourceId": "paymentMethodsHelpText",
          "labelResourceId": "paymentMethodsLabel",
          "defaultValue": "card",
          "displayAsCheckboxes": true,
          "options": [
            {
              "id": "card",
              "value": "card",
              "labelResourceId": "cardLabel"
            }
          ]
}
```

Depending on the data type it uses, a property will also support a number of data type-specific keys. For the stringType data type, you can add the following keys:

- `minLength`: Minimum length of the string value.

- `maxLength`: Maximum length of the string value.

- `pattern`: A Java regular expression pattern to use for validating the string value. The pattern key can also be used to handle number expressions. For example, the

configuration for a property that accepts a number in the range of 1 to 100 would look similar to the following:

```
{
            "id": "numberField",
            "type": "stringType",
            "name": "numberField",
            "helpTextResourceId": "numberHelpText",
            "labelResourceId": "numberLabel",
            "pattern": "^[1-9][0-9]?$|^100$" }
```

For the `optionType` and `multiSelectOptionType` data types, you can add an options key that contains a list of objects that describe the entries in the drop-down list. Each option has the following keys:

- `id`: Unique ID for the option.

- `value`: The value of the option.

- `labelResourceId`: The resource key used to display the option in the drop-down list.

As previously mentioned, the configurable settings you create are added to the widget's view model and can be referenced from templates using a `data-bind` attribute. For example:

```
<div class="quoteContainer" data-bind="style : {fontSize : quoteSize}">
        <!-- ko ifnot : quoteText() == '' -->
        <blockquote data-bind="css : {quoted : quoteStyle}">
          <p data-bind="text: quoteText"></p>
          <!-- ko ifnot : quoteSource() == '' -->
          <footer data-bind="text : quoteSource"></footer>
          <!-- /ko -->
        </blockquote>
        <!-- /ko --> </div>
```

Assign a global widget to multiple sites

Global widgets are widgets that are available to all sites in your environment. You can use these widgets to create consistency throughout your sites.

By default, global widgets apply to all sites in your Commerce instance. You may override this default and assign a global widget to be used on only specified sites. To do this, you issue a POST request using the `updateSiteAssociations` custom action of the `widgetDescriptors` resource and provide a list of sites in a `sites` property. For example, the following request updates `myGlobalWidget` to execute on `siteA` and `siteB` only:

```
POST /ccadmin/v1/widgetDescriptors/myGlobalWidget/
updateSiteAssociations {
   "sites": ["siteA", "siteB"]
 }
```

To remove site associations, issue a POST request using the same custom action with the `sites` property set to null, as displayed in the following example:

```
POST /ccadmin/v1/widgetDescriptors/myGlobalWidget/updateSiteAssociations
 {
   "sites": []
 }
```

The following is an example response for a call made using the `updateSiteAssociations` custom action, as shown in the following example:

```
{    "result": true,
    "links": [
      {
        "rel": "self",
        "href":
         http://localhost:9080/ccadmin/v1/widgetDescriptors/
recommendationsTracking_v1/
            updateSiteAssociations"
      }
    ]
 }
```

Note that any attempt to update site associations for a widget that is not global results in an error, as will attempting to associate a global widget with a site whose ID does not exist.

4

Fragment a Widget into Elements

You can separate widgets into discrete elements. This allows a business user to reposition individual elements on a widget without requiring knowledge of, or access to, the underlying presentation code.

You can also use elements to create reusable chunks of functionality that is shared by multiple widgets.

Understand elements

The following are two types of elements you can create:

- Widget-specific elements belong to a specific widget type and cannot be used by any other widget type.

- Stand-alone elements are widgets that are not tied to a specific widget type and can be used by multiple widget types.

As part of their configuration, elements define the widgets they can be placed on, either implicitly (in the case of widget-specific elements) or explicitly (in the case of stand-alone elements). Widgets can contain both widget-specific and stand-alone elements as necessary. Business users can re-arrange, show, or hide elements of both types using the tools in the Design page without the need for coding knowledge.

While it is not a hard and fast rule, the primary difference between widget-specific elements and stand-alone elements is in the location of the JavaScript on which they depend. For widget-specific elements, the JavaScript is typically contained in the parent widget's `<widget-type>.js` file. In this case, the element functions as a display mechanism for functionality that exists in the parent widget. Removing an element of this type from a widget on the Design page does not alter the functionality of the parent widget; it just removes the element from the widget's display.

Stand-alone elements typically have their own JavaScript functionality that is not dependent on any single widget. As such, they can be fully encapsulated, making it possible to share them among multiple widgets. Adding a stand-alone element to a widget adds additional JavaScript functionality to the widget, along with display mechanisms for that functionality. Note that, if you have an existing widget, you can add a stand-alone element to it without the need to upload the whole widget again.

For a widget-specific element, you do not need to explicitly specify that it belongs to its parent widget because you place the element's configuration and code underneath the parent widget's directory. For stand-alone elements, you do need to explicitly define which widgets can use the element. The Create the element directory structure and Define widget meta-data in widget.json sections provide more details on these topics.

Choosing which kind of element to create depends on the purpose of the element. For example, if you need to create a "today-element" to be used across multiple widgets, the wise choice would be to create a stand-alone element and calculate today's date within that element's JavaScript rather than rely on each widget to provide the date calculation functionality.

Create the element directory structure

The location where you create the element directory structure determines if an element is widget-specific or stand-alone.

Widget-specific elements are included in an `element/` directory under the directory structure for the widget type they pertain to. The following is an example:

```
<extension-name> : the root folder of your extension
     ext.json
     widget/
         <widget-type>/
             widget.json
             element/
                 <element-name>/
                       element.json
                       templates/
                             template.txt
             layouts/
                 <layout-name>
                            widget.template
             templates/
                 display.template
             [additional widget-related directories for templates, JS,
CSS, etc]
```

Stand-alone elements are included, in an `element/` directory, under the extension root. The following is an example:

```
 <extension-name> : the root folder of your extension
       ext.json
       element/
         <element-name>/
             element.json
             templates/
                 template.txt
```

Regardless of where you create your elements, each element must have a unique name. An element directory may also contain one or more named directories containing the JavaScript, HTML, and locale fragments related to the element, for example:

```
<element-name>/
     element.json
     js/
         element.js
     locales/
         <locale code, for example, en or en_US>/
             ns.<element-name>.json
         <other locales>/
             ns.<element-name>.json
```

```
templates/
        template.txt
```

Define element meta-data in an element.json file

You must define your new elements by providing meta-data that defines key properties.

Similar to extensions and widgets, an element requires a manifest file, called `element.json`, to define key properties. The contents of an `element.json` file look similar to the following:

```
{
   "inline" : false,
   "supportedWidgetType" : ["widget-type", "widget-type", ...],
   "translations" : [
     {
       "language" : "en_EN",
       "title" : "Title in English",
       "description" : "Description in English"
     },
     {
       "language" : "de_DE",
       "title" : "Title in German",
       "description" : "Description in German"
     }
     ]
 }
```

The attributes available for `element.json` manifests are as follows:

- `inline`: A flag denoting whether the element should be inserted as a span (`inline=true`) or div (`inline=false`) when it is added to a widget instance.

- `supportedWidgetType`: A list of widget-type names that determines the availability of the element when editing widgets on the **Design** page. Either this property or the `availableToAllWidgets` property, described below, is required for stand-alone elements. Widget-specific elements do not require either property because they are, by definition, consumed only by their parent widget.

- `availableToAllWidgets`: Set this property to `true` to allow a stand-alone element to be placed on all widget types; the element will appear in the Element library for all widgets. Omit this property altogether to restrict placement of the stand-alone element to the widget types defined in the `supportedWidgetType` property.

- `translations`: An array that provides translations for both the element title and description. The title is displayed in the Element library panel that you use to add an element to its parent widget's layout. The description is not displayed in the Commerce user interface but can provide helpful information to a developer.
  The translations property has three sub-properties, language, name and description. The language property can be either a two-letter language code (for example, `en`) or a two-letter language code and a two-letter country code with an underscore in between (for example, `en_US`). ISO 639-1 defines the two-letter language codes. `ISO 3166-1 alpha-2` defines the two-letter country codes.

Create the presentation for an element

When you create an element, you want to configure settings that allow you to present the element on your site.

To create the presentation for an element on the storefront, you have to do the following:

- Create the HTML content for the element
- Modify the widget's `display.template` to include the element, thereby creating an element-based widget.
- Add a `widget.template` file to manage editing of the element-based widget on the Design page.

Create the HTML for the element

Each element contains a block of HTML in a file called `template.txt`, located in `element-name`/templates. The format of this file is pure render-able HTML content without `doctypes` or non-body sections, for example:

```
<h1 data-bind="text: title"></h1>
```

Modify the `display.template` and `widget.template` to use elements

Element-based widgets require two of the following templates:

- The `display.template`, already discussed in Create the widget template file, provides the default presentation in the storefront for an element-based widget before any changes are made via the **Design** page. It is located in `extension-name`/widget/`widget-type`/templates.

- The `widget.template` provides the starting point for editing the widget's template in the **Design** page. It is located in `extension-name`/widget/`widget-type`/layouts/`layout-name`/widget.template.

Both templates are required and they must have identical contents.

When designing an element-based widget, you need to add some additional tags to both the `display.template` and `widget.template` files that enable elements to be rendered as part of the output page and to be edited on the Design page. An example of the tags is shown below:

```
<!-- oc layout: panels -->
  <div class="row">
    <div class="col-md-12" data-oc-id="panel-1">
      <!-- oc section: product-image -->
        <div data-bind="element: 'product-image'"></div>
      <!-- /oc -->
      <!-- oc section: product-image-carousel -->
        <div data-bind="element: 'product-image-carousel'">
    </div>
      <!-- /oc -->
    </div>
```

```
   </div>
<!-- /oc -->
```

The tags that support breaking a widget into elements include the following:

- The `oc layout` tag tells Commerce to start parsing this section of the template for use on the **Design** page. Any code that resides before or after the `oc` layout tag is ignored by the **Design** page. Code within the tag is editable on the **Design** page.

- The outer div, `<div class="row">`, creates a standard Bootstrap row to contain the elements. A widget template can have multiple rows to contain its elements.

- The inner div, `<div class="col-md-12" data-oc-id="panel-1">`, creates a panel within the row (note that this is a **Design** page panel, not a traditional Bootstrap panel). Panels contain draggable user interface elements that can be repositioned when editing the widget on the **Design** page. A row can contain multiple panels but their widths must add up to 12 (this is a requirement of the underlying Bootstrap grid). For information on Bootstrap, refer to the Bootstrap documentation.
  Note: The **Design** page uses the `data-oc-id` attribute to identify each panel. This custom attribute was created so that a page developer can alter the class or ID attributes of the panel div without breaking the **Design** page's functionality. Also, the **Design** page is currently restricted to Bootstrap's desktop grid classes, for example, `col-md-x`.

- The `oc` section tags identify the individual draggable UI elements. Everything contained in an `oc` section tag is repositionable as one atomic unit, even though there may be multiple lines of code or many sub-elements within the section.
  To specify an element within a panel, a `<div>` block is created and a data-bind is used with `element` as the binding attribute and the name of the element as a string. This name corresponds to the element's directory.

  Important: When you use the element binding, you must ensure that the current binding context is the widget, or that you can return easily to the widget. There are currently problems if you try to use the element binding within a `ko foreach` as each iteration of the loop is bound to a list item.

The tags described above all use HTML comment syntax. This syntax is useful because the tags do not need to be removed before being sent to the browser as they have no visible effect on the storefront pages. Also, this format is familiar to Knockout developers.

Create custom element JavaScript

Elements can include JavaScript if so required.

As with widgets, JavaScript for an element should be created as an anonymous `Require.js` module using the format shown in the code example below. The module must be named `element.js` and saved in the `<element-name>/js/` directory for the associated element. Within the module, you must create a variable called `elementName` and set that variable to the name of the element. For example, if the element files are added via an extension in `<extension-name>/element/my-element`, the value of `elementName` needs to be `my-element`. After these requirements are met, you can add whatever functionality your element requires to the module.

```
define(
    ['jquery', 'knockout', pubsub'],
    function($, ko, pubsub) {
            "use strict";
```

```
     return {
      elementName: 'my-element',
      };
    }
 );
```

Note: See http://requirejs.org/docs/api.HTML#defdep for more details on `Require.js` modules.

When a widget is loaded, the JavaScript for its elements gets added to the widget's view model. The element template is still loaded at the widget's view model scope and the element JavaScript is available via:

```
$data.elements['<elementName>']
```

It is a good practice to check for the element JavaScript before attempting to use it. This can be done in the element's template, as shown in the following example:

```
<!-- ko if: initialized()
&& $data.elements.hasOwnProperty('<elementName>') -->
              [Element template code] <!-- /ko -->
```

Note that without the `initialized() &&` condition, the JavaScript loads correctly but the template code within the block is not displayed. This is because the widget template has been rendered and the `if` statement already evaluated to false by the time the element JavaScript loads. The check on the widget's initialized observable ensures that the `if` statement will be re-evaluated when the value of initialized changes from `false` to `true`.

onLoad()

To run custom logic when the element is instantiated, add an `onLoad()` function to the element JavaScript's return object, as displayed in the following example:

```
onLoad: function(widget) {
   //onLoad code here.
}
```

The `onLoad()` function runs once the element has finished loading and is populated with the necessary data. This is the main access point to configure data for the element after its properties have been loaded and the system is ready to display the element. As the `onLoad()` function is only called the first time the element is instantiated, when returning to the same "page," the element does not need to be re-instantiated and so `onLoad()` is not called again.

5

Use Stacks for Increased Widget Layout Control

Page layouts are made up of regions into which widgets are placed.

By default, all widgets placed in a region take up the full width of the region and are rendered vertically, one on top of the other. To introduce greater control over how widgets are rendered within a region, you must create a stack extension. This chapter describes what a stack extension is and how to create one. It includes:

Understand stacks

You can use stacks to group your widgets into regions and flows, for example, creating a series of steps.

Internally, a stack is represented as a region at the same level as the other regions in the page layout. A stack contains sub-regions that, in turn, contain widgets. The following is an example:

```
Page layout
    [Other page regions]
    Stack [internally represented as a region]
        Sub-region 1
            Widget 1a
        Sub-region 2
            Widget 2a
            Widget 2b
        Sub-region 3
            Widget 3a
            Widget 3b
    [Other page regions]
```

The template for the stack contains any logic you need to manage the rendering of the stack's sub-regions and widgets. For example, a stack template can define a Bootstrap tabbed container where each sub-region correlates to a tab and the widgets contained in each sub-region are rendered on the associated tabs. This technique is used by the Progress Tracker stack, which is the only stack extension included with Commerce. Available on the Checkout layouts, the Progress Tracker allows you to create a series of steps for the checkout process, for example, Login/Checkout, Customer Details, Payment Details, and Review Order. Each checkout step is rendered in its own tab and each tab contains the widgets that are necessary to render the user interface for that step (along with a **Next** button to progress to the next step in the stack).

Bootstrap user interface controls that can help you manage the display of sub-regions and widgets in a stack include, but are not limited to, tabs, pills, collapsible panels, carousels, and modal dialog boxes. You can also create your own custom controls. You code the controls as you normally would and make calls to the `RegionViewModel` object, which represents the stack, to retrieve the sub-regions and widgets to be rendered within each control.

Note: For details on adding a Progress Tracker stack to your page layouts, see Customize your store layouts.

Create the stack structure

Stacks are added to an extension's structure in a `/stack` directory that is at the same level as the extension's `ext.json` file.

Each stack should have its own child directory in the `/stack` directory. The following example shows the directories and files that can be included for a stack:

```
<extension-name> : the root folder of your extension
    ext.json
    stack/
        <stack-type>/
            stack.json
            templates/
                stack.template
            less/
                stack.less
                stack-variables.less
            locales/
                <locale code>.json
                <another locale code>.json
```

Define stack meta-data in stack.json

The `stack.json` file, located in the `/stack/<stack-type>` directory, defines meta-data for a stack.

An example of a `stack.json` file is provided below:

```
{
   "availableToAllPages": true,
   "configurable":false,
   "configuration": {},
   "name": "Accordion Container",
   "regions": [
     {
       "name": "Accordion 1",
       "width": 12
     },
     {
       "name": "Accordion 2",
       "width": 12
     }
   ],
   "stackType": "accordionContainer",
   "styleSettings": {},
   "version": 1
 }
```

The following list describes all the properties that you might choose to include in a `stack.json` file.

- `availableToAllPages`: Set this property to true to allow the stack to be placed on all page types; the stack will appear in the Component library for all pages. Omit this property altogether to restrict placement of the stack to the page types defined in the `pageTypes` property. Note that this property and the `pageTypes` property are mutually exclusive but one of them must be set.

- `pageTypes`: Defines which page types the stack can be placed on; the stack will appear in the Component library for the specified pages. If you use this property, you must omit the `availableToAllPages` property. Note that this property and the `availableToAllPages` property are mutually exclusive but one of them must be set.
  Available page types include:

  - `product`

  - `category`

  - `home`

  - `cart`

  - `checkout`

  - `confirmation`

  - `article`

  - `error`

  - `searchResults`

  - `noSearchResults`

  The stack will appear in the Component library for the page types you specify.

- `configurable`: Set this property to false. It is for future use.

- `configuration`: Leave this array empty. It is for future use.

- `name`: Defines the display name for the stack in the component library.

- `regions`: An array that defines the default sub-regions that are available within the stack when a new instance of this stack is created. Each item in the array includes the following properties:

  - `name`: The display name of the default region.

  - `width`: This property is required to pass validation and should be set to 12.

- `stackType`: A unique identifier for the stack.

- `styleSettings`: Leave this array empty. It is for future use.

- `version`: The numeric version of the stack.

Create the stack template

Each stack requires a Knockout template called `stack.template` to render the stack.

The `stack.template` file must reside in the `/stack/<stack-type>/templates` directory. Using the stack's `RegionViewModel` object, the `stack.template` can access the sub-regions within the stack, via the `regions` observable array, and the widgets within each sub-region, via the `widgets` observable array for each sub-region.

This section provides some examples to show you how you might create a template that integrates Bootstrap UI controls with a stack's content. The following code sample shows the stack template for a tabbed container:

```
<div class="tabbedContainer">
   <!--  RENDER Bootstrap tabs -->
   <ul class="nav nav-tabs" data-bind="attr: { id: 'tabbedNav-'+id()+'-
pills'}">
      <!-- ko foreach: regions -->
        <li role="presentation" data-bind="css: {active: $index() ===
0},
            attr: { id: 'tabbedNav-'+$parent.id()+'-
pill-'+$index() }">
          <a data-toggle="tab" data-bind="
            attr: { 'href':'#tabbedContainer-' + $parent.id() + '-
tab-' + $index()}">
              <span data-bind="text: displayName"></span>
          </a>
        </li>
      <!-- /ko -->
   </ul>
   <!--RENDER tabbed content -->   <div class="tab-content">
     <!-- ko foreach: regions -->
       <div role="tabpanel" class="stage tab-pane" data-bind="
            attr: { id: 'tabbedContainer-'+$parent.id()+'-
tab-'+$index() },
            css: {active: $index() === 0}">
          <!--RENDER widgets in each tab -->
          <!-- ko foreach: widgets -->
```

```
            <div data-bind="
                  template: {name: templateUrl(),templateSrc: templateSrc()}">
            </div>
          <!-- /ko -->
        </div>
      <!-- /ko -->
    </div>
 </div>
```

This template includes two `ko foreach: regions` bindings. The first iterates over the stack's sub-regions and creates a tab for each sub-region. The second iterates over the sub-regions again and renders the contents of each tab. The content rendering is accomplished through a `ko for each: widgets` binding that iterates over the widgets contained in each sub-region.

Similarly, the following code excerpt shows the rendering of widgets in collapsible panels:

```
<!-- RENDER displayName from the stack -->
 <h2 data-bind="text:displayName"></h2>
 <div class="panel-group" id="accordion" role="tablist">
   <!-- RENDER stack sub-regions -->
   <!-- ko foreach: regions -->
     <div class="panel panel-default">
       <div class="panel-heading" role="tab" id="headingOne">
         <h4 class="panel-title">
           <a role="button" data-toggle="collapse" data-parent="#accordion"
               data-bind="attr: {
                       'href': '#accordionContainer-' + $parent.id() + '-
tab-' + $index()},
                        text:displayName" aria-expanded="true" aria-
controls="collapseOne>
           </a>
         </h4>
       </div>
       <div class="panel-collapse collapse" role="tabpanel"
           aria-labelledby="headingOne"
           data-bind="attr: {
               id: 'accordionContainer-'+$parent.id()+'-tab-'+$index() },
               css: {in: $index() === 0}">
       <div class="panel-body">
         <!-- RENDER widgets within each sub-region -->
         <!-- ko foreach: widgets -->
           <div data-bind="template: {
               name: templateUrl(),
               templateSrc: templateSrc()}">
           </div>
           <!-- /ko -->
       </div>
      </div>
     </div>
   <!-- /ko -->
 </div>
```

This code iterates over the sub-regions in the stack and renders a collapsible panel for each one. To define the content for each panel, the code iterates over the widgets in each sub-region and renders them using the template defined in the `template` data-bind.

Configure a stack's style

CSS specific to a stack is contained in the `/stack/<stack-type>/less` directory. The following is an example:

```
<extension-name> : the root folder of your extension
    ext.json
    stack/
        <stack-type>/
            stack.json
            less/
                stack.less
                stack-variables.less
```

A style file for a stack is always named `stack.less` and any variables referenced in this style file are contained in a companion file called `stack-variables.less`. An example of a `stack.less` file is shown below:

```
.tabbedContainer {
    .nav-tabs {
      span {
        text-align: @tabTitleAlignment;
      }
    }
    .nav-tabs > li > a {
      background-color: @tabBackgroundColor;
      color: @tabTextColor;
    }
    .nav-tabs > li.active > a {
      background-color: @activeTabBackgroundColor;
      color: @activeTabTextColor;
    }
 }
```

A supporting `stack-variables.less` file for this example would look similar to the following:

```
@activeTabBackgroundColor:#195D8E;
@activeTabTextColor:#FFFFFF;
@tabBackgroundColor:#d1d1d1;
@tabTextColor:#3d3d3d;
@tabTitleAlignment:center;
```

The `stack.less` file, like a `widget.less` file, can define a stack's styles using the Less language or native CSS. Less files across the entire storefront are compiled to make one CSS file.

Similar to widgets, it is important that any style overridden within a stack Less file only applies to that stack, and does not change the style across the storefront. One way to achieve this is to format the stack's styles using the Less nested format. For more details on this approach, see Configure a widget's style.

Note: If you design a custom style as part of an extension, the Less style may not be compiled immediately after uploading the extension package. If this happens, open the stack's style in the code editor (on the **Design** page) and make a superficial edit, then resave. This process forces compilation of the style.

Create a quick view popup using a popup stack

Popup stacks enable a shopper to view information on a popup screen and are mostly related to product listings.

For further details on popup stacks, see Customize your store layouts.

One instance of the popup stack is the Quick View Popup Stack which is available within the Collection and Search Results layouts. You can apply the same quick view logic to the Related Products widget, available on the Product layout, and the Product Recommendations widget, available on all layouts.

The following illustrates applying the quick view popup logic to the Related Products widget:

1. Create a new instance of the Related Products widget within the Product layout.

2. Edit the `related-products-carousel` element by adding the global Quick View element, within the `cc-item-detail` HTML element. This should be bound with an instance of a `ProductViewModel`.

```
 <!-- ko foreach: relatedProductGroups -->
   <div class="item row" data-bind="css:{'active': $index()==0}, for
each: $data">
     <div data-bind="css: $parents[1].spanClass()" class="cc-product-
item">
       <div class="cc-item-detail">
       <!-- Quick View -->
         <div data-bind="setContextVariable: {name:'product',
value: $data}"
          class="quickViewElement">
 <!-- ko with: $parents(1) -->
        <div data-bind="element: 'product-quickview', attr:
{ id:'product-quickview-grid-' + $parent.id() } ">
        </div>
          <!-- /ko -->
        </div>
      <!-- Quick View -->
```

3. Add CSS styling to the Quick View element within the Related Products widget.

```
#cc-relatedProducts {
  quick-view {
     position: absolute;
     display: none;
     padding: 5px 10px;
     background-color: #195d8d;
```

```
    color: white;
      border: 1px dotted white;
    cursor: pointer;
    p: {
       font-weight: bold;
       margin:0;
            }
  }
    .carousel.inner {
      .cc-item-detail {
        .quick-view {
           left: 32%;
           top: 50%;
        }
      }
  }
  .cc-item-detail: hover  {
    .quick-view {
      display:inline-block;
    }
  }
}
```

4. Save your changes and open the **Layout** tab.

5. Open the **Product** layout and select **Grid View** from the configuration toolbar.

6. Locate the **Product Details** widget, add a new row below it, and drag a **Popup Stack** to that new row.

7. Open the **Main** sub-region and drag the **Related Products** widget to it from the **Components** menu.

8. Open the Popup sub-region and drag the **Product Details** widget to it from the **Components** menu. Use the existing instance of the Product Quick View.

9. Publish the changes, and verify them on the storefront.

Notes

• Changing the Related Products Carousel element will not cause the Related Products widget to break when it is not implemented with the Quick View, as there is a check to ensure that the widgets exist within a stack before it displays the quick view link.

• The example assumes that some related products have already been set up in the catalog, otherwise the widget is not displayed.

6
Add Site Settings

An extension can contain site settings, which are configurable parameters that are globally accessible to the storefront code.

Site settings are added to the Site view model and are available for use by all widgets. Site settings allow you to create a single setting that controls a feature across multiple widgets. A site settings extension creates a custom settings panel in the administration interface after it has been uploaded. This panel allows merchandisers to configure the settings. To see a site settings panel, go to the Settings page and click the name of the site settings extension under Extension Settings. For example, the illustration below shows a Sample Site Settings extension that allows a merchandiser to configure a variety of settings.
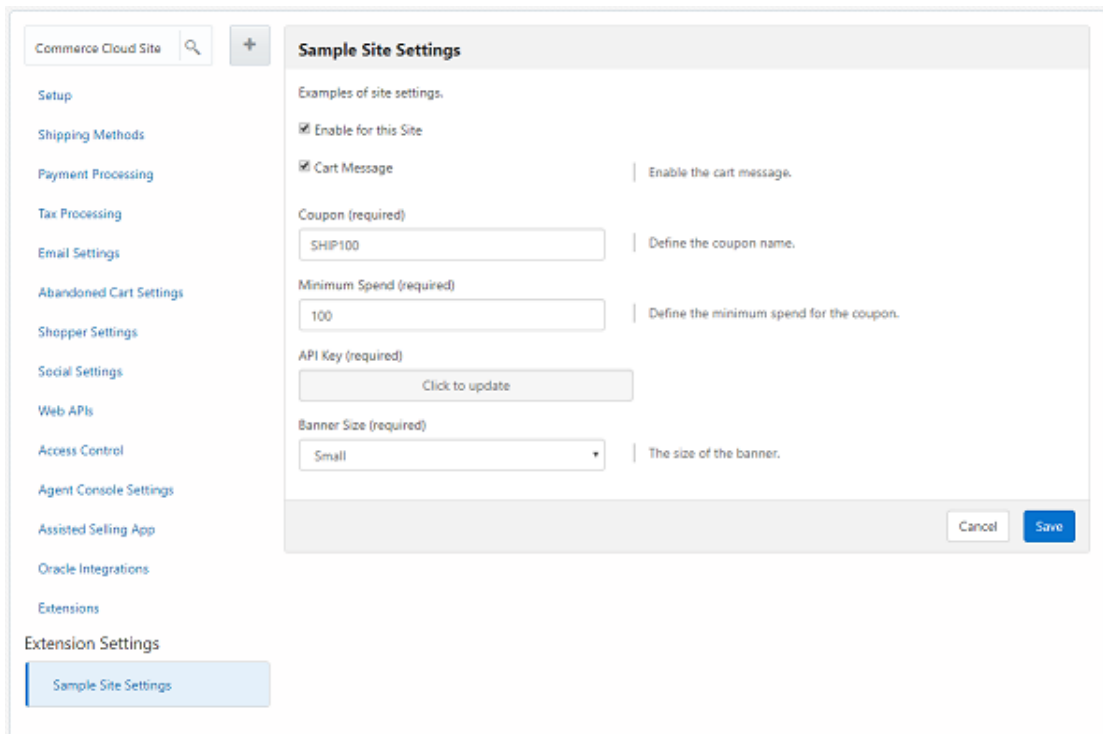
Any site settings you create can be referenced from a widget's HTML template using a data-bind attribute. Examples for creating the data-bind are provided later in this section.

Define site settings

When you create widgets, you want to define the settings that it follows to display your site.

Site settings are defined using a JSON-based schema. To add site settings to your storefront, add the following files to your directory structure:

```
<extension-name> : extension root directory
    ext.json
    config/
```

```
<settingsID>/ : site settings root directory
    config.json
    locales/
        en_US.json
        fr_FR.json
```

The resource bundles for site settings are stored in locale files under the `/config/locales` directory and look similar to the following:

```
{
  "resources" : {
    "enabledHelpText": "Enable the cart message.",
    "enabledLabel": "Cart Message",
    "couponHelpText": "Define the coupon name.",
    "couponLabel": "Coupon",
    "minSpendHelpText": "Define the minimum spend amount for the
coupon.",
    "minSpendLabel": "Minimum Spend",
    "sizeHelpText": "The size of the banner.",
    "sizeLabel": "Banner Size",
    "sizeSmallLabel": "Small",
    "sizeMediumLabel": "Medium",
    "sizeLargeLabel": "Large",
    "passwordHelpText": "Set the value for API key.",
    "passwordLabel": "API Key",
    "title": "Sample Site Settings",
    "description":"Examples of site settings."
  }
}
```

The structure of these files is identical to those for widget localization resources. Refer to Localize a widget for examples.

The structure of a `config.json` file looks similar to the following:

```
{
  "widgetDescriptorName": "multisiteconfigdemo",
  "titleResourceId": "title",
  "descriptionResourceId": "description",
  "properties": [
    {
      "id": "enabled",
      "type": "booleanType",
      "name": "enabled",
      "helpTextResourceId": "enabledHelpText",
      "labelResourceId": "enabledLabel",
      "defaultValue": true
    },
    {
      "id": "coupon",
      "type": "stringType",
      "name": "coupon",
      "helpTextResourceId": "couponHelpText",
      "labelResourceId": "couponLabel",
```

```
        "defaultValue": "SHIP100",
        "minLength": 6,
        "maxLength": 10,
        "required": true
      },
      {
        "id": "minSpend",
        "type": "stringType",
        "name": "minSpend",
        "helpTextResourceId": "minSpendHelpText",
        "labelResourceId": "minSpendLabel",
        "defaultValue": "100",
        "required": true
      },
      {
        "id": "password",
        "type": "passwordType",
        "name": "password",
        "helpTextResourceId": "passwordHelpText",
        "labelResourceId": "passwordLabel",
        "required": true
      },
      {
        "id": "bannerSize",
        "type": "optionType",
        "name": "bannerSize",
        "required": true,
        "helpTextResourceId": "sizeHelpText",
        "labelResourceId": "sizeLabel",
        "defaultValue": "s",
        "options": [
          {
            "id": "sizeSmall",
            "value": "s",
            "labelResourceId": "sizeSmallLabel"
          },
          {
            "id": "sizeMedium",
            "value": "m",
            "labelResourceId": "sizeMediumLabel"
          },
          {
            "id": "sizeLarge",
            "value": "l",
            "labelResourceId": "sizeLargeLabel"          }
        ]
      }
    ]
 }
```

The `titleResourceId` property specifies a key in the resource bundles that is used to retrieve the title for the panel in the administration interface; for example, "Sample Site Settings" in the illustration above. The `descriptionResourceId` property specifies a key for the

descriptive text that appears below the title. In the illustration, this is "Examples of site settings."

The remainder of the `config.json` file consists of a properties array that defines individual site settings and their key/value pairs. Site settings use the same standard keys as configurable widget settings, namely `id`, `name`, `type`, `helpTextResourceId`, `labelResourceId`, `defaultValue`, and `required`. Site settings can also use the same data types that are available to configurable widget settings, for example, `stringType`, `multiSelectOptionType`, and so on. Both the standard keys and the data types are described in full detail in Define a widget's configurable settings.

The Storefront endpoint also allows you to hide a setting or a property, such as creating a custom payment setting that should not be displayed on your storefront, or allowing a setting that you might not want customers to see to be sent to the storefront. By default, all properties are exposed in the Storefront endpoint, but you can use the `public` property to identify properties or settings that should not be displayed by setting the property to `"public": false`. Note that the public property does not work for password fields. The password field type, while visible in the site settings, will not function correctly, and is only applicable in custom payment settings.

Configure settings per site

Site settings can be configured on a site-by-site basis. If your Commerce instance is running multiple sites, the values a merchandiser specifies in a settings panel apply only to the currently selected site. The merchandiser can then select another site and supply different values for that site.

In some cases, a site settings panel may have settings that make sense for certain sites but not for others. In this situation, you can give merchandisers the option of disabling a site settings panel completely for individual sites. To do this, include the following in the `config.json` file of the extension that creates the panel:

```
"enableSiteSpecific": true
```

This setting must appear in top-level array in the file (that is, not within the properties array).

Setting `enableSiteSpecific` to true adds a checkbox to the panel for specifying whether the settings in the panel are enabled for the current site. The checkbox is initially selected for each site, but a merchandiser can deselect it for individual sites. Deselecting the checkbox disables the panel for a site and causes the fields in the settings panel to disappear for that site. The fields reappear if the merchandiser subsequently selects the checkbox again.

Reference site settings in widget templates

Any site settings you create are added to the Site view model and can be referenced from templates using a data-bind attribute, for example:

```
<p data-bind="text: site().extensionSiteSettings['settingsID']
['propertyID']"></p>
```

Where `settingsID` is the name of the site settings root directory and `propertyID` is the value of the `id` property for the property you want to access, as defined in the

properties array. For example, assuming the earlier example has a site settings root directory of `my-settings` and you want to access the `mediaPicker` property, you would use the following:

```
<p data-bind="text: site().extensionSiteSettings['my-settings']
['mediaPicker']"></p>
```

To use multiple properties from the same site settings extension, you can use something similar to the following:

```
<!-- ko with: site().extensionSiteSettings['settingsID'] -->
  <p data-bind="text: propertyID"></p>
  <p data-bind="text: otherPropertyID"></p>
<!-- /ko -->
```

7
Include Application-level JavaScript Modules

You can use an extension to upload application-level JavaScript files.

JavaScript files uploaded using this mechanism are loaded as part of the main module, before any endpoints have been fired and before any widgets have been loaded. As such, they can be referenced as a dependency in any widget, allowing you to build reusable modules that can be shared among widgets and elements. The remainder of this section describes how to create an extension that contains application-level JavaScript modules and provides some useful examples.

Create the extension structure for application-level JavaScript

To add application-level JavaScript files to an extension, create a `global/` directory in the extension's root folder. The following is an example:

```
<extension-name> : The root folder of your extension
    ext.json
    global/
        <application-level-module-1>.js
        <application-level-module-2>.js
        <application-level-module-n>.js
```

Place any application-level JavaScript files in this `global/` directory. There is no limit to the number of files you can place in the `global/` directory but the files themselves must follow these rules:

- They must be anonymous, that is, have no package name defined in the module.

- Structurally, they must be valid `RequireJS` modules.

- Each filename must be unique. If a file with the same name has been loaded in another extension, an error is returned.

Run custom logic upon module instantiation

Similar to widgets, you can use the `onLoad()` method to run custom logic once an application-level JavaScript module is instantiated, for example:

```
return {
   onLoad : function() {
     CCLogger.info("Loading Demo KO Bindings");
   }
};
```

Reference an application-level module in a widget

To reference an application-level module in a widget, you must list the module as a dependency using the `ccResourceLoader` library.

For example, the following code creates a dependency on the `demo.shared.viewmodels` module:

```
define(
        //------------------------------------------------------------
--------
        // DEPENDENCIES
        //------------------------------------------------------------
--------
        ['jquery', 'knockout', 'ccLogger',
          'ccResourceLoader!global/demo.shared.viewmodels'],
        //------------------------------------------------------------
--------
        // Module definition
        //------------------------------------------------------------
--------
        function ($, ko, ccLogger, sharedViewModel) {      'use
strict';
          return {
            onLoad : function(widget) {
                widget.firstName =
sharedViewModel.viewModel().firstName;
                widget.surname = sharedViewModel.viewModel().surname;
                widget.doMessage = sharedViewModel.doMessage;
            }
        };
    }
 );
```

Note that you cannot reference an application-level module from another application-level module. Application-level modules are loaded in parallel whenever a page is loaded, meaning that the order in which they are loaded is not guaranteed. This means that you cannot have a dependency form one application-level module to another.

Application-level JavaScript examples

This section provides some useful examples for how you might use application-level JavaScript modules.

This first example adds a reusable Knockout custom binding that can be used across widgets:

```
define(
        //------------------------------------------------------------
--------
        // DEPENDENCIES
```

```
         //----------------------------------------------------------------
--
         ['jquery', 'knockout', 'ccLogger'],
         //----------------------------------------------------------------
--
         // Module definition
         //----------------------------------------------------------------
--
         function($, ko, CCLogger) {
           'use strict';
           // A simple binding that highlights an element when it loses
focus.
           ko.bindingHandlers.highlight_on_blur = {
             init: function (element, valueAccessor, allBindings, viewModel,
                             bindingContext) {
               $(element).on('blur.demo.ko', function() {
                 $(this).css('background-color','yellow');
               });
               $(element).on('focus.demo.ko', function() {
                 $(this).css('background-color','white');
               });
             }
           };
           return {
             onLoad : function() {
               CCLogger.info("Loading Demo KO Bindings");
             }
           };
         }
       );
```

This example demonstrates how to create an application-level module with reusable methods or view models:

```
define(
  //----------------------------------------------------------------------
  // DEPENDENCIES
  //----------------------------------------------------------------------
  ['jquery', 'knockout', 'ccLogger'],
  //----------------------------------------------------------------------
  // Module definition
  //----------------------------------------------------------------------
  function($, ko, CCLogger) {
    'use strict';
    return {
      onLoad : function() {
        CCLogger.info("Loading Demo Shared View Models");
        },
      doMessage : function() {
        alert("Shared View Models");
      },
      viewModel : ko.observable({
        firstName : ko.observable('Bob'),
        surname : ko.observable('Test')
```

```
        })
      };
    }
  );
```

Assign an application-level JavaScript module to multiple sites

You can create a JavaScript module that spans multiple sites.

By default, application-level JavaScript modules apply to all sites in your Commerce instance. You may override this default and assign an application-level JavaScript module to be used on only specified sites. To do this, you issue a `POST` request using the `updateSiteAssociations` custom action of the `applicationJavaScript` resource and provide a list of sites in a sites property. For example, the following request updates `myJSModule.js` to execute on `siteA` and `siteB` only.

```
POST /ccadmin/v1/applicationJavaScript/myJSModule.js/
updateSiteAssociations
 {
   "sites": ["siteA", "siteB"]
 }
```

To remove site associations, issue a POST request using the same custom action with the `sites` property set to null. The following is an example:

```
POST /ccadmin/v1/applicationJavaScript/myJSModule.js/
updateSiteAssociations
 {
   "sites": []
 }
```

The following is an example response for a call using the `updateSiteAssociations` custom action:

```
{   "result": true,
    "links": [
      {
        "rel": "self",
        "href": http://localhost:9080/ccadmin/v1/applicationJavaScript/
           demo.ko.extenders.js/updateSiteAssociations"
      }
    ]
}
```

To retrieve a list of all application-level JavaScript modules along with the sites they are associated with, issue a GET request to the `applicationJavaScript` resource. An

empty sites list means that a module will be loaded on all sites. The following is an example:

```
GET /ccadmin/v1/applicationJavaScript
```

For example:

```
{
   "items": {
     "ext.ko.extenders.js": {
        "sites": []
     },
     "ext.shared.viewmodels.js": {
        "sites": [
            "siteUS"
        ]
     }
   },
   "links": [
     {
       "rel": "self",
       "href": http://localhost/ccadmini/v1/applicationJavaScript/?
sites=siteUS
     }
   ]
 }
```

8

Filter REST Responses

You can define filters that limit the data returned in REST responses, thereby improving performance.

Response filters are persistent and defined ahead of time, and then passed as part of a REST request. The server uses the response filter to determine which fields to return in response to the request. A response filter has an identifying name (the response filter key), and two lists of fields, one for fields to include in the response and one for fields to exclude. A response filter can configure fields to include, fields to exclude, or both. It is the response filter's key that is passed in the data that is sent with a REST request. You create and update response filters via the REST API. For details on how to do this, see Response filters. This section discusses how to use a response filter once it has been created.

Note: The response filter functionality described in this section is, at its core, a wrapper for the existing `fields` and `exclude` query parameters in REST requests and it behaves the same way. Instead of sending a complete list of fields to include or exclude in the query parameters, you can create a persistent filter containing that data and then pass that filter instead. To support backward compatibility, if you pass a `fields` or `exclude` query parameter and a filter in a request, the filter is ignored. For more information on the `fields` and exclude `query` parameters, see REST API query parameters.

Out-of-the-box response filters

A set of response filters has been defined out of the box and some out-of-the-box widgets and one view model use them by default.

The following sections list the response filters along with the entities that use them.

- `PLPData`, used by the Product Listing widget

- `categoryNavData`, used by the Collection Navigation and Collection Navigation – Basic widgets

- `collectionData`, used by the Collection widget

- `productData`, used by the `cartViewModel` when calling the `listProducts` endpoint.

You can send a request to the `listFilters` endpoint in the Admin API to retrieve a list of the fields that are included or excluded for each of these response filters noted above. For details on how to do this, see Response filters.

Pass a response filter key in a REST call made from a widget

Once you have created a response filter via the REST API, you can pass its key in the data a widget sends along with a REST request.

The server uses the key to locate the correct response filter and then returns data accordingly. The simplest way to include a response filter key is to hard code it in the REST call, for example:

```
data["filterKey"] = "my-filter-key";

 ccRestClient.request(url, data,
       this.successFunc.bind(this),
       this.errorFunc.bind(this));
```

Commerce also provides a mechanism for programmatically determining the response filter key to pass in a REST call, allowing you to avoid hard coding it into the widget itself. This mechanism is described in Programmatically determine the correct response filter key.

Programmatically determine the correct response filter key

Commerce also provides a mechanism for programmatically determining the response filter key to pass in a REST call, allowing you to avoid hard coding it into the widget itself.

The programmatic mechanism for determining which response filter key to pass in a REST request has several parts, as follows:

- A context object that is instantiated in the widget's JavaScript file.
- A filter map that is defined in an application-level JavaScript file.
- The `CCStoreConfiguation` library.

The context object contains the data required for locating the correct response filter key in the filter map. The widget instantiates the context object with the necessary data and then passes it to the `CCStoreConfiguration` library's `getFilterToUse()` method, which locates the correct response filter key in the filter map and returns it to the widget.

Enable programmatic filter key determination

You must enable programmatic filter key determination before you can use it. To do so, create an application-level JavaScript module that lists the `CCStoreConfiguration` library as a dependency and includes the following code:

```
define(
   //----------------------------------------------------------------
-
   // DEPENDENCIES
   //----------------------------------------------------------------
-
     ['ccStoreConfiguration'],
   //----------------------------------------------------------------
-
   // Module definition
   //----------------------------------------------------------------
-
   function(CCStoreConfiguration) {
     'use strict';
```

```
    return {
      onLoad : function() {
              CCStoreConfiguration.getInstance().enableFilter();
        },
      }
    }
 );
```

See Include Application-level JavaScript Modules for details on creating and uploading an application-level JavaScript module.

Understand the filter map

The filter map uses a prioritized structure of top-level objects and nested sub-objects. `CCStoreConfiguration` compares the data in the context object to that prioritized structure when locating the key. The following code sample shows the out-of-the-box filter map that Commerce uses. In it, the filter map sets the `priorityList` variable to `["endpoint","page","identifier"]`, meaning that `CCStoreConfiguration` will first try to find a top-level object that matches the endpoint in the context object, then it will search inside that object for a matching page object, then it will search inside that object for a matching identifier object. The following is an example:

```
define(
    //---------------------------------------------------------------------
    // DEPENDENCIES
    //---------------------------------------------------------------------
    ['ccStoreConfiguration'],
    //---------------------------------------------------------------------
    // Module definition
    //---------------------------------------------------------------------
    function(CCStoreConfiguration) {
      'use strict';
      return {
        onLoad : function() {
          console.log("Loading Application Level JS");
              var priorityList = ["endpoint","page","identifier"];
              var newFilterMap = {
                    "getCollection":{
                       "megaMenuNavigation": {"ccFilterConfigKey":
"categoryNavData"},
                       "categoryNavigation": {"ccFilterConfigKey":
"categoryNavData"}
                    },
                    "listProducts":{
                      "productListingData": {"ccFilterConfigKey": "PLPData"},
                      "collectionWidget": {"ccFilterConfigKey":
"collectionData"},
                      "getProductData": {"ccFilterConfigKey": "productData"},
                      "getProductDataAndRedirect": {"ccFilterConfigKey":
"productData"}
                    }
                  };

CCStoreConfiguration.getInstance().updateFiltersToUse(newFilterMap);
```

```
            },
        }
    }
);
```

The top-level objects in the out-of-the-box filter map correspond to endpoints and their sub-objects correspond to identifiers. In other words, `getCollection` and `listProducts` represent endpoints and their children (`megaMenuNavigation`, `categoryNavigation`, `productListingData`, and so on) represent identifiers. (Note that the `getCollection` and `listProducts` endpoints return data that is page-independent so, even though the priority list includes page, page objects are not defined for these two endpoints in the out-of-the-box filter map.)

To understand how `CCStoreConfiguration` compares the contents of a context object to the filter map, we will compare the following context object to the out-of-the-box filter map:

```
var contextObj = {};
    contextObj["endpoint"] = "getCollection";
    contextObj["identifier"] = "categoryNavigation";
```

When considering this context object, `CCStoreConfiguration` first looks for a matching endpoint among the top-level objects in the filter map (because endpoint is first in the priority list). In this case, `CCStoreConfiguration` finds the `getCollection` top-level object. Next, `CCStoreConfiguration` looks for a matching `page` sub-object within the `getCollection` top-level object (because page is second in the priority list). The context object does not have page data, however, so `CCStoreConfiguration` moves on to find the next piece of data in the priority list, which is `identifier`. The thing to note here is that `CCStoreConfiguration` continues to look for the next piece of data in the current object. In other words, it looks for a `categoryNavigation` sub-object in the `getCollection` top-level object. When `CCStoreConfiguration` finds the `categoryNavigation` sub-object, it sees that the object has a `ccFilterConfigKey` defined for it. `CCStoreConfiguration` retrieves this filter key, `categoryNavData`, and returns it to the widget.

You can set your priority list and the object structure of your filter map in any way that makes sense for your implementation and then define context objects in your widgets that use that updated structure. However, keep in mind that the out-of-the-box filters, and the widgets that use them, may be affected by changes you make and may need modifications as a result.

Create a context object and use it to retrieve the response filter key

To create a context object and use it to retrieve a response filter key, add code similar to the following to the widget's JavaScript file. Note that you must also add a dependency on the `CCStoreConfiguration` library. The following is an example:

```
// Add the CCStoreConfiguration library as a dependency for this widget
 // Create the context object and populate it
 var contextObj = {};
     contextObj["endpoint"] = "endpoint-name";
     contextObj["identifier"] = "identifier-in-filter-map";
 // Call the getFilterToUse method to retrieve the response filter key
 var filterKey =
```

```
CCStoreConfiguration.getInstance().getFilterToUse(contextObj);
 // Add the filterKey to the data passed with the REST call
 if (filterKey) {
   data["filterKey"] = filterKey;
 }
 //Make the REST call
 ccRestClient.request(url, data,
       this.successFunc.bind(this),
       this.errorFunc.bind(this));
         }
```

Add a new response filter key to the out-of-the-box filter map

The following code sample creates new identifiers in the out-of-the-box filter map for calls made to the getCollection and productListing endpoints. The new identifier for the getCollection endpoint is customIdentifer1 and the response filter key that is returned for it is customFilterKey1. The new identifier for the productListing endpoint is customIdentifer2 and the response filter key that is returned for it is customFilterKey2. The following is an example:

```
define(
    //-------------------------------------------------------------------
    // DEPENDENCIES
    //-------------------------------------------------------------------
    ['ccStoreConfiguration'],
    //-------------------------------------------------------------------
    // Module definition
    //-------------------------------------------------------------------
    function(CCStoreConfiguration) {
      'use strict';
      return {
        onLoad : function() {
          console.log("Loading Application Level JS");
             var priorityList = ["endpoint","page","identifier"];
             var newFilterMap = {
             "getCollection":{
                 "megaMenuNavigation": {"ccFilterConfigKey":
"categoryNavData"},
                 "categoryNavigation": {"ccFilterConfigKey":
"categoryNavData"},
                 "customIdentifier1": {"ccFilterConfigKey": "customFilterKey1"}
               },
               "listProducts":{
                 "productListingData": {"ccFilterConfigKey": "PLPData"},
                 "collectionWidget": {"ccFilterConfigKey": "collectionData"},
                 "getProductData": {"ccFilterConfigKey": "productData"},
                 "getProductDataAndRedirect": {"ccFilterConfigKey":
"productData"},
                 "customIdentifier2": {"ccFilterConfigKey": "customFilterKey2"}
               }
            };

CCStoreConfiguration.getInstance().updateFiltersToUse(newFilterMap);
        },
```

```
        }
      }
  );
```

Note that, when you override the filter map, the top-level objects you define completely replace any existing top-level objects. In other words, if you created a new filter map that looked as follows:

```
// This code overwrites the getCollection top-level object entirely
var newFilterMap = {
   "getCollection":{
            "customIdentifier1": {"ccFilterConfigKey":
"customFilterKey1"}
   },
 };
```

You would lose the `megaMenuNavigation` and `categoryNavigation` identifiers defined out of the box for the `getCollection` top-level object. However, the `listProducts` top-level object would remain unchanged because no new top-level object definition for it has been introduced. For this reason, you should be careful to include the default identifiers, shown earlier, along with any new identifiers you create unless you explicitly intend to overwrite them.

Use defaults in the filter map

The filter map supports the concept of defaults at each object level. When `CCStoreConfiguration` cannot find a match for a piece of data in the context object, it looks for a default. If it finds a default, it searches within that default object's children for the next piece of data in the priority list. If it cannot find a match or a default, it will not return a response filter key.

The concept of default objects can exist at any level in a filter map. For example, consider this filter map that sets its priority list to
`["endpoint","page","identifier","viewport"]`:

```
var newFilterMap = {
        "endpoint1": {
                "page1":{"cc-filter-config-key": "key1"},
                "page2":{"cc-filter-config-key": "key2"},
                "page3":{
                    "identifier1":{"cc-filter-config-key":
"key3"},
                    "identifier2":{"cc-filter-config-key":
"key4"},
                    "cc-filter-config-key": "key11",
                    "default":{
                        "viewport1":{"cc-filter-config-key":
"key5"},
                        "viewport2":{"cc-filter-config-key":
"key6"},
                        "default":{"cc-filter-config-key":
"key7"}
                    }
                },
```

```
                            "cc-filter-config-key": "key8",
                            "default":{
                                "identifier1":{"cc-filter-config-key": "key9"},
                                "default":{"cc-filter-config-key": "key10"}
                            }
                }
        };
```

The following table lists a variety of sample context objects and the response filter key that
would be returned for them based on this filter map:

Context Object Data	Filter Key Returned
endpoint1	key8
endpoint1, page1	key1
endpoint1, page3	key11
endpoint1, page3, identifier1	key3
endpoint1, page3, identifier2	key4
endpoint1, page1, identifier3	key1
endpoint1, page3, identifier3, viewport1	key5
endpoint1, page3, identifier3, viewport3	key7
endpoint1, page4	null
endpoint1, page4, identifier1	key9
endpoint1, page4, identifier2	key10
endpoint1, page1, identifier1	key1

Change response filters used by out-of-the-box widgets

If you want to change the response filter used in one of the default widgets (described in Out-
of-the-box response filters), Oracle recommends that you create a new response filter and
assign it to the endpoint/identifier combination the widget uses in its context object.

To do this, you create an application-level JavaScript module that requires in the
CCStoreConfiguration library and creates a modified version of the filter map. As a reminder,
the default filter map looks like this:

```
define(
    //----------------------------------------------------------------
    // DEPENDENCIES
    //----------------------------------------------------------------
    ['ccStoreConfiguration'],
    //----------------------------------------------------------------
    // Module definition
    //----------------------------------------------------------------
    function(CCStoreConfiguration) {
      'use strict';
      return {
        onLoad : function() {
          console.log("Loading Application Level JS");
                  var filterMap = {
                      "getCollection":{
                         "megaMenuNavigation": {"ccFilterConfigKey":
```

```
"categoryNavData"},
                              "categoryNavigation": {"ccFilterConfigKey":
"categoryNavData"}
                            },
                            "listProducts":{
                              "productListingData": {"ccFilterConfigKey":
"PLPData"},
                              "collectionWidget": {"ccFilterConfigKey":
"collectionData"},
                              "getProductData": {"ccFilterConfigKey":
"productData"},
                              "getProductDataAndRedirect":
{"ccFilterConfigKey": "productData"},
                            }
                        };

CCStoreConfiguration.getInstance().updateFiltersToUse(newFilterMap);
                },
            }
        }
 );
```

The code in this example changes the response filter key for the Collection Navigation
widget from `categoryNavData` to `customFilterKey1`.

```
define(
    //------------------------------------------------------------------
-
    // DEPENDENCIES
    //------------------------------------------------------------------
-
    ['ccStoreConfiguration'],
    //------------------------------------------------------------------
-
    // Module definition
    //------------------------------------------------------------------
-
    function(CCStoreConfiguration) {
      'use strict';
      return {
        onLoad : function() {
          console.log("Loading Application Level JS");
                  var filterMap = {
                      "getCollection":{
                        "megaMenuNavigation": {"ccFilterConfigKey":
"customFilterKey1"},
                        "categoryNavigation": {"ccFilterConfigKey":
"categoryNavData"}
                      },
                      "listProducts":{
                        "productListingData": {"ccFilterConfigKey":
"PLPData"},
                        "collectionWidget": {"ccFilterConfigKey":
"collectionData"},
```

```
                               "getProductData": {"ccFilterConfigKey":
"productData"},
                               "getProductDataAndRedirect": {"ccFilterConfigKey":
"productData"},
                          }
                     };

CCStoreConfiguration.getInstance().updateFiltersToUse(newFilterMap);
                },
            }
        }
 );
```

Remember that the top-level objects you define in the filter map override any default top-level objects. In other words, if you modify a top-level object in the filter map, only the identifiers you explicitly include your top-level object are used by the `CCStoreConfiguration` library. For this reason, you should be careful to include the default identifiers in top-level objects to avoid overwriting them. See Add a new response filter key to the out-of the-box filter map for more information.

The following table defines which identifier and filter key combination is used by the default widgets:

Widget	Endpoint	Identifier	Response filter Key
Collection Navigation	getCollection	megaMenuNavigation	categoryNavData
Collection Navigation – Basic	getCollection	categoryNavigation	categoryNavData
Product Listing	listProducts	productListingData	PLPData
Collection	listProducts	collectionWidget	collectionData

Filter REST calls made from within a view model

REST calls may also be made from within view models as the view models interact with the server while doing do their work.

IMPORTANT: This section describes how to filter REST calls made from within a view model. Please note that dependencies exist between view models such that data retrieved by one view model may be used by another. Care must be taken when filtering view model REST calls so that you do not filter out data needed by another view model.

Filtering the responses for view model REST calls uses the context object and filter map combination described in Programmatically determine the correct response filter key. The context objects are defined within the view models themselves, however, so all you have to do to filter REST calls made from view models is create the filter map and upload it in an application-level JavaScript module (see Include Application-level JavaScript Modules for more information on creating this type of module).

The following filter map example shows the top-level objects that define which response filters are used for REST calls made from view models. It is not necessary to include all of these top-level objects in your filter map, only those for REST calls you want to filter.

However, keep in mind that the top-level objects you include in your filter map will overwrite any existing top-level objects.

```
define(

  //----------------------------------------------------------------
  // DEPENDENCIES
  //----------------------------------------------------------------
  ['ccStoreConfiguration'],

  //----------------------------------------------------------------
  // Module definition
  //----------------------------------------------------------------
  function(CCStoreConfiguration) {

    'use strict';

    return {
      onLoad : function() {

        console.log("Loading Application Level JS");

            var priorityList = ["endpoint","page","identifier"];

            var filterMap = {
            "getCollection":{
               "megaMenuNavigation": {"ccFilterConfigKey":
"categoryNavData"},
               "categoryNavigation": {"ccFilterConfigKey":
"categoryNavData"}
            },
            "listProducts":{
               "productListingData": {"ccFilterConfigKey": "PLPData"},
               "collectionWidget": {"ccFilterConfigKey":
"collectionData"},
               "getProductData": {"ccFilterConfigKey": "productData"},
               "getProductDataAndRedirect": {"ccFilterConfigKey":
"productData"}
            }
            "listMembers": {
              "ccFilterConfigKey": "key-name"
            },
            "getGiftWithPurchaseChoices": {
              "ccFilterConfigKey": "key-name"
            },
            "getAllOrdersForProfile": {
              "ccFilterConfigKey": "key-name"
            },
            "listScheduledOrdersByProfile": {
              "ccFilterConfigKey": "key-name"
            },
            "getItemType": {
              "ccFilterConfigKey": "key-name"
            },
            "getCurrentProfile": {
```

```
                "userData": {"ccFilterConfigKey": "key-name"}
              },
              "getAllPrices": {
                "ccFilterConfigKey": "key-name"
              },
              "getStockStatus": {
                "productStockStatus": {"ccFilterConfigKey": "key-name"},
                "stockStatusForProdValidation": {"ccFilterConfigKey": "key-
name"}
              },
              "getPaymentGroup": {
                "ccFilterConfigKey": "key-name"
              },
              "getOrder": {
                "orderForSubmit": {"ccFilterConfigKey": "key-name"},
                "templateOrder": {"ccFilterConfigKey": "key-name"},
              },
              "getScheduledOrder": {
                "loadOrder": {"ccFilterConfigKey": "key-name"}
              },
              "getPage": {
                "home": {
                  "layoutOnly":{"ccFilterConfigKey": "key-name"},
                  "cachableData":{"ccFilterConfigKey": "key-name"},
                  "currentData":{"ccFilterConfigKey": "key-name"}
                }
              },
              "getIncompleteOrder": {
                "loadCartForProfile": {"ccFilterConfigKey": "key-name"},
                "refreshCart": {"ccFilterConfigKey": "key-name"}
              },
              "getStockStatuses": {
                "stockStatusesForCart": {"ccFilterConfigKey": "key-name"},
                "stockStatsToValidateCart":{"ccFilterConfigKey": "key-name"},
                "stockStatsForItem":{"ccFilterConfigKey": "key-name"}
              },
              "getMetadata": {
                "dynamicProperties": {"ccFilterConfigKey": "key-name"}
              },
              "listSkus": {
                "skuListing": {"ccFilterConfigKey": "key-name"}
              }
            };

CCStoreConfiguration.getInstance().updateFiltersToUse(newFilterMap);
      },
    }
  }
);
```

The following tables describe the data that is returned for the view model REST calls to assist you as you decide what calls you want to filter.

CartViewModel

This table describes the REST calls made from the `CartViewModel` and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
{endpoint:"getIncompleteOrder", identifier:"loadCartForProfile"}	This context object is used for calls made to the `getIncompleteOrder` endpoint. The `CartViewModel` makes this call to retrieve the current incomplete order and load it in the `UserViewModel` and the `CartViewModel`. This may happen when an anonymous shopper logs in or creates an account. Also, when a shopper accepts a quoted order and moves to another page (other than the checkout page), the quoted order is removed and the incomplete order is loaded. Similarly, if a shopper is viewing an order that is pending payment and then moves to a page other than the checkout page, the pending payment order is removed and the incomplete order is loaded.
{endpoint:"getIncompleteOrder", identifier:"refreshCart"}	This context object is used for calls made to the `getIncompleteOrder` endpoint. The `CartViewModel` makes this call when the shopper changes pages or the system reloads the cart. The call retrieves any incomplete order data for the logged-in shopper and then populates the `CartViewModel` and `UserViewModel` properties with the data it has retrieved.
{endpoint:"getMetadata", identifier:"dynamicProperties"}	This context object is used for calls made to the `getMetadata` endpoint. The `CartViewModel` makes this call to retrieve metadata for dynamic order properties. **Note** that if you are using dynamic order properties, you must customize the widget code to use the `markDirty` flag depending on your preferences. You can, therefore choose to handle changes to dynamic order properties when all properties have been set, or update on selection of each dynamic property. Marking the `CartViewModel.isDirty()` flag to true would trigger an update order call.
{endpoint:"getOrder", identifier:"templateOrder"}	This context object is used for calls made to the `getOrder` endpoint. The Scheduled Order widget triggers the `CartViewModel` to make this call when the shopper clicks the Place Order button. Clicking this button places the contents of the scheduled order into the shopping cart, allowing the shopper to place a one-time order based on a scheduled order. See Configure page layouts for scheduled orders for more details.

Context Object	Description
`{endpoint:"getStockStatus", identifier:"stockStatusForProdValidation"}`	This context object is used for calls made to the `getStockStatus` endpoint. The `CartViewModel` makes this call to retrieve stock status information when the quantity of a product on the cart page is updated.
`{endpoint:"getStockStatuses", identifier:"stockStatusesForCart"}`	This context object is used for calls made to the `getStockStatuses` endpoint. The `CartViewModel` makes this call to get stock status information when it is refreshing product data for items in the cart. This call is also triggered to get stock status information when a configurable product is reconfigured. This call does not get made for orders in the `PENDING_PAYMENT` or `PENDING_PAYMENT_TEMPLATE` state because these orders cannot be edited; in other words, since the items in the order cannot be edited, stock status for those items is irrelevant.
`{endpoint:"getStockStatuses", identifier:"stockStatsToValidateCart"}`	This context object is used for calls made to the `getStockStatuses` endpoint. The `CartViewModel` makes this call to retrieve stock status information for the products in the cart during the checkout process. The call is made when the shopper has clicked the checkout link on the Checkout page and the prices of the products in the cart have not changed. **Note:** If the prices have changed, the shopper is redirected to the cart page and this call is not made.
`{endpoint:"getStockStatuses", identifier:"stockStatsForItem"}`	This context object is used for calls made to the `getStockStatuses` endpoint. The `CartViewModel` makes this call to get stock status information for all the SKUs (base product and child SKUs) of a configurable product when that product is added to the cart. (Note that the view model method that makes this call is generic enough that it can be used for adding a product to the cart that is not configurable.)
`{endpoint:"listProducts", identifier:"getProductData"}`	This context object is used for calls made to the `listProducts` endpoint. The `CartViewModel` makes this call when it needs to check whether the order has stale product data, for example, when the shopper moves from one page to another. Using this call, the `CartViewModel` retrieves product data from the server and then uses it to update the products in the cart on the client side. By comparing the new product data with the existing product data, Commerce can determine if the cart has become stale (for example, prices have changed or a product has been marked inactive) and whether repricing should be triggered.

Context Object	Description
`{endpoint:"listProducts", identifier:"getProductDataAndRedirect"}`	This context object is used for calls made to the `listProducts` endpoint. The `CartViewModel` makes this call when the shopper has clicked the checkout link and the view model needs to check whether the order has stale product data. Using this call, the `CartViewModel` retrieves product data from the server and then uses it to update the products in the cart on the client side. By comparing the new product data with the existing product data, Commerce can determine if the cart has become stale (for example, prices have changed or a product has been marked inactive) and whether repricing should be triggered or the shopper should be redirected to the cart page.
`{endpoint:"listSkus", identifier:"skuListing"}`	This context object is used for calls made to the `listSkus` endpoint. The `CartViewModel` makes this call to get product data for all the SKUs (base product and child SKUs) of a configurable product when that product is added to the cart. (Note that the view model method that makes this call is generic enough that it can be used for adding a product to the cart that is not configurable.)

delegatedAdminContacts view model

This table describes the REST calls made from the `delegatedAdminContacts` view model and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
`{endpoint:"listMembers"}`	This context object is used for calls made to the `listMembers` endpoint. The `delegatedAdminContacts` view model makes this call to retrieve a list of an account's contacts.

GiftProductListingViewModel

This table describes the REST calls made from the `GiftProductListingViewModel` and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
`{endpoint:"getGiftWithPurchaseChoices"}`	This context object is used for calls made to the `getGiftWithPurchaseChoices` endpoint. The `GiftProductListingViewModel` makes this call to retrieve the gift choices for a gift-with-purchase promotion that allows the shopper to choose her gift.

LayoutContainer view model

This table describes the REST calls made from the layout-container view model and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description

`{endpoint:"getPage",`
`page:"page",`
`identifier:"layoutOnly"}`
	This context object is used for calls made to the `getPage` endpoint. The `layout-container` view model makes this call to get the layout data for the page.

`{endpoint:"getPage",` `identifier:"cachableData"}`	This context object is used for calls made to the `getPage` endpoint. The `layout-container` view model makes this call to get page data that is appropriate to store in a cache, for example, site data.
`{endpoint:"getPage",` `identifier:"currentData"}`	This context object is used for calls made to the `getPage` endpoint. The `layout-container` view model makes this call to get page data that should not be stored in a cache, for example, user data.

OrderViewModel

This table describes the REST calls made from the `OrderViewModel` and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
`{endpoint:"getOrder",` `identifier:"orderForSubmit"}`	This context object is used for calls made to the `getOrder` endpoint. The `OrderViewModel` makes this call when moving a quoted or scheduled order to the submitted state. Specifically, when the shopper views a quoted or scheduled order's details and then chooses to check out the order, this call is triggered.

OrderHistoryViewModel

This table describes the REST calls made from the `OrderHistoryViewModel` and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
`{endpoint:"getAllOrdersForProfile"}`	This context object is used for calls made to the `getAllOrdersForProfile` endpoint. The `OrderHistoryViewModel` makes this call to retrieve the orders that are displayed on the order history page.

PaymentAuthResponseViewModel

This table describes the REST calls made from the `PaymentAuthResponseViewModel` and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
`{endpoint:"getPaymentGroup"}`	This context object is used for calls made to the `getPaymentGroup` endpoint. The `PaymentAuthResponseViewModel` makes this call to get the current authorization status when a shopper places an order using CyberSource.

ProductViewModel

This table describes the REST calls made from the `ProductViewModel` and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
`{endpoint:"getAllPrices"}`	This context object is used for calls made to the `getAllPrices` endpoint. The `ProductViewModel` makes this call to get the prices for a product when a shopper moves to product page.
`{endpoint:"getStockStatus", identifier:"productStockStatus"}`	This context object is used for calls made to the `getStockStatus` endpoint. The `ProductViewModel` makes this call to get stock information for the product when a shopper moves to product page. The view model also makes this call to get stock information for the gift choices a shopper can choose when she is presented with a gift-with-purchase promotion that allows her to choose her own gift.

scheduled-order view model

This table describes the REST calls made from the scheduled-order view model and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
`{endpoint:"getScheduledOrder", identifier:"loadOrder"}`	This context object is used for calls made to the `getScheduledOrder` endpoint. The `scheduled-order` view model makes this call to load a selected scheduled order's details so that it can be displayed by the Scheduled Order widget.

scheduledOrderList view model

This table describes the REST calls made from the scheduledOrderList view model and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
{endpoint:"listScheduledOrdersByProfile"}	This context object is used for calls made to the listScheduledOrdersByProfile endpoint. The scheduledOrderList view model makes this call to retrieve the list of scheduled orders for the current profile.

skuPropertiesHandler view model

This table describes the REST calls made from the skuPropertiesHandler view model and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
{endpoint:"getItemType"}	This context object is used for calls made to the getItemType endpoint. The skuPropertiesHandler view model makes this call to get the properties for a SKU.

UserViewModel

This table describes the REST calls made from the UserViewModel and provides details on the context objects that are used to locate a response filter for each type of REST call the view model makes.

Context Object	Description
{endpoint:"getCurrentProfile"}	This context object is used for calls made to the getCurrentProfile endpoint. The UserViewModel makes this call to get profile data for the logged-in shopper.

9
Resize Images

Images are automatically sized for your customer based on the devices that they use. However you can customize the image sizes as needed.

When a shopper views a page that contains images, Commerce automatically sizes them on the client side for display on different devices, such as laptops, tablets, and mobile phones. To improve your storefront's performance, you can resize images before they are downloaded to the client browser. To do this, Commerce provides the `/images` REST endpoint which allows you to format the images returned from the server. This endpoint, however, requires a URL with a number of parameters that can be challenging to specify manually. To assist you in using the `/images` REST endpoint, Commerce provides the `ccResizeImage` custom Knockout binding. You can use this binding in your widgets to create the URL that is sent to the /images endpoint. The `ccResizeImage` binding also handles specifying a default image size as well as sizes for various viewports. This section provides information on using the `ccResizeImage` binding as well as general information about the `/images` endpoint.

Note: Commerce also includes an earlier custom Knockout binding, `productImageSource`, that creates the URL sent to the `/images` endpoint. It has some limitations, however, in that it only works with product images and it does not automatically detect the viewport. Oracle recommends using the `ccImageResize` binding going forward but the `productImageSource` binding will continue to work. For more information on the `productImageSource` binding, refer to the View Model JSDoc for Commerce.

Default image sizes

There are default image sizes available.

By default, Commerce uses the following maximum sizes (in pixels) for images:

- Extra Small: 100x100
- Small: 300x300
- Medium: 475x475
- Large: 940x940

Resize images using the ccResizeImage binding

The custom `ccResizeImage` binding provides scaled images for display on the UI.

It also provides the ability to specify an alternate image and image text to be loaded in the event that the image cannot be found. The `ccResizeImage` binding must be used inside an `<img>` tag, for example:

```
<img data-bind="ccResizeImage: {
            source: '/file/v2/products/AntiqueWoodChair_full.jpg',
            alt:'Antique Wood Chair',
```

```
                        errorSrc:'images/noImage.png',
                        errorAlt:'No Image Found'}"></img>
```

Set override dimensions for specific viewports

When using the `ccResizeImage` binding, you can specify override dimensions for specific viewports. You can also specify a default size for any viewport for which no override dimension is provided. For example, in the following code snippet, the `ccResizeImage` binding returns an image of size 80x80 and 120x120 for `xsmall` and `medium`viewports, respectively. For all other viewports, it returns an image of size 50x50.

```
<img data-bind="ccResizeImage: {
            source: '/file/v2/products/AntiqueWoodChair_full.jpg',
            xsmall: '80,80',
            medium: '120,120',
            size:'50,50',
            alt: 'Antique Wood Chair',
            errorSrc:'images/noImage.png',
            errorAlt:'No Image Found'}"></img>
```

Convert images to JPEG format

The `ccResizeImage` binding can be used to convert images to JPEG using the optional `outputFormat` attribute. When `outputFormat` is set to JPEG (the only option currently supported), a source image is converted to a JPEG image. You can specify an optional quality attribute to adjust the quality of the resulting JPEG image (0.0 is the lowest quality, 1.0 is the highest quality). For PNG images with a transparency layer, you can control the background color of the converted JPEG (which does not support transparency) by setting the optional `alphaChannelColor` attribute. For example, the following `<img>` tag converts the `logo.png` image to a JPEG with a quality factor of 0.8 and replaces the transparent layer with the color black.

```
<img data-bind="ccResizeImage: {
     source: '/img/logo.png',
            outputFormat: 'JPEG',
     alphaChannelColor: '000000',
            quality: '0.8'}"></img>
```

Note: GIF images cannot be resized or converted to JPEG as they may contain animation which is lost after resizing and conversion.

Use a srcset to specify the image to load

HTML 5 introduced the `srcset` and sizes attributes to the `<img>` tag, which allow you to specify a set of images and the conditions under which each image should be loaded. The `ccResizeImage` binding can take advantage of this functionality by making a set of differently sized images available to the browser. The browser picks the image to load based on the width that is available for the image. As the browser is resized, or as the orientation of the view port is changed, the correct image is loaded. Also, on view ports that have a higher pixel density, the browser is able to pick a higher resolution image to load that is better suited to the view port. In all cases, bandwidth use is improved because an image that is correctly sized for the circumstances is loaded.

To enable the `srcset` feature, you must set the `isSrcSetEnabled` attribute to true for the `ccResizeImage` binding:

```
<img data-bind="ccResizeImage: {
          source: '/file/v2/products/AntiqueWoodChair_full.jpg',
          isSrcSetEnabled: true,
          alt: 'Antique Wood Chair',
          errorSrc:'images/noImage.png',
          errorAlt:'No Image Found'}"></img>
```

When the `isSrcSetEnable` attribute is set to true, the `ccResizeImage` binding uses the `/ccstore/v1/images` endpoint to create a set of differently sized versions of the image defined by the source attribute. It also creates an accompanying sizes attribute that specifies which image to load based on available width. The HTML generated for the example above looks similar to this:

```
<img data-bind="ccResizeImage: {
   isSrcSetEnabled : true,
   source:'/file/v2/products/AntiqueWoodChair_full.jpg',
   alt:'Antique Wood Chair',
   errorSrc:'/img/no-image.jpg',
   errorAlt:'No Image Found'}",
   alt="Antique Wood Chair",
   src="/file/v2/products/
AntiqueWoodChair_full.jpg&amp;height=475&amp;width=475",
   srcset="
          /file/v2/products/
AntiqueWoodChair_full.jpg&amp;height=100&amp;width=100 100w,
          /file/v2/products/
AntiqueWoodChair_full.jpg&amp;height=300&amp;width=300 300w,
          /file/v2/products/
AntiqueWoodChair_full.jpg&amp;height=475&amp;width=475 475w,
          /file/v2/products/
AntiqueWoodChair_full.jpg&amp;height=940&amp;width=940 940w",
   sizes="
          (max-width:480px) 100px,
          (min-width:481px) and (max-width:768px) 300px,
          (min-width:769px) and (max-width:979px) 475px,
          (min-width:980px) 940px",
   style="display: block;}">
</img>
```

You can exercise even more control over which image is loaded for specific view ports by using one of the following attributes:

```
xsmall_img= "url-to-xsmall-image";  // Image size should be 100 * 100 px
small_img = "url-to-small-image";   // Image size Should be 300 * 300 px
medium_img = "url-to-medium-image"; // Image size should be 475 * 475 px
large_img = "url-to-large-image"    // Image size should be 940 * 940 px
```

These attributes provide URLs to specific images that have been uploaded to your storefront's Media library (as opposed to the resized images generated by the `ccResizeImage` binding). Media library images are used when they are available and, when they are not, the

resized images created by `ccResizeImage` are used. In this example, the `AntiqueWoodChair_large.jpg` image will be used for the large view port while the other view ports will use the resized images generated by `ccResizeImage`. The `AntiqueWoodChair_large.jpg` image will be resized up to 300 x 300 pixels but no larger, as is dictated by the optional `large: '300,300'` attribute.

```
<img data-bind="ccResizeImage: {
            source: '/file/v2/products/AntiqueWoodChair_small.jpg',
            isSrcSetEnabled: true,
            large_img: "/file/v2/products/AntiqueWoodChair_large.jpg',
            large:'300,300',
            alt: 'Antique Wood Chair',
            errorSrc:'images/noImage.png',
            errorAlt:'No Image Found'}"></img>
```

Reserve a minimum height for an image

By default, the `ccResizeImage` binding reserves a minimum height on the page layout to accommodate an image before the image loads. This prevents the layout from shifting after the image loads. To make this possible, the `ccResizeImage` binding injects a `<div>` tag into the HTML that wraps around the `<img>` tag and sets it to the minimum height of the image. In general, this approach provides for a superior shopper experience, however, there are occasions where it may need to be disabled. For example, some browsers have problems with `<div>` tags placed inside `<td>` tags. For this reason, you have the option to disable the addition of the `<div>` tag by setting the `setMinHeightBeforeImageLoad` attribute to `false`, for example:

```
<img data-bind="ccResizeImage: {
            source: '/file/v2/products/AntiqueWoodChair_full.jpg',
            alt:'Antique Wood Chair',
            errorSrc:'images/noImage.png',
            errorAlt:'No Image Found',
            setMinHeightBeforeImageLoad:false}"> </img>
```

ccResizeImage attributes

The following table describes the attributes you can use with the `ccResizeImage` binding.

Attribute	Description
source	The image source URL.
large	The override dimensions for the `large` viewport, expressed as a comma-separated list of two values, the first for height and the second for width.
medium	The override dimensions for the `medium` viewport, expressed as a comma-separated list of two values, the first for height and the second for width.
small	The override dimensions for the `small` viewport, expressed as a comma-separated list of two values, the first for height and the second for width.

Attribute	Description
xsmall	The override dimensions for the `xsmall` viewport, expressed as a comma-separated list of two values, the first for height and the second for width.
size	The dimensions used if an override dimension has not been specified for the current viewport. The value for this attribute can be a comma-delimited list of two values, the first for height and the second for width, for example: `size: '50,50',` Alternatively, the value can be one of the following: `large`, `medium`, `small`, or `xsmall`. If one of these values is specified and an override dimension is provided for that same size, then that override dimension is used. For example, if `size: 'medium'` and `medium: '120,120'` are set and the image is being viewed on a viewport without a specific override dimension, the image will be sized to 120 x 120 pixels. If `size` is set to `large`, `medium`, `small`, or `xsmall` and no override dimension is provided for that same size, then the default dimensions are used, which are: `xsmall`: 100X100 `small`: 300X300 `medium`: 475X475 `large`: 940X940
alt	The alternative text for the image.
errorSrc	The error image URL.
errorAlt	The alternative text for the error image.
outputFormat	The format of the converted images. Only JPEG is supported. **Note:** GIF images cannot be resized or converted to JPEG as they may contain animation which is lost after resizing and conversion.
alphaChannelColor	The hexadecimal color code for the replacement color of the PNG alpha channel (default is white, FFFFFF).
quality	A number that lets you control the image resolution quality. The value of quality is a number from 0.0 (worst resolution but fastest load time) to 1.0 (best resolution but slowest load time). For example, you might want to reduce the resolution of product listing images to speed up image loading times. The default value of, `quality` is 1.0.

Attribute	Description
isSrcSetEnabled	Makes a set of variously sized images available to the browser and defines the conditions under which each one is loaded. See Use a srcset to specify the image to load for more details.
xsmall_img	A URL to an image in the Media library, used when the available width for displaying the image is less than 100 pixels. See Use a srcset to specify the image to load for more details.
small_img	A URL to an image in the Media library, used when the available width for displaying the image is less than 300 pixels. See Use a srcset to specify the image to load for more details.
medium_img	A URL to an image in the Media library, used when the available width for displaying the image is less than 475 pixels. See Use a srcset to specify the image to load for more details.
large_img	A URL to an image in the Media library, used when the available width for displaying the image is less than 940 pixels. See Use a srcset to specify the image to load for more details.
setMinHeightBeforeImageLoad	Reserves a minimum height on the page layout to accommodate an image before the image loads. This prevents the layout from shifting after the image loads. See Reserve a minimum height for an image for more details.
id	This parameter is used by the binding to create a new wrapper <div> with a unique ID. This prevents all wrappers from using the same ID.

Understand the image resizing REST APIs

Commerce Service REST APIs allow you to resize images displayed on your store while optimizing load times and maintaining image quality.

Note: See Use the REST APIs for information you need to know before using the REST APIs.

View image file names and paths

To view file names and paths for uploaded images for a product or collection, issue a GET request to the /ccstore/v1/products/{id} or /ccstore/v1/collections/{id} endpoint. For example:

```
GET /ccstore/v1/products/prod10007 HTTP/1.1
 Authorization: Bearer <access_token>
```

The following portion of the sample response shows the URLs of the product's large image:

```
  "primaryLargeImageURL":"/ccstore/v1/images/?source=/file/
v7875483805069966233/products/
APP_WeekendTrouser_large.jpg&height=940&width=940",
```

The path returned here is the full path to the location of the image file. (See Manage Media for Your Store for more information about the locations of uploaded images.)

Resize an image via the REST API

To resize an existing image, you use the Store API, which provides access to the storefront. Issue a GET request to the /ccstore/v1/images endpoint. The following table describes the query parameters you specify in the request.

Property	Description
source	(Required) String that specifies the fully qualified URL for the image to resize. This is returned in the response to GET /ccstore/v1/products/{id} or GET /ccstore/v1/collections/{id}.
height	(Required) The maximum height for the resized image, in pixels. If the request does not include either height or width, the source image is not resized.
width	(Required) The maximum width for the resized image, in pixels. If the request does not include either height or width, the source image is not resized.
quality	A number that lets you control the image resolution quality. The value of quality is a number from 0.0 (worst resolution but fastest load time) to 1.0 (best resolution but slowest load time). For example, you might want to reduce the resolution of product listing images to speed up image loading times. The default value of quality is 1.0.
outputFormat	The format of the converted images. Only JPEG is supported.
alphaChannelColor	The hexadecimal color code for the replacement color of the PNG alpha channel (default is white, FFFFFF).

For example, the following request resizes a product image to 500x500:

```
GET /ccstore/v1/images/?source=/file/v7875483805069966233/products
        /APP_WeekendTrouser_full.jpg&height=500&width=500
```

Manage image caching

Commerce images can be cached by the browser and the content delivery network (CDN) by default. Caching is desirable for performance reasons, but you must ensure that it does not lead to out-of-date images displaying on your pages.

On a production site, calls to Commerce endpoints that refer to images, such as product, SKU, and collection endpoints, respond with image URLs that include a sequence of numbers. For example:

```
/file/v7875483805069966233/products/APP_WeekendTrouser_full.jpg
```

In this example, `v7875483805069966233` is a checksum of the file. Commerce updates this value automatically when the image is changed and published. The checksum ensures that an earlier version of the image that has been cached in the browser or CDN is not displayed.

If you generate your own image URLs (for example, if you use images that are not explicitly associated with products, SKUs, or collections), then when you modify an image, its URL typically does not change. As a result, an out-of-date cached version of the image may continue to be displayed. Similarly, if you reuse an existing image URL for a new image, the old image may be displayed instead.

To prevent these issues, you should use a cache-busting strategy to ensure only up-to-date images are displayed. For example, the URLs you generate could incorporate a timestamp that gets updated each time the image is updated. You can implement this by using the `lastpublishtimestamp` value from the `ccRestClient` module. Each time an image loaded with this parameter is modified and published, the value of the parameter changes, ensuring that previously cached versions are not displayed. Alternatively, you can get the full pathname for an image, including the checksum value, by calling the `getFileURLs` endpoint in the Store API. For example:

```
PUT /ccstore/v1/files/urlMappings  HTTP/1.1

{
  "filePaths": [
    "/products/cat5cable_LARGE.jpg"
  ]
}
```

The response includes the complete pathname, including the current checksum value:

```
{
    "/products/cat5cable_LARGE.jpg": "http://myserver.example.com:7002/
file/v2441433713947926995/products/cat5cable_LARGE.jpg",
    "links": [
        {
            "rel": "self",
            "href": "http://myserver.example.com:7002/ccstore/v1/files/
urlMappings"
        }
```

```
                ]
        }
```

10
Manage Storefront Event Notification

The `PubSub` library allows you to manage your storefront notifications.

Commerce includes two mechanisms for storefront event notification:

- The `PubSub` library provides a way for code to publish and subscribe to a set of "global" messages. That is, any code in the application can publish messages about an event and any code can listen for those messages.

- The `event-dispatcher` module provides the ability to listen for events triggered by a specific instance of an object.

Understand the `PubSub` library

The `PubSub` library is a publishing and subscription system based on `jQuery.Callbacks` functions.

With the `PubSub` library, messages can be published by any object in the system when events happen and subscribers can listen for those messages and perform additional tasks as needed. The backbone of the `PubSub` library is a list of topics identified by `ID`, for example, `PAGE_READY` and `CART_ADD`. Publishers (which are typically Commerce widgets) publish messages to these topics when events happen. Subscribers to a given topic receive the messages published to that topic along with supporting data.

You can create custom topics and add them to the `PubSub` library; however, these custom topics may only be used by custom widgets that you build. Out-of-the-box widgets will not have awareness of or access to custom `PubSub` topics.

Include the pubsub dependency

In order to use the `PubSub` library from within a widget, you need to include `pubsub` as a dependency in the widget's JavaScript module with a statement similar to the following:

```
//-------------------------------------------
 // DEPENDENCIES
 //-------------------------------------------
  ['pubsub'],
```

Subscribe to a topic

You use the `subscribe` function to subscribe your callback function to a topic.

The data that has been published to a topic is then passed to the callback function.

There are two ways to use the `subcribe()` function. You can either provide the callback function's name or you can provide code for the callback function in-line. In this example,

whenever the `PAGE_CHANGED` topic has a message published to it, the `getPageUrlData()` function is called.

```
$.Topic(pubsub.topicNames.PAGE_CHANGED).subscribe(
            widget.getPageUrlData);
```

In this example, whenever the `PAGE_CHANGED` topic has a message published to it, the anonymous, in-line function is executed.

```
$.Topic(pubsub.topicNames.PAGE_CHANGED).subscribe(
     function(value){
        widget.isDisplayErrorPins(false);
     }
 );
```

Note that the value for `this` in your callback function may vary, depending on whether the message that triggered the callback function was published with context or not, so you should save the current value of `this` if you need to guarantee that it remains constant. See Publish messages for more details on publishing messages with context.

To unsubscribe from a topic, use the `unsubscribe()` function, for example:

```
$.Topic(pubsub.topicNames.PAGE_LAYOUT_LOADED).unsubscribe(
            widget.resetOrderDetails);
```

Publish messages

Two functions, `publish()` and `publishWith()`, are used to publish messages to topics.

The `publish()` function takes one parameter, the object you want to publish. In the example below, the `publish()` function will send the data object to callback functions that are subscribed to the `PAGE_READY` topic.

```
$.Topic(PubSub.topicNames.PAGE_READY).publish(data);
```

In some cases, you may need to control the context in which subscribers receive the published data. When a publisher provides context, it is providing the subscriber with access to data or operations the subscriber needs to do its job. To control context, use the `publishWith()` function instead of the `publish()` function. The `publishWith()` function takes two parameters; the first is the object to be used as this in the subscriber's callback function, the second is the object you want to publish. For example, the following code publishes the billing address as the context:

```
$.Topic(pubsub.topicNames.CHECKOUT_BILLING_ADDRESS).publishWith(
        widget.billingAddress(), [{
            message: "success"
        }]);
```

This allows the subscriber to update the billing address with the value of this:

```
$.Topic(pubsub.topicNames.CHECKOUT_BILLING_ADDRESS).subscribe(
          function() {
              self.billingAddress(this);
          }
);
```

Create new topics

The `PubSub` library includes a number of topics out of the box and these topics are described later in this section.

If you need to create a custom topic, you can do so using the `topic()` function, passing in the ID of your topic. The `topic()` function returns an existing topic if the ID already exists. If no topics exist that match the passed ID, a new topic is created and returned.

Custom topic objects have the publish, `publishWith`, subscribe, and unsubscribe functions. Note that custom topics are available for subscription by custom widgets only; default widgets have no knowledge of or access to custom topics. They are used exclusively to let one custom widget know about an event that has happened in another custom widget. Oracle recommends that any custom topics you create include a merchant-specific prefix in the topic ID to avoid conflicts with default topics.

The following example shows the creation of a topic named `MY_TOPIC` in a custom widget:

```
$.Topic("MY_TOPIC.memory").publish("Message is here") ;
```

This example shows a subscription to `MY_TOPIC` in another custom widget:

```
$.Topic("MY_TOPIC.memory ").subscribe(function(message)
 {
    console.log("Message is: " + message);
 });
```

Note that this custom topic makes use of the `.memory` suffix, which enables memory for the topic. Typically, for a callback function to be triggered, it has to be subscribed to a topic before any messages are published to that topic. The `.memory` suffix allows a callback function to be triggered for the most recently published message even if the function has subscribed to the topic after the message was published.

PubSub topics

There are topics included by default with the `PubSub` system.

Topics that have memory enabled are marked accordingly. As a reminder, having memory enabled for a topic allows a callback function to be triggered for the most recently published message even if the function has subscribed to the topic after the message was published. See Create new topics for more information.

CART_ADD_SUCCESS

A message is published to this topic whenever a product is successfully added to the cart.

```
$.Topic(pubsub.topicNames.CART_ADD_SUCCESS).publish(product);
```

Arguments

`product`: A JSON object that includes data for the added product, for example:

```
{
            "primaryFullImageURL" : "/ccstore/v1/images/?source=/
file/v2/products/
                mymovie_LARGE.jpg",
            "smallImageURLs" : ["/ccstore/v1/images/?source=/file/v2/
products/
                mymovie_LARGE.jpg&height=300&width=300"],
            "orderLimit" : null,
            "shippingSurcharges" : null,
            "salePrices" : null,
            "type" : null,
            "listPrices" : null,
            "primaryImageAltText" : "My Movie",
            "height" : null,
            "shippingSurcharge" : null,
            "listPrice" : 21.99,
            "description" : "A great movie that you should not miss.",
            "fullImageURLs" : ["/ccstore/v1/images/?source=/file/v2/
products/
                mymovie_LARGE.jpg"],
            "longDescription" : null,
            "unitOfMeasure" : null,
            "primaryMediumImageURL" : "/ccstore/v1/images/?source=/
file/v2/products/
                mymovie_LARGE.jpg&height=475&width=475",
            "CountryOfOrigin" : "US",
            "mediumImageURLs" : ["/ccstore/v1/images/?source=/file/v2/
products/
                mymovie_LARGE.jpg&height=475&width=475"],
            "primarySourceImageURL" : "/ccstore/v1/images/?source=/
file/v2/products/
                mymovie_LARGE.jpg&height=300&width=300",
            "seoKeywordsDerived" : "My Movie,Psychological
Thrillers,Blockbuster,
                All Products,Thrillers-Clearance",
            "width" : null,
            "primaryThumbImageURL" : "/ccstore/v1/images/?source=/
file/v2/products/
                mymovie_LARGE.jpg&height=100&width=100",
            "primarySmallImageURL" : "/ccstore/v1/images/?source=/
file/v2/products/
                mymovie_LARGE.jpg&height=300&width=300",
            "relatedMediaContent" : [],
```

```
                "active" : true,
                "largeImageURLs" : ["/ccstore/v1/images/?source=/file/v2/
products/
                    mymovie_LARGE.jpg&height=940&width=940"],
                "salePrice" : null,
                "fractionalQuantitiesAllowed" : false,
                "primaryLargeImageURL" : "/ccstore/v1/images/?source=/file/v2/
products/
                    mymovie_LARGE.jpg&height=940&width=940",
                "relatedProducts" : null,
                "weight" : null,
                "parentCategory" : null,
                "avgCustRating" : 4.5,
                "productImagesMetadata" : [{}
                ],
                "id" : "Product_36Exy",
                "sourceImageURLs" : ["/ccstore/v1/images/?source=/file/v2/
products/
                    mymovie_LARGE.jpg&height=300&width=300"],
                "seoMetaInfo" : null,
                "variantValuesOrder" : {},
                "length" : null,
                "relatedArticles" : [],
                "defaultProductListingSku" : null,
                "seoDescriptionDerived" : "My Movie,A great movie that you
should not miss.",
                "parentCategories" : [{
                    "id" : "cat70011",
                    "categoryImages" : [],
                    "route" : "/thrillers-clearance/category/cat70011",
                    "description" : null,
                    "longDescription" : null,
                    "active" : true,
                    "displayName" : "Thrillers-Clearance",
                    "repositoryId" : "cat70011"
                    }
                ],
                "childSKUs" : [{
                    "salePrices" : null,
                    "primaryFullImageURL" : null,
                    "primaryLargeImageURL" : null,
                    "smallImageURLs" : [],
                    "thumbnailImage" : null,
                    "listPrices" : null,
                    "sourceImageURLs" : [],
                    "listPrice" : 21.99,
                    "fullImageURLs" : [],
                    "productListingSku" : null,
                    "quantity" : null,
                    "smallImage" : null,
                    "unitOfMeasure" : null,
                    "primaryMediumImageURL" : null,
                    "mediumImageURLs" : [],
                    "primarySourceImageURL" : null,
                    "largeImage" : null,
```

```
                    "primaryThumbImageURL" : null,
                    "primarySmallImageURL" : null,
                    "repositoryId" : "Sku_36Fxy",
                    "thumbImageURLs" : [],
                    "salePriceEndDate" : null,
                    "dynamicPropertyMapLong" : {},
                    "images" : [],
                    "largeImageURLs" : [],
                    "salePrice" : null,
                    "salePriceStartDate" : null,
                    "fractionalQuantitiesAllowed" : false
                }
            ],
            "repositoryId" : "Product_36Exy",
            "thumbImageURLs" : ["/ccstore/v1/images/?source=/file/v2/
products/
                mymovie_LARGE.jpg&height=100&width=100"],
            "primaryImageTitle" : "My Movie",
            "route" : "/my-movie/product/Product_36Exy",
            "brand" : null,     "seoUrlSlugDerived" : "My Movie",
            "displayName" : "My Movie",
            "seoTitleDerived" : "My Movie",
            "selectedOptions" : [],
            "orderQuantity" : 1
        }
```

Memory enabled

No

CART_READY

A message is published to this topic when the cart is loaded with items, either from local storage for an anonymous shopper or from the persistent cart for a registered shopper, and is ready for use. This topic allows subscribers to access the cart items and their associated product details.

```
$.Topic(PubSub.topicNames.CART_READY).publish(cart);
```

Arguments

`cart`: The current `CartViewModel`. See the View Model JSDoc for Commerce for details on what this view model contains.

Memory enabled

No

CART_UPDATED

A message is published to this topic when the cart is updated with the latest pricing response and saved to local storage.

```
$.Topic(pubsub.topicNames.CART_UPDATED).publish(cart);
```

Arguments

`cart`: The current `CartViewModel`. See the View Model JSDoc for Commerce for details on what this view model contains.

Memory enabled

No

HISTORY_PUSH_STATE

A message is published to this topic when a shopper navigates from one page to another or refreshes the page.

```
$.Topic(PubSub.topicNames.HISTORY_PUSH_STATE).publish(path);
```

Arguments

`path`: The path for the page being navigated to or refreshed, for example, `/home`, `/cart`, `/checkout`, `/profile`, and so on.

Memory enabled

Yes

ONERROR_EXCEPTION_HANDLER

A message is published to this topic when a runtime error occurs.

```
$.Topic(PubSub.topicNames.ONERROR_EXCEPTION_HANDLER).publish(
                errorMessage,  errorUrl, errorLineNumber);
```

Arguments

`errorMessage`: Contains the error message, for example:

```
"Uncaught Error: Script error for: //abc.com/EEAdmin/js/ee/js/
 atgsvcs-test.js http://requirejs.org/docs/errors.html#scripterror"
```

`errorUrl`: The URL value when the error occurred, for example:

```
http://localhost:8080/shared/js/libs/oraclejet/libs/require/require.js
```

`errorLineNumber`: The line of code the error occurred in.

Memory enabled

No

ORDER_SUBMISSION_SUCCESS

A message is published to this topic when the submission of an order is successfully completed.

```
$.Topic(PubSub.topicNames.ORDER_SUBMISSION_SUCCESS).publish(orderDetails);
```

Arguments

`orderDetails`: A JSON array that includes data for the submitted order, for example:

```
[{
    message : " success",
    id : "o30501",
    uuid : " 14f03075-1376-46dd-9e2c-1448255efa8f"
 }]
```

The properties in the JSON object include:

- `message`: This property has a value of success to indicate the order was submitted

- `successfully.id`: A system-generated ID for the order in the Order

- `repository.uuid`: A system-generated universally unique identifier for the `order`.

Memory enabled

No

PAGE_CHANGED, PAGE_VIEW_CHANGED

A message is published to these topics when a page has a view change or a context change. A view change means that the page type has changed, for example, from `/home` to `/checkout`. In this case, both the layout and the data change. A context change occurs when the page type remains the same but the data the page contains changes, for example, `/product/prod1` to `/product/prod2`.

```
$.Topic(PubSub.topicNames.PAGE_CHANGED).publish(pageEventData);
```

Arguments

`pageEventData`: A JavaScript object that includes data related to the page change event, for example:

```
{
    pageId:"category"
    contextId:"cat60041"
    seoslug:"controllers"
    previousContextId:"Product_36Exy"
 }
```

The properties in the JavaScript object include:

- `pageId`: An identifier for the page requested that represents either the page type (for example, `product`, `collection`, `home`, and so on) or, for article pages, the page address that has been assigned in the Layout settings (for example, `aboutUs`, `shipping`, or `returns`, are all pages of type article).

- `contextId`: For contextual pages, a `contextId` is included and it may be a product ID, category ID, an order ID, or an order confirmation ID. To determine the ID for a specific product or category, you can view the item's details on the Catalog page in Commerce. Order and order confirmation IDs are system-generated and cannot be known beforehand but you can check for their existence in the JSON object.

- `seoslug`: An SEO-friendly, internally-generated name for the product or category, based on the product or category name. This property only has a value when changing to a product or category page.

- `previousContextId`: The context ID, if one exists, for the page that was rendered before the `PAGE_CHANGED` event occurred

Memory enabled

Yes

`PAGE_VIEW_CHANGED` is the same as `PAGE_CHANGED`, except that `PAGE_VIEW_CHANGED` reloads only the layout, while `PAGE_CHANGED` reloads the entire page.

PAGE_READY

A message is published to this topic when all the regions and widgets on a page are loaded.

```
$.Topic(PubSub.topicNames.PAGE_READY).publish(pageEventData);
```

Arguments

`pageEventData`: A JavaScript object that includes data about the loaded page, for example:

```
{
    pageId:"category",
    contextId:"cat60041",
    seoslug:" psychological-thrillers"
    previousContextId: null,
    pageRepositoryId: "categoryPage"
}
```

The properties in the JavaScript object include:

- `pageId`: An identifier for the page requested that represents either the page type (for example, `product`, `collection`, `home`, and so on) or, for article pages, the page address that has been assigned in the Layout settings (for example, `aboutUs`, `shipping`, or `returns`, are all pages of type article).

- `contextId`: For contextual pages, a `contextId` is included and it may be a product ID, category ID, an order ID, or an order confirmation ID. To determine the ID for a specific product or category, you can view the item's details on the Catalog page in Commerce. Order and order confirmation IDs are system-generated and cannot be known beforehand but you can check for their existence in the JSON object.

- `seoslug`: An SEO-friendly, internally-generated name for the product or category, based on the product or category name. This property only has a value when changing to a product or category page.

- `previousContextId`: The context ID, if one exists, for the page that was rendered before the `PAGE_CHANGED` event occurred.

- `pageRepositoryId`: The identifier for the page in the page repository. For the out of the box pages, this value can be `homePage`, `categoryPage`, `profilePage`, `checkoutPageWithGiftCard`, `userSpacesPage`, `orderHistoryPage`, `orderDetailsPage`, or `userSpacesSettingsPage`. For any additional pages created in Commerce, a system-generated ID is created, for example, pa100001.

Memory enabled

Yes

PRODUCT_VIEWED

A message is published to this topic when the shopper views a product.

```
$.Topic(pubsub.topicNames.PRODUCT_VIEWED).publish(product);
```

Arguments

`product`: The current `ProductViewModel`. See the View Model JSDoc for Commerce for details on what this view model contains.

Memory enabled

Yes

SEARCH_RESULTS_UPDATED

A message is published to this topic when search results are retrieved or when an error is returned while searching for a product. If search results are retrieved successfully, those results are published with the message. If an error occurs, the error is published with the message.

```
$.Topic(pubsub.topicNames.SEARCH_RESULTS_UPDATED).publishWith(searchRes
ults, message);
```

Arguments

`searchResults`: The contents of this argument vary depending on whether the search was successful or not. If the search was successful, the argument consists of a JSON object that includes the search results and meta-data about the search, for example:

```
{
    searchResults : Array[5],
    isNewSearch:false,
    navigation : Array[5],
    pageCount : 15,
    pagingActionTemplate : JSONObject,
    recordOffSet : 0,
    recordsPerPage : 5,
    searchAdjustments :JSONObject,
    totalRecordsFound : 74
    breadcrumbs:JSONObject
 }
```

The properties in the successful search JSON object include:

*   `searchResults`: An array that contains the search results. The following code sample shows what an item in that array might look like:

```
{
            "product.category" : ["Cameras and Camcorders"],
```

          "product.language" : ["en"],
          "record.type" : ["camera", "sku-camera"],
          "sku.listingId" : ["camera_1"],
          "product.priceRange" : ["JPY0 - JPY9"],
          "sku.activePrice" : ["3.990000"],
          "product.creationDate" : ["1466057558000"],
          "sku.url" : ["atgrep:/ProductCatalog/sku-camera/
             camerasku_1_55?_product=camera_1&locale=en&priceListPair=
             salePrices_listPrices"],
          "product.priceListPair" : ["salePrices_listPrices"],
          "product.active" : ["1"],
          "sku.salePrice" : ["3.990000"],
          "sku.creationDate" : ["1466057558000"],
          "record.source" : ["ProductCatalog"],
          "product.url" : ["atgrep:/ProductCatalog/sku-camera/
             camerasku_1_55?_product=camera_1&locale=en&priceListPair=
             salePrices_listPrices"],
          "sku.availabilityStatus" : ["OUTOFSTOCK"],
          "parentCategory.displayName" : ["Cameras and Camcorders"],
          "Endeca.Document.Language" : ["en"],
          "product.daysAvailable" : ["0"],
          "record.type.raw" : ["camera", "sku-camera"],
          "sku.baseUrl" : ["atgrep:/ProductCatalog/sku-camera/
             camerasku_1_55"],
          "product.dateAvailable" : ["1466057558000"],
          "sku.repositoryId" : ["camerasku_1_55"],
          "product.baseUrl" : ["atgrep:/ProductCatalog/sku-camera/
             camerasku_1_55"],
          "sku.listPrice" : ["399.990000"],
          "sku.onSale" : ["0"],
          "record.id" : ["sku-camera-camerasku__1__55..camera__1.en.
             salePrices__listPrices"],
          "product.route" : ["/digital-compact-system-camera/product/
             camera_1"],
          "product.longDescription" : ["This digital compact system camera
             14.1-megapixel,\n   4/3 Live MOS sensor allows you to
             capture sharp photos and record high-definition video
footage with

             up\n   to 1920 x 1080 resolution. Built-in Wi-Fi lets
             you easily share stored files.\n   "],
          "sku.listingOptionIndex" : ["10000"],
          "id" : ["camera_1"],
          "repositoryId" : ["camera_1"],
          "displayName" : "Digital Compact System Camera",
          "shippingSurcharge" : 0,
          "listPrice" : 399.99,
          "salePrice" : null,
          "route" : "/digital-compact-system-camera/product/camera_1",
          "primaryImageAltText" : ["Digital Compact System Camera"],
          "primaryImageTitle" : ["Digital Compact System Camera"],
          "primaryLargeImageURL" : "/ccstore/v1/images/?source=
             /file/v2/products/OM_D_E_M10_LARGE.jpg&height=940&width=940",
          "primarySmallImageURL" : "/ccstore/v1/images/?source=
             /file/v2/products/OM_D_E_M10_LARGE.jpg&height=300&width=300",
          "primaryThumbImageURL" : "/ccstore/v1/images/?source=

```
                       /file/v2/products/
OM_D_E_M10_LARGE.jpg&height=100&width=100",
                "primaryMediumImageURL" : "/ccstore/v1/images/?source=
                       /file/v2/products/
OM_D_E_M10_LARGE.jpg&height=475&width=475",
                "primaryFullImageURL" : "/ccstore/v1/images/?source=
                       /file/v2/products/OM_D_E_M10_LARGE.jpg",
                "childSKUs" : [{
                       "salePrice" : "3.990000",
                       "listPrice" : "399.990000",
                       "repositoryId" : ["camerasku_1_55"],
                       "largeImage" : {
                          "url" : ""              },
                       "smallImage" : {
                          "url" : ""              },
                       "thumbnailImage" : {
                          "url" : ""
                        }
                     }
                   ],
                "maxActivePrice" : "399.990000",
                "minActivePrice" : "3.990000"
}
```

- isNewSearch: A Boolean value that specifies whether this is a new search or not.

- navigation: An array of refinements elements, each of which contains the refinements available for a given dimension. These refinements are used to present the shopper with a set of refinement links that she can use to further refine the current search results. For example, the following JSON sample shows an excerpt of the refinements available for the product.category dimension:

```
"refinements": [
      {
             "multiSelect": false,
             "navigationState":
                 "?
N=4058976224&Nr=AND%28product.priceListPair%3AsalePrices_listPrices

%2Cproduct.language%3Aen%29&Nrpp=12&Ntl=en&Ntt=cam&language=en&

searchType=simple&visitId=-14ddf4e9%3A156217da656%3A-582e-10.170.102.15
0&

visitorId=1298EspxGMOA_YEnEkrCS9sDr1JYUdQ6jpex06ADSHQzGuU30A4",
             "contentPath": "\/guidedsearch",
             "count": 2,
             "@class": "com.endeca.infront.cartridge.model.Refinement",
             "siteRootPath": "\/pages\/Default\/services",
             "siteState": {
               "contentPath": null,
               "siteId": "@error:siteNotFound",
               "@class": "com.endeca.infront.site.model.SiteState",
               "siteDisplayName": null,
               "validSite": false,
```

```
            "properties": {
            },
            "siteDefinition": null,
            "matchedUrlPattern": null      },
        "label": "All Products",
        "properties": {
          "dimval.prop.category.ancestorCatalogIds": "cloudLakeCatalog",
          "dimval.prop.category.rootCatalogId": "cloudLakeCatalog",
          "dimval.prop.displayName_en": "All Products",
          "dimval.prop.category.repositoryId": "AllProdPagination",
          "dimval.prop.category.catalogs.repositoryId":
            "cloudCatalog,cloudLakeCatalog",
            "record.id":
            "atgrep:\/ProductCatalog\/category\/AllProdPagination?
             categoryPath=\/AllProdPagination",
          "DGraph.Spec": "AllProdPagination"
        }
      },
      {
        "multiSelect": false,
        "navigationState":
          "?
N=3881554706&Nr=AND%28product.priceListPair%3AsalePrices_listPrices%

2Cproduct.language%3Aen%29&Nrpp=12&Ntl=en&Ntt=cam&language=en&

searchType=simple&visitId=-14ddf4e9%3A156217da656%3A-582e-10.170.102.150&

visitorId=1298EspxGMOA_YEnEkrCS9sDr1JYUdQ6jpex06ADSHQzGuU30A4",
        "contentPath": "\/guidedsearch",
        "count": 1,
        "@class": "com.endeca.infront.cartridge.model.Refinement",
        "siteRootPath": "\/pages\/Default\/services",
        "siteState": {
          "contentPath": null,
          "siteId": "@error:siteNotFound",
          "@class": "com.endeca.infront.site.model.SiteState",
          "siteDisplayName": null,
          "validSite": false,
          "properties": {
          },
          "siteDefinition": null,
          "matchedUrlPattern": null
        },
        "label": "Blockbuster",
        "properties": {
          "dimval.prop.category.ancestorCatalogIds": "cloudLakeCatalog",
          "dimval.prop.category.rootCatalogId": "cloudLakeCatalog",
          "dimval.prop.displayName_en": "Blockbuster",
          "dimval.prop.category.repositoryId": "topLeaf",
          "dimval.prop.category.catalogs.repositoryId":
            "cloudCatalog,SiteContextDummyCatalog,ClassicalMoviesCatalog,
             cloudLakeCatalog",
          "record.id": "atgrep:\/ProductCatalog\/category\/topLeaf?
            categoryPath=\/topLeaf",
```

```
            "DGraph.Spec": "topLeaf"
          }
        },
      …Additional refinements here…
      ],
      "multiSelect": false,
      "@type": "RefinementMenu",
      "name": "product.category",
      "ancestors": [
      ],
      "displayName": "Category",
      "dimensionName": "product.category",
  "whyPrecedenceRuleFired": null
  },
```

- pageCount: The total number of pages of records. Each page contains the number of records specified by recordsPerPage.

- pagingActionTemplate: A JSON object that returns data used to retrieve the next page of search results.

```
"pagingActionTemplate" : {
      "navigationState" :
              "?&Ntl=en&Ntt=camera&language=en&searchType=simple",
      "contentPath" : "/guidedsearch",
      "@class" :
              "com.endeca.infront.cartridge.model.NavigationAction",
      "siteRootPath" : "/pages/Default/services",
      "siteState" : {
              "contentPath" : null,
              "siteId" : "@error:siteNotFound",
              "@class" : "com.endeca.infront.site.model.SiteState",
              "siteDisplayName" : null,
              "validSite" : false,
              "properties" : {},
              "siteDefinition" : null,
              "matchedUrlPattern" : null
      },
      "label" : null
  }
```

- recordOffset: Identifies the page being viewed. For example, if you have 12 records per page and recordOffset = 12, then page 2 is being viewed.

- recordsPerPage: The number of records to show per page, used for pagination controls.

- searchAdjustments: A JSON object that contains information about any adjustments made to the original search terms.

  – @type: Identifies the search adjustments portion of the JSON response.

  – originalSearchTerms: The original search terms as entered by the shopper, including any wildcards.

  – `originalTerms`: The original search terms entered by the shopper, minus any
    wildcards.

- `totalRecordsFound`: The total number of records that match the search criteria.

- `breadcrumbs`: A JSON object that provides details on the user actions that led to the
  current navigation state, for example:

```
"breadcrumbs" : {
        "removeAllAction" : {
            "navigationState" : "?Ntl=en&language=en&searchType=simple&
                visitId=null&visitorId=null",
            "contentPath" : "/guidedsearch",
            "@class" :
                "com.endeca.infront.cartridge.model.NavigationAction",
            "siteRootPath" : "/pages/Default/services",
            "siteState" : {
                "contentPath" : null,
                "siteId" : "@error:siteNotFound",
                "@class" : "com.endeca.infront.site.model.SiteState",
                "siteDisplayName" : null,
                "validSite" : false,
                "properties" : {},
                "siteDefinition" : null,
                "matchedUrlPattern" : null        },
            "label" : null
        },
        "refinementCrumbs" : [],
        "geoFilterCrumb" : null,
        "@type" : "Breadcrumbs",
        "endeca:auditInfo" : {
            "ecr:innerPath" : "breadcrumbs",
            "ecr:resourcePath" : "/pages/Default/services/guidedsearch"
        },
        "searchCrumbs" : [{
            "@class" :
                    "com.endeca.infront.cartridge.model.SearchBreadcrumb",
                "terms" : "camera",
                "removeAction" : {
                    "navigationState" :
                        "?
Nr=AND%28product.priceListPair%3AsalePrices_listPrices%
                        2Cproduct.language%3Aen%29&Nrpp=12&Ntl=en&language=en&
                        searchType=simple&visitId=null&visitorId=null",
                    "contentPath" : "/guidedsearch",
                    "@class" :
                     "com.endeca.infront.cartridge.model.NavigationAction",
                    "siteRootPath" : "/pages/Default/services",
                    "siteState" : {
                        "contentPath" : null,
                        "siteId" : "@error:siteNotFound",
                        "@class" :
                            "com.endeca.infront.site.model.SiteState",
                        "siteDisplayName" : null,
                        "validSite" : false,
                        "properties" : {},
```

```
                        "siteDefinition" : null,
                        "matchedUrlPattern" : null
                    },
                    "label" : null
                },
                "correctedTerms" : null,
                "key" : "All",
                "matchMode" : "allpartial"
            }
        ],
        "rangeFilterCrumbs" : []
}
```

When the search has failed, the `searchResults` object contains details about the error, including:

- `message`: The runtime error message that indicates that the search has failed.

- `status`: Any HTTP status codes associated with the failed request.

- `errorCode`: An Commerce error code, which may be

```
GET_SEARCH_INTERNAL_ERROR = 31040,
SEARCH_ADMIN_INTERNAL_ERROR = 31041,
 or
SEARCH_ADMIN_INDEXING_INTERNAL_ERROR = 31042.
```

For example:

```
{
    "message":"Unable to perform a search at this time",
    "status":"500",
    "errorCode":"31040"
 }
```

`message`: The second argument returned for the `SEARCH_RESULTS_UPDATED` topic contains a JSON array that includes either a success or failure message and the `SearchViewModel` object, for example:

```
[{
    message: "success",
    requestor: SearchViewModel
}]
```

In the case of a failed search, this message property has a value of fail.

```
[{"message":"fail"}]
```

Memory enabled

Yes

SKU_SELECTED

A message is published to this topic when a shopper selects all the variant options for a product, thereby identifying a specific SKU.

```
$.Topic(pubsub.topicNames.SKU_SELECTED).publish(product, selectedSku,
variantOptions);
```

Arguments

`product`: The current `ProductViewModel`. See the View Model JSDoc for Commerce for details on what this view model contains.

`selectedSku`: A JSON object that includes data for the selected SKU, for example:

```
selectedSku = {
     "salePrices" : null,
     "resolution" : "14.1-megapixels",
     "listPrice" : 399.99,
     "quantity" : 5,
     "repositoryId" : "camerasku_1_18",
     "color" : "Black",
     "dynamicPropertyMapLong" : {
                "sku-camera_resolution" : 0,
                "sku-camera_color" : 8
     },
     "salePrice" : 399.99,
     "salePriceStartDate" : null,
     "fractionalQuantitiesAllowed" : false
 }
```

`variantOptions`: An array whose items reflect the variant options (such as color, size, finish, resolution, and so on) that can be selected for the SKU, along with any currently selected variant options. For example, the following JSON sample shows the `color` and `resolution` variants returned for the camera selected above.

```
[
  {
     "optionDisplayName" : "color",
     "parent" : {
        "basePath" : "/",
                "deferredInit" : {},
                "prevHistoryLength" : 7,
                "newHistoryLength" : 8,
                "localeSubscription" : {
                    "R" : true
                },
                "WIDGET_ID" : "productDetails",
                "imgMetadata" : [{}
                ],
                "firstTimeRender" : true
            },
            "optionId" : "sku-camera_color",
```

```
                "actualOptionId" : "color",
                "optionCaption" : "Select color ...",
                "selectedOptionValue" : {
                  "key" : "Antique Brass",
                  "value" : 0
                },
                "originalOptionValues":[
                  {
                    "key" : "Antique Brass",
                    "value" : 0
                  },
                  {
                    "key" : "Apricot",
                    "value" : 1
                  },
                  {
                    "key" : "Aquamarine",
                    "value" : 2
                  },
                ]
              },
              {
                "optionDisplayName" : "resolution",
                "parent" : {
                  "basePath" : "/",
                  "deferredInit" : {},
                  "prevHistoryLength" : 7,
                  "newHistoryLength" : 8,
                  "localeSubscription" : {
                    "R" : true
                  },
                  "WIDGET_ID" : "productDetails",
                  "imgMetadata" : [{}
                  ],
                  "firstTimeRender" : true
                },
                "optionId" : "sku-camera_resolution",
                "actualOptionId" : "resolution",
                "optionCaption" : "Select resolution ...",
                "selectedOptionValue" : {
                  "key" : "14.1-megapixels",
                  "value" : 0
                },
                "originalOptionValues":[
                  {
                    "key" : "14.1-megapixels",
                    "value" : 0
                  },
                  {
                    "key" : "24.3-megapixels",
                    "value" : 1
                  },
                ]
              }
            }
          ]
```

Memory enabled

No

USER_LOGIN_SUCCESSFUL

A message is published to this topic when the shopper has successfully logged in. You can subscribe to this topic to introduce logic that is specific to logged-in shoppers.

```
$.Topic(pubSub.topicNames.USER_LOGIN_SUCCESSFUL).publish();
```

Arguments

None

Memory enabled

No

USER_LOGOUT_SUCCESSFUL

A message is published to this topic when the shopper successfully logs out.

```
$.Topic(pubSub.topicNames.USER_LOGOUT_SUCCESSFUL).publish(message);
```

Arguments

`message`: A JSON array that includes a `message` property whose value is `success`.

```
[{"message":" success "}]
```

Memory enabled

No

USER_PASSWORD_EXPIRED

A message is published to this topic when the user password expires.

```
$.Topic(pubSub.topicNames.USER_PASSWORD_EXPIRED).publish();
```

Arguments

None

Memory enabled

No

USER_PROFILE_UPDATE_SUCCESSFUL

A message is published to this topic when the shopper's profile is updated successfully with a new address, email, or name.

```
$.Topic(pubSub.topicNames.USER_PROFILE_UPDATE_SUCCESSFUL).publish(profileData
);
```

Arguments

`profileData`: A JSON object that contains the updated user profile data, for example:

```
{
            "receiveEmail" : "yes",
            "lastName" : "Spencers",
            "locale" : "en_US",
            "contactBillingAddress" : "Address Object",
            "links" : [{
                "rel" : "self",
                "href" : http://localhost:9080/ccstore/v1/profiles/
current
                }
            ],
            "contactShippingAddress" : "Address Object",
            "repositoryId" : "se-570032",
            "parentOrganization" : null,
            "id" : "se-570032",
            "dynamicProperties" : [],
            "email" : "mspens@abc.com",
            "shippingAddresses" : ["Address Object1","Address
Object2"],
            "daytimeTelephoneNumber" : null,
            "secondaryAddresses" : {
                "Mom' s house" : "Address Object",
                "Work" : "Address Object",
                "Home" : "Address Object",
            },
            "firstName" : "Mark",
            "shippingAddress" : "Address Object"
}
```

Note that the `Address Object` references in the example above look similar to the following:

```
"contactBillingAddress" : {
            "lastName" : "Spencers",
            "postalCode" : "36130",
            "phoneNumber" : "555-555-1234",
            "county" : null,
            "state" : "AL",
            "address1" : "123 Main St",
            "address2" : null,
            "firstName" : "Mark",
            "companyName" : null,
            "repositoryId" : "se-990032",
            "city" : "Montgomery",
            "country" : "US"
},
```

Memory enabled

Yes

USER_SESSION_EXPIRED

A message is published to this topic when the user session expires. You can subscribe to this topic to clear any session-specific data once the session expires.

```
$.Topic(pubsub.topicNames.USER_SESSION_EXPIRED).publish();
```

Arguments

None

Memory enabled

Yes

Listen for messages from a particular object instance

The `PubSub` library provides a way for code to publish and subscribe to a set of "global" messages.

Any code in the application can publish messages to a topic and any code can listen for those messages. There may be a need, however, to listen for messages coming only from a specific instance of an object. To manage this use case, Commerce includes the `event-dispatcher` module. Any object created from a module that extends `event-dispatcher` can trigger events and allow other objects to register as listeners for those events.

The `event-dispatcher` module provides two functions, `trigger()` and `on()`, that facilitate event triggering and event listening, respectively. In the example below, the `ExampleDialog` view model extends `event-dispatcher` and defines two events that can be fired, `save-event` and `cancel-event`. To trigger the events, the view associated with `ExampleDialog` could call the `trigger()` functions as needed, for example, when the **Save** or **Cancel** button is clicked.

```
/**  * ExampleDialog.js  */
        define (["shared/ccLibs/event-dispatcher"],
function(EventDispatcher) {   "use strict";
        /**
         * Example of a module that extends EventDispatcher, enabling
instances of the
         * class to fire events to any listeners/event handlers registered.
         * @see EventDispatcher
         */
      function ExampleDialog() {
           // Invoke superclass constructor to inherit any instance
properties.
           EventDispatcher.call(this);
           this.p1 = "property 1 value";
           this.p2 = "property 2 value";
         }
        // Extend EventDispatcher so we can dispatch events.
        ExampleDialog.prototype = Object.create(EventDispatcher.prototype);
        ExampleDialog.prototype.constructor = ExampleDialog;
        // Define constants for any events we want to fire. These need
only be unique
        // within this module because corresponding listeners receive
```

```
events from a
          // given instance of this module.
          ExampleDialog.SAVE = "save-event";
          ExampleDialog.CANCEL = "cancel-event";
          /**
           * Corresponding view could be set up to call this method
when the
           * dialog's Save button is clicked, for example.
           */
          ExampleDialog.prototype.save = function() {
            // Fire SAVE event. Any additional arguments (beyond the
first) are
            // passed along to the event handler/listener.
            this.trigger(ExampleDialog.SAVE, this.p1);
          };
          /**
           * Corresponding view could be set up to call this method
when the
           * dialog's cancel button is clicked, for example.
           */
          ExampleDialog.prototype.cancel = function() {
            // Fire CANCEL event.
            this.trigger(ExampleDialog.CANCEL);
          };
          /**
           * Displays the dialog.
           */
          ExampleDialog.prototype.show = function() {
            // Do something to display the dialog. Perhaps trigger an
event as well.
          };
          return ExampleDialog;
});
```

This code sample creates an instance of `ExampleDialog` and provides listeners, using
the `on()` function, for the save and cancel events triggered in that instance.

```
/**
  * Example.js
  */
  define (["path/to/ExampleDialog"], function(ExampleDialog) {
    "use strict";
        /**
         * Example of a module that listens for ExampleDialog
events. Does nothing
         * real or useful other than establishing how to add
listeners to an
         * EventDispatcher.
         * @see ExampleDialog    */
         function Example() {
            // Note: The code in this constructor might more
typically be in an
            // initialize method or somewhere else.
            var dialog = new ExampleDialog();
```

```
                   // Add listeners for save and cancel events. 3rd arg, if
provided, sets the
                   // context for the listener
                   dialog.on(ExampleDialog.SAVE, this.onSave, this);
                   dialog.on(ExampleDialog.CANCEL, this.onCancel);
                   dialog.show();
               }
               /**
                 * Listener/handler for dialog save events.
                 * @param args {Array} Any args passed along when event is fired.
                 */
                 Example.prototype.onSave = function(args) {
                // Do something...
                console.log(args[0]);
               };
               /**
                 * Listener/handler for dialog cancel events.
                 * @param args {Array} Any args passed along when event is fired.
                 */
               Example.prototype.onCancel = function(args) {
                // Do something else...
                if (args.length === 0) {
                  console.log("No args passed with cancel event.");
                }
               };
               return Example;
         });
```

Glossary

Index