Custom Route Settings

Custom routing allows you to display SEO-friendly URLs on your site that map behind-the-scenes to conventional Kibo eCommerce resources such as a product page or a search results page. With custom routing, you gain advanced control over the URL structures on your site and can more visibly highlight the products or categories your shoppers are interested in purchasing.

For example, a category page for women's tops from a certain designer might have a URL in Kibo eCommerce that looks like: `yourSite.com/tops/c/45` . However, for SEO reasons you may prefer that the category page use a URL such as `yourSite.com/womens/tops/designerName` . With custom routing, you can use the SEO-friendly URL and let Kibo eCommerce map it to the correct category page.

Kibo eCommerce parses incoming URLs for your site and matches them to internal routes using specific rules that you set in the Custom Routing JSON Editor. To open the editor, log in to Admin and go to **System** > **Customization** > **Custom Routes**.

Define URL Routes

To define a custom route you have to:

1. Create templates to identify URL patterns for custom routing.
2. Specify the internal routes to send matching URLs to.
3. (**Optional**) Create mappings between URL parameters and Kibo eCommerce objects.
4. (**Optional**) Use validators to restrict which URL values result in a match.
5. (**Optional**) Choose which URLs are canonical.
6. (**Optional**) Choose the URL scheme for the route.
7. Verify that the order of templates in the JSON code does not cause conflicts.

Examples

The following code block shows a completed example of a custom route that you would enter into the JSON editor at **System** > **Customization** > **Custom Routes**, and the subsequent sections in this topic explain the details of the code. A productCode is always required in a custom route, but this example shows that the productCode parameter can be mapped to the productName. This allows the productName to be referenced in the URL template when the name and code are the same. For more real-world examples, see the Custom Routing Examples section.

Capitalization is not important for the JSON code described in this topic.

```
{
  "mappings": {
    "productMap": {
      "type": "direct",
      "mappings" : {
        "productName" : "ProductCode"
      }
    }
  },
  "validators": {
    "colorVal": {
      "type": "productAttribute",
      "attributeFqn" : "color"
    }
  },
  "routes" : [
    {
      "template": "home/{documentName}",
      "defaults": {
        "documentListName" : "pages@mozu"
      },
      "internalRoute": "CmsPage",
      "mappings": {},
      "validators": {},
      "canonical": true,
      "urlScheme": "https"
    },
    {
      "template": "{productName}/p/{brand}/{attribute}",
      "defaults": {},
      "internalRoute": "ProductDetails",
      "canonical": true,
      "urlScheme": "https",
      "mappings": {
        "productMap" : [
          "productName"
        ]
      },
      "validators": {
        "colorVal": [
          "attribute"
        ]
      }
    }
  ]
}
```

If at any point you want to return a route to the Kibo eCommerce default, simply delete the applicable custom routes from the JSON editor.

In addition to the routes discussed in this topic, you can also create routes to API Extension functions.

Create Templates to Identify URL Patterns for Custom Routing

Templates allow Kibo eCommerce to identify which URLs belong to a custom route by specifying the pattern of URL constants, variables, and segments that should match to a specific route. For example, you may want to apply a custom

route to a URL that looks like `yourSite.com/promotions/summer/july`. This URL has three segments after the domain name. One of the segments includes a constant (`"promotions"`) and the other segments include variables. A template that identifies this type of URL might look like the following example when you enter it into the JSON editor:

```
"routes" : [
   {
      "template": "promotions/{season}/{month}"
   }
]
```

Template Syntax

When creating templates, use the following syntax rules:

- Define variables in the URL as template parameters by enclosing them in braces `{ }`.
- You can name URL parameters whatever you wish. However, within a given template, every parameter name must be unique.
- Take advantage of the `{categorySlug}`, `{categoryCode}`, and `{categoryId}` parameters, which provide automatic validation for categories in a URL.
- Use forward slashes `/` as delimiters for URL segments.
- Any characters that are not within braces or are not forward slashes are treated as constants that must appear in the URL for a match to occur.
- To include more than one parameter within a set of delimiters, separate the parameters with a constant value. For example, `{categorySlug}-{designer}/{page}` separates the `categorySlug` and `designer` parameters with a hyphen.
- Use an asterisk to handle a variable number of URL segments. For example, `{categorySlug}/{*pages}`.
- Make sure your templates do not conflict with the default Kibo eCommerce routes.

Examples of Templates

The following table shows a list of templates and examples of URLs that match the templates.

Template	Example of matching URL
`{categoryCode}/p/{productCode}`	`bicycles/p/CAN-209`
`{a}-{b}/sale/{page}`	`mens-shoes/sale/new`
`{categorySlug}/{brand}/{color}`	`apparel/designer/black`

Template	Example of matching URL

`new/{*pages}`	`new/shoes`
	`new/shoes/sandals`
	`new/office`
	`new/home/kitchen/silverware`
	etc.

Kibo eCommerce identifies a match for a URL parameter so long as there are characters in the URL segment where the URL parameter is located. For example, in the first row of the preceding table, `"CAN-209"` matches to the `productCode` parameter, but any number of other characters would also match, such as `"bogusCode123"`. You may be wondering how you can create a match only when certain conditions apply, such as when a URL parameter matches an existing attribute value on your site. In a later section, you learn how to use validators to apply constraints to the values that can match to a particular parameter.

Default Kibo eCommerce Routes

The following table lists the default routes in Kibo eCommerce. You cannot overwrite these routes and if you create a template that conflicts with one of these routes, your template will not work, so you should make sure to avoid a naming conflict.

Relative URL	Internal Route
`user/signup`	User Signup page
`cart`	Cart page
`user/login`	User Login page
`logout`	User Logout page
`user/forgotpassword`	Forgot Password page
`c/{categoryCode}` or `{categorySlug}/c/{categoryCode}` (if slug is present)	Category page
`p/{productCode}` or `{productSlug}/p/{productCode}` (if slug is present)	Product page

Relative URL	Internal Route
`p/{productCode}?vpc={productVariationCode}` or `{productSlug}/p/{productCode}?vpc={productVariationCode}` (if slug is present)	Product Variant page
`home` or `/`	Home page
`about-us`	About Us page
`contact`	Contact page
`location`	Store Locator page
`myaccount`	My Account page
`checkout/{orderId}`	Checkout page
`checkout/{orderId}/confirmation`	Order Confirmation page

Specify the Internal Routes to Send URLs To

For each template, you need to specify the internal route that a matching URL should use. You specify this route using the `"internalRoute"` object.

In addition, some routes require specific parameters, such as a product code, to complete the route successfully. You specify these parameters using either the `"defaults"` object or by extracting them from the URL.

The following example demonstrates how you can create an internal route from `yourSite.com/pendants` to a search results page for the query, `"pendant lights"`.

```
"routes": [
   {
      "template" : "pendants",
      "internalRoute": "Search",
      "defaults": {
         "query": "pendant lights"
      }
   }
]
```

The preceding example uses the `"internalRoute"` object to identify the type of internal route and the `"defaults"` object to provide the value for the `"query"` parameter, which is a global parameter that you can apply to all routes.

As mentioned, you can also extract the parameter values from the URL. For example, let's say you want to create an internal route to a product details page. This type of internal route requires a product code. The following example obtains the value of the product code from the URL instead of from the `"defaults"` object:

```
"routes" : [
   {
      "template": "sports/{categorySlug}/{productCode}",
      "internalRoute": "ProductDetails"
   }
]
```

Because the URL parameter name in the template, i.e. `"{productCode}"`, matches the name of the required parameter for the internal route, i.e. `"ProductCode"` (see the *Available Internal Routes* table), Kibo eCommerce uses the value of the URL parameter to complete the internal route. So if a URL on your site looks like `yourSite.com/sports/soccer/ball-203`, Kibo eCommerce routes the URL to the correct product details page using the product code "ball-203".

If you want to extract a value from the URL but do not want to give the URL parameter the same name as the required parameter for the internal route, you can use a mapping.

Available Internal Routes and Corresponding Parameters

Refer to the following table for a list of the internal routes available in Kibo eCommerce and the parameters they require. You also have access to global parameters available to all routes.

Capitalization doesn't matter when specifying either the route or the parameters.

Internal Routes and Parameters

Internal Route	Required Parameters	Optional Parameters
`ProductDetails`	ProductCode	vpc
`Category`		CategoryId or CategoryCode Page
`Search`		CategoryId Page

Internal Route	Required Parameters	Optional Parameters
`CmsPage`	DocumentListName DocumentName	
`CmsList`	DocumentListName	ListView
`Cart`		

Refer to the following tables to learn about the required and optional route parameters. In addition to route-specific parameters, Kibo eCommerce provides global parameters accessible to all routes.

Route-Specific Parameters

Route-Specific Parameter	Description
`ProductCode`	Specifies the Product Code property of a product.
`vpc`	Specifies the Variation Product Code property of a product with configurable options. This code enables you to route to a product variant.
`CategoryId` / `CategoryCode`	Specifies the numeric ID of the category or the value of the Category Code property for the category. **Note:** When creating a route, use either the CategoryId parameter or the CategoryCode parameter, but not both.
`Page`	Specifies at which page to start displaying page results when you do not specify a start index. When you specify the Page parameter, Kibo eCommerce determines the start index for page results by multiplying Page by the page size for your template.
`DocumentListName`	Specifies a document list name.
`DocumentName`	Specifies a document name.
`ListView`	Specifies a list view.

Global Parameters

Global Parameter	Description
`SortBy`	Specifies how to sort page results. You sort by a number of different properties available in Kibo eCommerce, such as Category ID, Product Price, etc., and provide an "asc" or "desc" direction for ascending or descending order. For example, `"sortBy: ProductPrice asc"` This syntax matches what displays in URLs when sorting is applied to your site: `yourSite.com/products?sortBy=productprice asc`.
`Query`	Specifies the query string for searches.
`StartIndex`	Specifies at which item to start displaying page results. The default is 0, which corresponds to the first item in the results.
`PageSize`	Specifies how many results to display per page.
`FacetValueFilter`	Specifies the facet to filter results with.

Create Mappings Between URL Parameters and Kibo eCommerce Objects

Mappings allow you to map the values of URL parameters to other JSON variables or to Kibo eCommerce objects such as facet values. Mappings work by adding or replacing entries in the Kibo eCommerce route dictionary. The following types of mappings are available for use:

- Direct
- Facet
- Regex
- Use the beforeRouting Boolean for Category Mappings

To configure mappings, you create a `"mappings"` object at the same level as the `"routes"` object. This `"mappings"` object contains the details of all the mappings available to your site. You also create a `"mappings"` object at the same level as the `"templates"` object within the `"routes"` object. This `"mappings"` object specifies the name of the mapping to use for a particular route.

If you want to apply a mapping to a template that includes a `categoryCode`, `categoryId`, or `categorySlug` parameter, you must add the `"beforeRouting"` Boolean to the mapping.

Direct Mappings

Direct mappings reassign the value of a URL parameter to another value.

The following example demonstrates a route that reassigns the value of the `"{product}"` URL parameter. For the URL `yourSite.com/flowers/scarletRoses`, Kibo eCommerce executes a route to a product details page in the `"flowers"` category using `"redRoses"` as the new product code by applying the `"productMap"` mapping to the parameter `"product"`. Note that you must enable the `"beforeRouting"` property so that the mapping takes place before the route occurs.

```
{
   "mappings": {
      "productMap": {
         "type": "direct",
         "beforeRouting": true,
         "mappings" : {
            "scarletRoses" : "redRoses"
         }
      }
   },
   "routes" : [
      {
         "template": "{categorySlug}/{product}",
         "defaults": {},
         "internalRoute": "ProductDetails",
         "mappings": {
            "productMap": [
               "product"
            ]
         }
      }
   ]
}
```

Facet Mappings

Facet mappings allow you to assign the value of a URL parameter to one of the following search parameters:

- `Query`
- `CategoryId`
- `SortBy`
- `StartIndex`
- `Page`
- `PageSize`
- `FacetValueFilter`

Facet mappings are designed to work with Category and Search routes; they do not have practical value if used in conjunction with the other internal routes.

The following example demonstrates a route that assigns the value of the `"{color}"` URL parameter to the `"facetValueFilter"` search parameter. For the URL `yourSite.com/accessories/ties/red`, Kibo eCommerce executes a route to a `"accessories/ties"` category page using `"red"` as the faceting value by applying the `"facetMap"` mapping to the parameter `"color"`. In this example, the benefit of the mapping is that you can use an SEO-friendly URL for a faceted search results page that would otherwise require a character-heavy query string in the

URL to display a list of red ties. Note that you must enable the `"beforeRouting"` property so that the mapping takes place before the route occurs.

```
{
  "mappings": {
    "facetMap": {
      "type": "facetValueFilter",
      "mapTo": "facetValueFilter",
      "beforeRouting": true,
      "facetId": "color"
    }
  },
  "routes" : [
    {
      "template": "{parent-categorySlug}/{categorySlug}/{color}",
      "defaults": {},
      "internalRoute": "Category",
      "mappings": {
        "facetMap": [
          "color"
        ]
      }
    }
  ]
}
```

You can also map URL parameters to facets using the `"facetId-facet"` token in the template. This syntax recognizes that the URL parameter should be mapped to a facet and doesn't require you to code the mapping details. Using this syntax, the previous example looks like:

```
{
  "routes" : [
    {
      "template": "{parent-categorySlug}/{categorySlug}/{tenant~color-facet}",
      "defaults": {},
      "internalRoute": "Category",
      "mappings": {}
    }
  ]
}
```

Regex Mappings

With a regex mapping, you can search for a string pattern in a URL parameter and replace it with another string of your choosing. Optionally, you can leave the URL parameter intact, and instead apply the new pattern to a custom key-value pair in the routeData global variable, which can later be accessed in a Hypr template or through an API Extension application.

The following example demonstrates a regex mapping that applies the `"toSpace"` mapping to the `"{categorySlug}"` parameter. The `"toSpace"` mapping searches for non-whitespace characters, such as underscores, and replaces them with spaces. The optional field, `"mapTo"`, applies the result of the replacement to the a custom key-value pair in the `routeData` global variable. If you wanted to apply the replacement to the URL parameter that the mapping targets, you would omit the `"mapTo"` field.

```
{
  "mappings": {
    "toSpace": {
      "type": "regex",
      "mapTo": "categoryName",
      "pattern": "\\S",
      "replacement": " "
    }
  },
  "validators": {},
  "routes": [
    {
      "template": "clearance/{categorySlug}/index",
      "defaults": {
        "documentListName": "pages@mozu"
      },
      "internalRoute": "CmsPage",
      "toSpace": {
        "": [
          "categorySlug"
        ]
      },
      "canonical": true,
      "validators": {}
    }
  ]
}
```

Use the "beforeRouting" Boolean to Apply Mappings to Category Parameters

When you apply a mapping to a template that includes a category parameter, such as `categoryCode`, `categoryID`, or `categorySlug`, you must include the `"beforeRouting"` Boolean (set to `true`) in the route. Otherwise, the route uses the value of the category parameter before the mapping has had a chance to take place. The following example demonstrates a mapping that reassigns `"coat"` category parameter values to `"jackets"`. For such a case, the `"beforeRouting"` Boolean is necessitated.

```
{
  "mappings": {
    "categoryMapping": {
      "type": "direct",
      "beforeRouting": true,
      "mappings": {
        "coats": "jackets"
      }
    }
  },
  "routes": [
    {
      "template": "winter/{categoryCode}",
      "internalRoute": "Category",
      "mappings": {
        "categoryMapping": [
          "categoryCode"
        ]
      }
    }
  ]
}
```

Use Validators to Restrict Which URL Values Result in a Match

Validators require that URL parameters meet certain conditions before Kibo eCommerce considers the template a match. Take the following template as an example: `{categorySlug}/{attribute}`. Without a validator, a URL can have any number of values in the location of the `{attribute}` parameter and still match the template. With a validator in place, Kibo eCommerce checks whether the parameter value meets a set of specified criteria, for example, whether it corresponds with an existing attribute defined on your site, and only matches the template when this check is true.

You can use the following types of validators:

- Attribute
- List
- Category
- Facet

Attribute Validator

Attribute validators check whether a URL parameter corresponds to an existing attribute on your site. These validators require you to provide the administration name of an attribute in order for Kibo eCommerce to look up the values of the attribute and determine if a match exists.

The following example demonstrates how to implement an attribute validator. Let's assume that you have defined an attribute on your site with the administration name of `"color"` and that the attribute contains the values `"red"`, `"green"`, and `"blue"`. With the validator in place, the URL `yourSite.com/products/green` matches the template: Kibo eCommerce applies the `"colorVal"` validator to the `"{colorValue}"` parameter, checks whether `"green"` matches any of the values of the attribute with the administration name of `"color"`, and finds a match. On

the other hand, the URL `yourSite.com/products/purple` does not match the template because `"purple"` is not a defined value for the `"color"` attribute.

```
{
   "validators": {
      "colorVal": {
         "type": "productAttribute",
         "attributeFqn" : "color"
      }
   },
   "routes" : [
      {
         "template": "products/{colorValue}",
         "internalRoute": "ProductDetails",
         "validators": {
            "colorVal": [
               "colorValue"
            ]
         }
      }
   ]
}
```

List Validator

List validators check whether a URL parameter matches any value within a list that you define in the JSON code.

The following example demonstrates how to implement a list validator. With the validator in place, the URL `yourSite.com/products/vacuum` matches the template: Kibo eCommerce applies the `"productVal"` validator to the `"{productValue}"` parameter, checks whether `"vacuum"` matches any of the values defined in the list, and finds a match.

```
{
   "validators": {
      "productVal": {
         "type": "stringlist",
         "values" : [
            "broom", "mop", "vacuum"
         ]
      }
   },
   "routes" : [
      {
         "template": "products/{productValue}",
         "internalRoute": "ProductDetails",
         "validators": {
            "productVal": [
               "productValue"
            ]
         }
      }
   ]
}
```

Category Validator

Kibo eCommerce contains logic to automatically validate your category tree when you create routes to category and product pages (`"internalRoute": "Category"` and `"internalRoute": "ProductDetails"` , respectively). This saves you the trouble of having to explicitly add validators to check whether the value of a URL parameter corresponds to an existing category on your site.

To take advantage of this logic, name category parameters in your template according to the following syntax rules:

- To identify a URL parameter as a category that Kibo eCommerce should validate, use one of the names described in the following table. During validation, Kibo eCommercechecks whether the parameter value matches the corresponding property value of a category on your site.

URL Parameter Name	Matching Category Property
`{categorySlug}`	SEO Friendly URL
`{categoryId}`	Auto-generated numeric ID
`{categoryCode}`	Category Code

- To identify a URL parameter as a parent category, use the prefix `parent-` . This allows you to validate and display a category structure in a URL, such as `yourSite.com/office-supplies/pens` .
- To identify a second level of parent category, use the prefix `grandParent-` .
- To identify additional levels of parent categories, use the prefix `great-` before `grandParent-` . You can string together additional `great-` prefixes to create additional category levels.
- As an alternative to the `parent/grandParent` syntax, you can place a colon after the category parameter and use the `ancestors(n)` token to specify parent levels, where `n` is the number of parent categories the template requires.
- To identify two separate category trees within the same URL, specify a constant prefix for one of the category trees.

The following table lists examples of templates that employ category parameters. The table also shows examples of URLs that contain category codes, SEO-friendly names, and IDs that match the pattern of the templates. Behind the scenes, Kibo eCommerce checks whether the values match to existing category values on your site.

Template	Example of matching URL
`{categorySlug}/{product}`	`sofas/recliner-01`
`{parent-categoryCode}/{categoryCode}/{product}`	`Seating3/Sofas/recliner-01`
`{grandParent-categoryId}/{parent-categoryId}/{categoryId}/{product}`	`19/14/21/recliner-01`

Template	Example of matching URL
`{great-grandParent-categorySlug}/{grandParent-categorySlug}/{parent-categorySlug}/{categorySlug}/{product}`	`furniture/seating/living-room/sofas/recliner-01`
`{great-great-grandParent-categorySlug}/{great-grandParent-categorySlug}/{grandParent-categorySlug}/{parent-categorySlug}/{categorySlug}/{product}`	`home/furniture/seating/living-room/sofas/recliner-01`
`{categorySlug:ancestors(5)}/{product}`	`home/furniture/seating/living-room/sofas/recliner-01`
`specials/{specials-categorySlug}?cat={parent-categorySlug}/{categorySlug}`	`specials/dresses?cat=womens/designer-dresses`

Facet Validator

Kibo eCommerce contains logic to automatically validate whether a URL parameter corresponds to a valid facet value. To take advantage of this logic, use the following syntax for facet parameters in your template:

`{`*namespace*`~`*attributeName*`-facetValue}`

If you construct a parameter using this syntax, Kibo eCommerce automatically validates whether the facet value exists for a given attribute (which is identified by its namespace and administration name). The following table shows an example of a template that employs a template parameter and a URL that matches the template. Before identifying a match, Kibo eCommerce validates whether `"south-america"` is a valid facet value for the `"tenant~region"` attribute.

Template	Example of matching URL
`{categorySlug:ancestors(2)}/{tenant~region-facetValue}`	`food/fruit/south-america`

Choose Which URLs Are Canonical

Custom routing allows you to map multiple URLs to the same content. For example, the URLs `yourSite.com/shop/green-tea` and `yourSite.com/healthy-drinks` can both link to the same category page for green teas. While there are many benefits to having multiple paths to the same content, search engines can penalize websites for displaying the same content on more than one web page.

To counteract this negative effect on SEO, Kibo eCommerce uses the `canonical` flag to mark a URL structure as the preferred one for search engine crawlers to index and for shoppers to see in the URL bar after the route executes. By default, the standard Kibo eCommerce URL structure (for example, `yourSite.com/{categorySlug}/c/{categoryCode}` for category pages) is marked as canonical. However, if

you want the URL structure defined by one of your templates to be canonical instead (thereby displaying it to shoppers after the route executes and marking it to be indexed by search engine crawlers), apply the `canonical` flag.

```
{
  "routes" : [
    {
      "template": "shop/{categoryCode}",
      "internalRoute": "Category",
      "canonical" : true
    }
  ]
}
```

Non-canonical routes execute a 301 redirect to the canonical route, and any values specified in the `"defaults"` object of the non-canonical route are overwritten by the values in the `"defaults"` object of the canonical route. In the preceding example, if a shopper were to type in the standard URL to a Kibo eCommerce category page (`yourSite.com/{categorySlug}/c/{categoryCode}`), Kibo eCommerce would redirect them to `yourSite.com/shop/{categoryCode}` .

Choose the URL Scheme for the Route

You can set the URL scheme for the route to either `http` or `https` . This allows you to set encryption on a route generated from a non-secure request. If you do not specify this property, the default is `http` .

```
{
  "routes" : [
    {
      "template": "shop/{categoryCode}",
      "internalRoute": "Category",
      "urlScheme" : "https"
    }
  ]
}
```

The scheme you set in the JSON overrides the scheme of the incoming request. For example, if an incoming HTTP request matches a template whose `"urlScheme"` property is set to `https` , then the route occurs as normal but changes the scheme to HTTPS.

Verify that the Order of Templates Does Not Cause Conflicts

When Kibo eCommerce searches for a routing template to match an incoming URL request, it checks the templates in the order that the they are defined in the Custom Routing JSON Editor. Once Kibo eCommerce finds the first match, it does not evaluate the remaining templates in the editor to determine if there is a better match. This means that Kibo eCommerce may never evaluate a template if a more general template precedes it in the JSON code.

The following example shows the correct way to order your templates when there is a possibility of a match conflict. In the example, the `"offers/{price}"` template includes a validator that checks whether the `"{price}"` parameter matches to a value in the specified list. The `"offers/{new}"` template is more general and does not include a validator. If we have two incoming URLs, `yourSite.com/offers/under35` and `yourSite.com/offers/desk-`

`ornaments`, the first URL matches to the `"offers/{price}"` template and the second URL matches to the `"offers/{new}"` template.

However, what would happen if we reversed the order of the templates in the JSON editor? In this case, a match conflict renders one of the templates useless because Kibo eCommerce would match both URLs to the `"offers/{new}"` template and never evaluate the more restrictive `"offers/{price}"` template.

```
{
  "validators": {
    "priceVal": {
      "type": "stringlist",
      "values" : [
        "under35", "35-50", "over50"
      ]
    }
  },
  "routes" : [
    {
      "template": "offers/{price}"
      "internalRoute": "Category",
      "validators": {
        "priceVal": [
          "price"
        ]
      }
    },
    {
      "template": "offers/{new}",
      "internalRoute": "Category",
    }
  ]
}
```

Create a Route to an API Extension Function

You can use the `http.storefront.routes` action to bind an API Extension function to a route on your site. This allows you to run the API Extension function when the route is requested versus having to respond to a specific event on a page—providing you more flexibility to run functions independently from the events used in other functions. For example, you can create an arbitrary URL for a third-party service to POST to and have the function process the data when it is received.

Before completing the steps in this topic, you should be familiar with the API Extensions framework and also with setting up custom routes:

- Configure Custom Routing
- API Extensions Help

Complete the following steps to create the route:

1. Scaffold an API Extension application that contains the `http.storefront.routes` action.
2. Add code to the custom function tied to the action. For example, in the `http.storefront.routes.js` file:

```
module.exports = function(context) {
   context.response.body = "Hello Custom Routing!";
   context.response.end();
};
```

3. Note the name of the function in the `storefront.manifest.js` file, located in the `assets/src` directory. The default name matches the action name, but you can change the name if you wish. In the following example, the default function name is changed to `hello_custom_routing`.

```
'hello_custom_routing': {
   actionName:'http.storefront.routes',
   customFunction: require('./domains/storefront/http.storefront.routes')
}
```

4. Build and upload the application to Dev Center.
5. Install the application to a sandbox.
6. From Admin, ensure that the `http.storefront.routes` action is enabled in the Action Management JSON Editor.
7. In the Custom Routing JSON Editor, create a route to the function using:
   - `"internalRoute": "Arcjs",`
   - `"functionId": "`*yourFunctionName*`",`
8. You can add mappings, validators, and any other custom route features that you wish, as discussed in the main custom routing section.

```
{
   "mappings": {},
   "validators": {},
   "routes": [
      {
         "template": "shop/deals",
         "defaults": {},
         "internalRoute": "Arcjs",
         "functionId": "hello_custom_routing",
         "mappings": {},
         "validators": {}
      }
   ]
}
```

Custom Routing Examples

View JSON sample code for the following custom routing scenarios:

- Route to a Document
- Route to a Product
- Route to a Document Through a Regex Mapping
- Modify the Category Slug for Routes to Specific Designer Pages
- Include Key-Value Pairs Within a Document Route

- Route to an API Extension Function

Route to a Document

`yourSite.com/sales/may` routes to the document named `may` within the `sales@yourSite` document list.

```
{
  "mappings": {},
  "validators": {},
  "routes": [
    {
      "template": "sales/{documentname}",
      "defaults": {
        "documentListName": "sales@yourSite"
      },
      "internalRoute": "CmsPage",
      "mappings": {},
      "validators": {},
    }
  ]
}
```

Route to a Product

The default route to a product details page is `yourSite.com/{productSlug}/p/{productCode}`. If you want to provide an alternative route to a product details page, use the following custom route as an example:

If you want your alternative route to be indexed by search engines and to be the URL that shoppers see on the product details page, set the `canonical` flag to `true`.

```
{
  "mappings": {},
  "validators": {},
  "routes": [
    {
      "template": "shop/{productSlug}/{productCode}/info",
      "defaults": {},
      "internalRoute": "ProductDetails",
      "mappings": {},
      "canonical": false,
      "validators": {},
    }
  ]
}
```

Route to a Document Through a Regex Mapping

You can route to a document using a regex mapping, as shown in the following example, which removes whitespace from category slugs that match a specific format and then renders a resulting route to the correct document. A case where this may be useful is when mapping a productName parameter to a productCode, as shown in the first example of this guide. Using a regex allows you to strip or replace the spaces from your productName to match the productCode.

```
{
  "mappings": {
    "category-regex": {
      "type": "regex",
      "mapTo": "documentname",
      "pattern": "\\s",
      "replacement": ""
    }
  },
  "validators": {},
  "routes": [
    {
      "template": "variations/{variations-categorySlug}/index",
      "defaults": {
        "documentListName": "pages@mozu"
      },
      "internalRoute": "CmsPage",
      "mappings": {
        "category-regex": [
          "variations-categorySlug"
        ]
      },
      "canonical": true,
      "validators": {}
    }
  ]
}
```

Modify the Category Slug for Routes to Specific Designer Pages

You can replace or modify how specific designer category URLs display, as shown in the following example, where for specific designer pages any instance of `mens` in the URL category slugs is replaced with `men` instead:

```
{
  "mappings": {
    "singular": {
      "type": "regex",
      "beforeRouting": true,
      "pattern": "mens",
      "replacement": "men"
    }
  },
  "validators": {
    "designers": {
      "type": "stringlist",
      "values": [
        "designerA",
        "designerB",
        "designerC"
      ]
    }
  },
  "routes": [
    {
      "template": "{designer}/{categorySlug:ancestors(5)}",
      "defaults": {
        "categorySlug": "shop",
        "isDesignerPage": "true"
      },
      "internalRoute": "Category",
      "mappings": {
        "singular": [
          "categorySlug",
          "parent-categorySlug",
          "grandParent-categorySlug",
          "great-grandParent-categorySlug",
          "great-great-grandParent-categorySlug",
          "great-great-great-grandParent-categorySlug"
        ]
      },
      "canonical": true,
      "validators": {
        "designer": [
          "designers"
        ]
      }
    }
  ]
}
```

Include Custom Key-Value Pairs Within a Document Route

You can route to a document while including key-value pairs in the `"defaults"` object, which are then accessible in the resulting page's Hypr template through use of the routeData global variable. In the following example, the key-value pairs that you make accessible to the Hypr template are `"list": "pages@mozu"` and `"isOffersPage": "true"`. You can then use these specific values to modify the content that the template displays in response to the route that executes.

```
{
  "mappings": {},
  "validators": {},
  "routes": [
    {
      "template": "pages/{name}/{documentListFQN}/{documentProperty-page_type_definition
}",
      "defaults": {
        "list": "pages@mozu",
        "isOffersPage": "true"
      },
      "internalRoute": "CmsPage",
      "mappings": {},
      "canonical": false,
      "validators": {}
    }
  ]
}
```

Route to an API Extension Function

You can create a route that serves as an endpoint for a custom API Extension function to execute on. For example, you can create a route that runs a custom function for your PayPal integrations whenever the `paypal/checkout` endpoint is hit:

```
{
  "mappings": {},
  "validators": {},
  "routes": [
    {
      "template": "paypal/checkout",
      "defaults": {},
      "internalRoute": "Arcjs",
      "functionId": "myPayPalFunction",
      "mappings": {},
      "validators": {}
    }
  ]
}
```