Implementing Commerce I/O EDP

Overview

This document shows how data is handled in Commerce I/O .

Data Modeling

To get the most out of the SAP Hybris - Jahia integration, you must index the product catalog in Elasticsearch (ES). Indexing enables you to implement a performant External Data Provider (EDP) in Jahia so that users can enjoy standard edit mode features and use the ES index directly for speedy search queries at the page level. To achieve this the original catalog data structure was remodeled. After you index for the first time, you will notice that two indices were created. One index is the catalog and it contains products. The second index is the category tree that is available on the site. Note that our modeling is the bare minimum to provide all needed functionality but you could always improve/enhance your model to be able to add additional features.

Product model

In the product model, consider the following key points when you customize search functionality:

- In some cases, the model applies several different analysers to the same entry to provide additional flexibility.
- Each product entry contains basic information such as a name and description and is uniquely identified by its SKU.
- Each entry contains a *mountedPath* property which specifies the path for the product in the JCR. Because the JCR is a tree structure, placing many children under the same node significantly degrades performance as the number of children increases. Therefore, a splitting rule generates the paths, balances the tree and helps to optimize performance. You can find details about the splitting rule in the *org.jahia.modules.commerce.io.edp.datasource.model.Product* class.
- Each entry also contains a *baseProduct* property and *variants* property, if they exist for that product.

Note: You can find custom settings and mappings for the index in *hybris-store* module in *src/main/resources/META-INF* folder.

The following example shows the product object as stored in Elasticsearch.

```
{
```

```
"name": "The Rubber Re-Run",
"sku": "74818",
"code": "74818",
"summary": "...",
"baseProduct": "BK340962",
"price": { "currencyIso": "GBP","formattedValue": "£80.96","value": 80.96,"priceType": "FROM"},
"images": [...]
"mountedPath": "/p/hi/ib/dj/74818",
"vanityUrl": "/product/en/74818.html",
"extendedCategories": [...],
"variants": { "features": [ "style"],"style": [ "sky blue"]},
"selectedFeatures": null
}
```


Images are stored in the images array and provide basic information about image type and path to the images on your Hybris system.

The *extendedCategories* property is an array of available categories for given product. The property is extended, meaning that it includes the category name, category ID, path to that category, and path by ID to that category.

Categories model

Categories are modeled to keep a flat view of the tree. As with products, there are settings and mapping that you can view for additional info on the index.

Each category entry has a name, path, idPath, children and childIds property. The idPath is a path to that category by category ids. The childIds property is an array containing child category ids. The following example shows a category entry in Elasticsearch.


```
{
"name": "Eyewear Women",
"path": "/Categories/Eyewear-Women",
"idPath": "/categories/240000",
"children": [
"Shades-Women"
],
"childIds": [
"240100"
],
"id": "240000"
}
```

Vanity URLs in EDP

Commerce I/O implements Jahia's built in Vanity URL system to create human-readable URLs for a better user experience.
Vanity URLs are used to access and retrieve product and catalog pages. Using Vanity URLs, Commerce I/O can work with and parse the paths in an easier manner when performing node discovery in the External Data Provider (EDP).

In the EDP, Commerce I/O implements the *search* method to search the Elasticsearch index for the specified product and to perform product and catalog Vanity URL node discovery.

You can change and modify prefixes in the *StoreDataSource* class found in *org.jahia.modules.commerce.io.edp.datasource* package.

Vanity URLs are used only in Live mode. In Preview and Edit mode, URLs use the node path mounted by the EDP.

URL Syntax

The Vanity URL structure contains a prefix that identifies the type of resource to retrieve, for example, products or catalogs. The prefix is followed by the locale which enables you to provide I18N resources. The final part of the path is a unique string that identifies the actual resource. The identifier for a product is the sku/code and the identifier for a category is the name of the category. You can access child categories by providing a hierarchical path, for example, /brands/burton.html.

The following shows the syntax for product and catalog vanity URLs.

- Product - */products/{locale}/{sku}.html*
- Catalog - /catalogs*/{locale}/{category}.html*
    - Child categories can be accessed by: /catalogs*/{locale}/{parentCategory}/{childCategory}.html*

Implementing the External Data Provider

This section describes the main components of the *StoreDataSource* class.

Path navigation

The *StoreDataSource* class in the *org.jahia.modules.commerce.io.edp.datasource* package is responsible for providing external data. As with all external providers in Jahia, the *getItemByPath* method resolves external data for an item, given its path.

Let's look at the method in detail to understand it a little better. For the "/" ciont:categoryRoot node is returned after which there two major top level paths: ones that start with "/p" (paths for products) and ones that start without "/p" (paths for categories).

When it comes to product path there are several cases to consider.
If the path:

- Ends with product sku then external data for that product is returned.
- Ends with "/variants" then the ciont:variantsList node is returned.
- Contains "/variants/" then the ciont:variantFeatures node is returned or ciont:variantOption depending on whether the path contains "features".
- Contains "/medias" then the ciont:medias node is returned.
- Ends with "/vanityUrlMapping" then the jnt:vanityUrls node is returned.
- Contains "/vanityUrlMapping" then the jnt:vanityUrl node is returned.

In all other cases "/p" jnt:contentFolder is returned.

The category part is a lot simpler. If the path:

- Ends with "/vanityUrlMapping" then jnt:vanityUrls node is returned.
- Path contains "/vanityUrlMapping" then jnt:vanityUrl node is returned.

Otherwise the category ciont:category node is returned.

In all other cases PathNotFoundException is returned.

Children

The getChildren method is also a standard method in External Provider, which returns a list of child paths for a given path. As with the *getItemByPath* method, there are two major cases for paths: those that contain "/p" and those that do not.

The category path case is very simple.

- If the path is a category path and it ends with "/vanityUrlMapping", a list of Vanity URLs spanning every available language is returned.
- Otherwise, a list of child paths for that category is returned.

The product path case is a bit more complex as we need to take into account medias and variants.

- If it is a pure product path (ending with sku), then child paths for media and variants are returned.
- If the path ends with "/medias" then a list for every available media child is returned.
- If the path ends with "/vanityUrlMapping" similarly to category path a list of vanity urls spanning every available language is returned.
- If it ends with "/variants" a list of available child variant paths is returned.

In the case when none of the conditions match an empty list is returned.

Search

The *search* method in the class applies when the product is searched for in Edit mode in DX or when user is trying to access page with a vanity url. The method returns a list of available paths which are then fed into *itemByPath* method.

The first check is done to see if the search is for product(s) and if it is the mounted path(s) for that product(s) is returned. The mounted path is the path you see in JCR which has the following structure: "/p/td/re/vf/<SKU>".

Another check catches two types of vanity urls: one for product and one for category. In each case mounted path for that product or category is returned.

Caching

Why Caching?

Caching is used throughout the system to guarantee quick access to information and reduce the load on the system. All catalog entries are cached and can be accessed without making calls to Elasticsearch after initial calls are made.

There two types of data that gets cached in Commerce I/O: catalog (product and category) and user data.

Catalog Cache Configuration

You can find the classes used for product and category caching in the *org.jahia.modules.commerce.io.edp.datasource.cache* package. The main class containing cache configuration is *StoreCacheManager* (declared as a bean). The *createStoreCache* method creates the cache and sets the expiration time to 3600 seconds. From that point on every request for a product or category in the *StoreDataSource* class is cached. The cache is invalidated in the *indexCatalogAndProduct* method of the *CatalogIndexerImpl* class on every successful index operation.

To be able to work with i18n data and the external provider information for every international property is included in cache as *valuesForLazyProperties* map in the model class. These values are resolved at run time in the *geti18nPropertyValues* method of *StoreDataSource*. If the product or category is already cached in some language and you try to access it in a language that does not have map entry, a query to Elasticsearch will be made again.

User Cache Configuration

The class used for user caching can be located in the *org.jahia.commerceplatform.users.cache* of the **Commerce IO SAP Hybris Customer Provider** module. The cache is created inside *createHybrisCache* method with an expiration time of 90,000 seconds. That expiration time is the default expiration time for Hybris token. The cache is used in the *HybrisUsersGroupsProvider* class.

Checking Cache

The cache can be explored in tools. Navigate to *<your host>/modules/tools/index.jsp* and then click on **Cache management**. You will see four caches that belong to Commerce I/O: **HybrisGroupsCache**, **HybrisUsersCache**, **StoreCatalogCache** and **StoreProductCache**.

HybrisUserCache uses userId as key while HybrisGroupsCache is not yet used but can be utilized to enhance caching features.

StoreProductCache keys contain information about environment (elasticsearch connection name and index prefix separated by underscore to be exact) and product code and have the following structure: *<connection name>_<prefix>p<product code>*.

StoreCategoryCache uses similar key structure as above with minor exceptions. Product code is replaced by category path, “p” is replaced by “c” and “_categories_” is added:

<connection name>_<prefix>_categories_c<category path> .