Platform, Services, and Utilities

Generated on: 2024-11-08 15:03:39 GMT+0000

SAP Commerce | 2205

PUBLIC

Original content: https://help.sap.com/docs/SAP_COMMERCE/d0224eca81e249cb821f2cdf45a82ace?locale=en-US&state=PRODUCTION&version=2205

Warning

Digital Asset Management

Digital Asset Management allows you to organize media and make them broadly accessible.

SAP Commerce supports media files of all sorts. A media can be anything that can be saved on a file system, such as a Flash animation file, a JPEG image, an MPEG video file, a CSV file, a text file, an XML file, and other. These media can be stored in various locations. You can keep them locally or remotely using Amazon S3, Windows Azure Blob or MongoDB GridFS solutions. Access to your media can be also secured.

The topics include:

Media Items

A media item in SAP Commerce isn't a physical file, but a reference to that file. A media item in SAP Commerce is a container object that holds a reference to a file.

Media Folders

SAP Commerce allows you to create folders for organizing and managing your media. You can use the web server or application server to limit access to media files by media folder.

Storing Many Files for One Media Item

Normally, a single **Media** item holds reference to only one file. However, you may need to have a **Media** item that refers to more than one file. For example, you have many pictures of one product and you want to have only one **Media** item for all these pictures.

Media URLs

Media URLs have different structures depending if the access to a media item is secured or not.

Media Storage

The concept of media storage means how and where binary data of any Media item is stored and read.

Secure Media Access

Secure media access means giving access to media after checking permissions. In other words, you cannot access specific media by just guessing its URL. Permission check while accessing medias is ensured by a special filter, which you can add to your custom Web application.

WebAppMediaFilter

The `WebAppMediaFilter` filter unifies the way media is served. It handles both secure and non-secure types of media as opposed to `MediaFilter` and `SecureMediaFilter`.

Using ETag for Web Cache Validation

SAP Commerce uses HTTP ETags in web cache validation.

Customer Response Headers for Media Filters

Set custom response headers to improve security of uploaded media.

Media Items

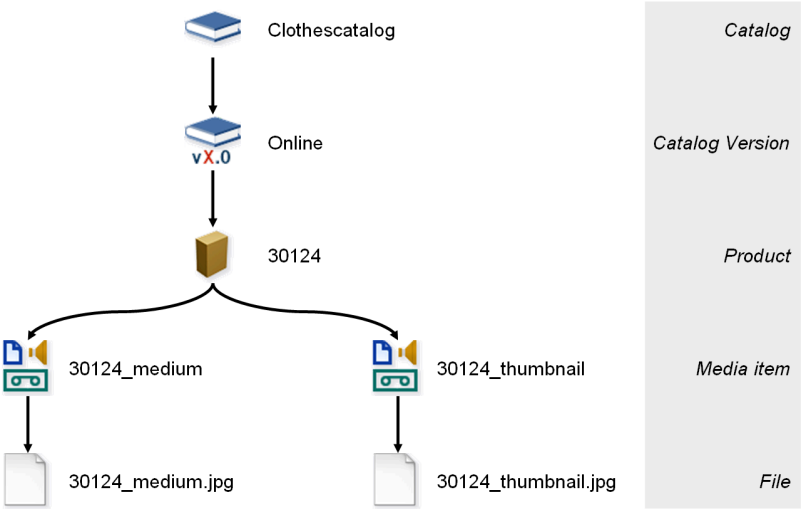
A media item in SAP Commerce isn't a physical file, but a reference to that file. A media item in SAP Commerce is a container object that holds a reference to a file.

Actual media files can be stored in SAP Commerce or may be located on a remote server or system.

Media Item Attributes

A media item has an identifier and is assigned to a catalog version. The identifier is used for the logical reference, such as, for matching a product identifier. Media items can be synchronized along with the catalog version to which they're assigned. That way, you can make sure that product images match the catalog version.

In addition, a media item has a URL that points to the location of the actual file. To reference a file in an SAP Commerce application, you need to retrieve and use the media item's URL.

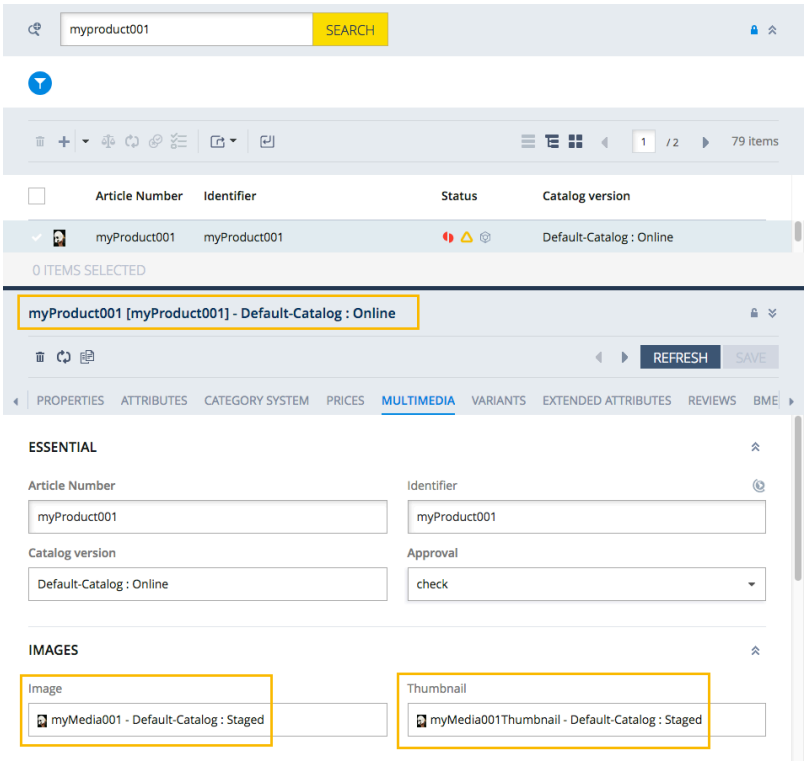


In the graphic, the catalog **Clothescatalog** contains one catalog version (**Online**), which holds one product (**30124**). This product references two media items; **30124_medium** and **30124_thumbnail**. The media item, **30124_medium** references the file, **`30124_medium.jpg`**, whereas the media item, **30124_thumbnail** references the file, **`30124_thumbnail.jpg`**.

One File Per Media, Several Media Per File

A single media item in SAP Commerce references one single file only.

In the screenshot, you can see that both the actual `myProduct001` product image and its thumbnail are individual media items and, by consequence, individual files.



Even though both images are related in that they're assigned to the same product, they aren't represented by the same media item. This is indicated by both the individual filenames and individual URLs for the images. To check it, click the names of the images to display their data.

You can also have more than one file matched to one media item. For more details, see Storing Many Files for One Media Item.

Using Localized Media

Media items in SAP Commerce aren't localized by default. A media item doesn't reference individual files depending on the language. Instead, the referenced file is the same for all languages. To use localized files, you need to use an attribute of type **localized:Media** or a **RelationType** in the **`items.xml`** file.

Using a Media Item

Since a media item in SAP Commerce is a container rather than an actual file, you need to reference the actual location of the file represented by the media item. The location is held in the media item's **`url`** attribute.

```
<%
   Media media = ...
   out.println("<img src=\"" + media.getURL() + "\"/>");
%>
```

Supported MIME Types

See a list of MIME types supported in media files:

File Extension	MIME Type
xls	application/excel
xlsx	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
doc	application/msword
docx	application/vnd.openxmlformats-officedocument.wordprocessingml.document
dot	application/msword
dotx	application/vnd.openxmlformats-officedocument.wordprocessingml.template
wrd	application/msword
bin	application/octet-stream
dms	application/octet-stream
lha	application/octet-stream
lzh	application/octet-stream
exe	application/octet-stream
class	application/octet-stream
iso	application/octet-stream
pdf	application/pdf
pgp	application/pgp
ai	application/postscript

File Extension	MIME Type
eps	application/postscript
ps	application/postscript
ppsx	application/vnd.openxmlformats-officedocument.presentationml.slideshow
ppt	application/powerpoint
pptx	application/vnd.openxmlformats-officedocument.presentationml.presentation
rtf	application/rtf
wp5	application/wordperfect5.1
bcpio	application/x-bcpio
bz2	application/x-bzip2
pgn	application/x-chess-pgn
z	application/x-compress
cpio	application/x-cpio
dvi	application/x-dvi
gtar	application/x-gtar
tgz	application/x-gta
gz	application/x-gzip
phtml	application/x-httpd-php
pht	application/x-httpd-php
php	application/x-httpd-php
js	application/x-javascript
kwd	application/x-kword
kwt	application/x-kword
ksp	application/x-kspread
kpr	application/x-kpresenter
kpt	application/x-kpresenter
chrt	application/x-kchart
latex	application/x-latex
com	application/x-msdos-program
bat	application/x-msdos-program
pl	application/x-perl
rpm	application/x-rpm

File Extension	MIME Type
shar	application/x-shar
swf	application/x-shockwave-flash
sit	application/x-stuffit
tar	application/x-tar
tcl	application/x-tcl
tex	application/x-tex
texinfo	application/x-texinfo
texi	application/x-texinfo
zip	application/zip
au	audio/basic
snd	audio/basic
mid	audio/midi
midi	audio/midi
kar	audio/midi
mpga	audio/mpeg
mp3	audio/mpeg
m3u	audio/x-mpegurl
aif	audio/x-aiff
aifc	audio/x-aiff
aiff	audio/x-aiff
ra	audio/x-realaudio
wav	audio/x-wav
bmp	image/bmp
gif	image/gif
ief	image/ief
jpeg	image/jpeg
jpg	image/jpeg
jpe	image/jpeg
png	image/png
tiff	image/tiff
tif	image/tiff

File Extension	MIME Type
wrl	model/vrml
vrml	model/vrml
css	text/css
html	text/html
htm	text/html
asc	text/plain
txt	text/plain
asm	text/plain
c	text/plain
java	text/plain
csv	text/csv
impex	text/x-comma-separated-values
cc	text/plain
h	text/plain
cpp	text/plain
rtx	text/richtext
rtf	text/rtf
sgml	text/sgml
sgm	text/sgml
vcs	text/x-vCalendar
vcf	text/x-vCard
xml	text/xml
dtd	text/xml
xsl	text/xml
mp2	video/mpeg
mp3	video/mpeg
mpeg	video/mpeg
mpg	video/mpeg
gt	video/quicktime
mov	video/quicktime
vm	text/plain

File Extension	MIME Type
wmv	video/x-ms-wmv
groovy	text/plain
svg	image/svg+xml

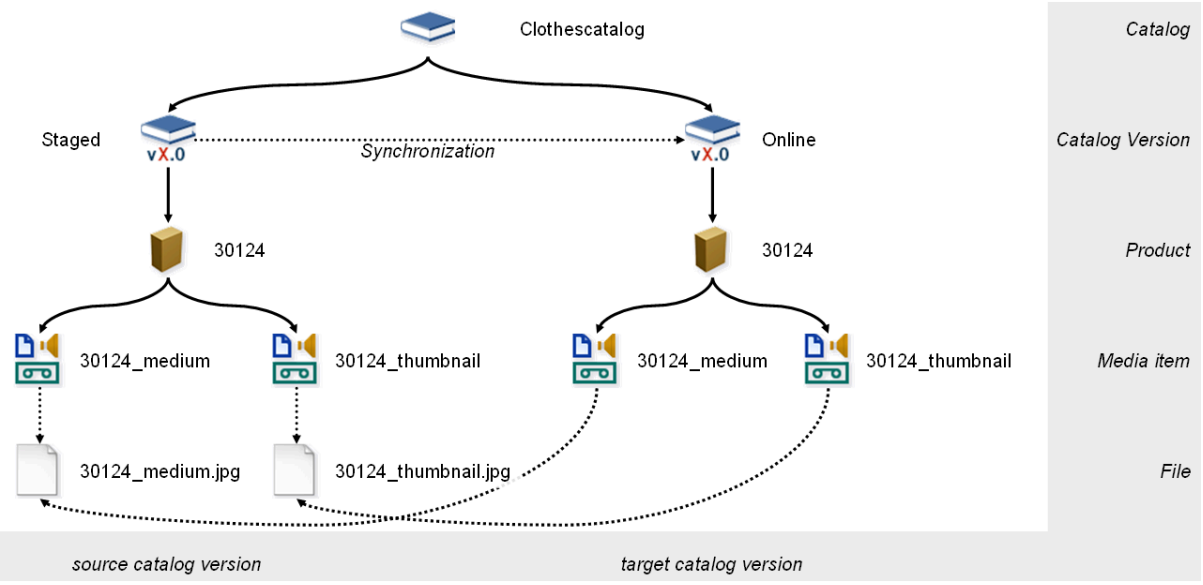
Media Items Synchronization

During the synchronization of a catalog version, all the media items in the source catalog version are copied. After synchronization, every media item is available twice: once in the source catalog version, and once in the target catalog version.

However, the files referenced by the media items will not be duplicated and are available only once. Instead of creating an individual copy of a Media-item-referenced file in the target catalog version, the Media item in the target catalog version holds a reference to the original file. The concept is similar to symbolic links. For more information, see Synchronizing Catalog Versions.

For example, in the graphic, there is one catalog version **Staged**, which is used as the source catalog version for synchronization. This **Staged** catalog version holds one product, **30124**. The product **30124** holds two Media items, **30124_medium** and **30124_thumbnail**. Each Media item has an individual image file, **`30124_medium.jpg`** and **`30124_thumbnail.jpg`**, respectively.

After a synchronization from **Staged** to **Online**, the product **30124** is available twice: once in the **Staged** catalog version, and once more in the **Online** catalog version. The Media items, **30124_medium** and **30124_thumbnail** are also available twice. However, the actual files represented by the media items, **`30124_medium.jpg`** and **`30124_thumbnail.jpg`** are also available once. The media items in the **Online** catalog version do not have an individual copy of the files, but holds a reference to the **`30124_medium.jpg`** and **`30124_thumbnail.jpg`** files instead.

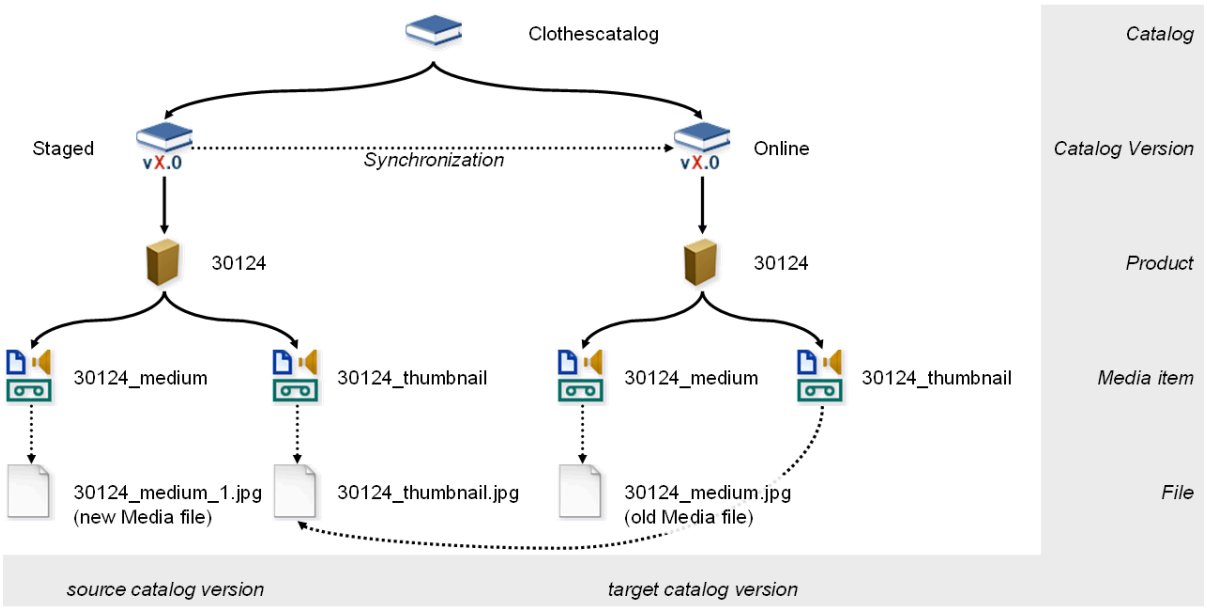


When you use different catalog versions in a synchronization scenario, you have to modify media every now and then to match an updated assortment. If you modify a media file in the source catalog version, SAP Commerce ensures that references to the original media file remain correct for target catalog versions. If the media file was simply overwritten and no references were updated in the process, then the target catalog version would still hold the reference to the media file of the source catalog version. As this file would have been overwritten, however, the target catalog version would then wrongly reference the new media file. In essence, you would end up with the target catalog version also showing the new media, when it should be displaying the old media.

Therefore, SAP Commerce modifies the references of the media item to hold the old media. Technically, this is done using the ensures that references to the original media file remain correct for target catalog versions. If the media file was simply overwritten and no references were updated in the process, then the target catalog version would still hold the reference to the media file of the source catalog version. As this file would have been overwritten, however, the target catalog version would then wrongly reference the new media file. In essence, you would end up with the target catalog version also showing the new media, when it should be displaying the old media.**dataPK** attribute of a media item. Do not modify this attribute or the reference mechanism may fail to work correctly. This reference consistency mechanism is fully automated and transparent. You do not have to worry about maintaining references manually. Just modify, upload, scale, convert your images to your liking, and SAP Commerce will deal with the references.

In the graphic, the original **`30124_medium.jpg`** ensures that references to the original media file remain correct for target catalog versions. media file in the **Staged** catalog version was replaced by a modified file. The **30124_medium** media item in the **Online** catalog version still references the original **`30124_medium.jpg`** file. SAP Commerce now makes sure that the **30124_medium** media item in the **Staged** catalog version and the **30124_medium** media item in the **Online** catalog version each reference the respective **`30124_medium.jpg`** file.

During a synchronization, the **30124_medium** media item in the **Online** catalog version will be modified to also reference the new **`30124_medium.jpg`** file. This means that after a synchronization, the previously used **`30124_medium.jpg`** file will no longer be used by either catalog version.



Grouping and Structuring Media Items

SAP Commerce enables a few means of grouping and structuring media items:

- Assigning a **Media Format** to media items to keep track of the format of the media
- Assigning multiple media to a **Media Container** to group the media logically
- Assigning multiple media formats to a **Media Context** to define a replacement list for media formats in a given context

Keeping Track of Formats Through Media Formats

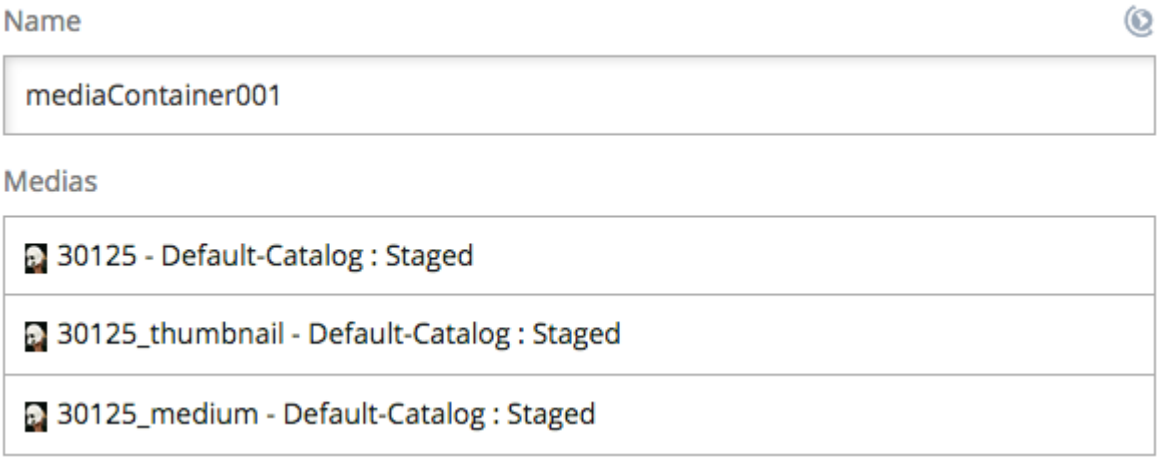
Each media item can have one **Media Format** assigned. A **Media Format** is simply a tag assigned to a media item, called a logical label. It does not invoke any functionality such as automatic conversion. For example, if a media item has a “50x50px” media format assigned to it, it doesn't mean the file is necessarily 50 by 50 pixels in size, nor has been converted or scaled to 50x50 pixels automatically. It only means that the media has the “50x50px” label assigned to it. You have to provide any conversion or re-scaling functionality explicitly by using a media asset management system.

Grouping Media Using Media Containers

Media items in SAP Commerce can be assigned to individual **Media Containers** for logical grouping. One single media item can only be assigned to one **Media Container,** which holds all media items that are different formats of one certain media item. For example, a **Media Container** can hold all media items of a certain product, no matter what file type or measurement of the actual files. In essence, a **Media Container** is a rule of which media items to use for which media format. A media item can only be assigned to a **Media Container** if the media item has a media format set.

A **Media Container** is intended primarily for images. Generally, product-relevant media items of which, only one version is available at any given time such as data sheets, spare-parts lists, and manuals in PDF format are not a useful field of application for **Media Containers**. However, if you have product data sheets in different versions such as individual layouts for ***Letter*** and ***DIN A4*** formats, then a **Media Container** may be useful.

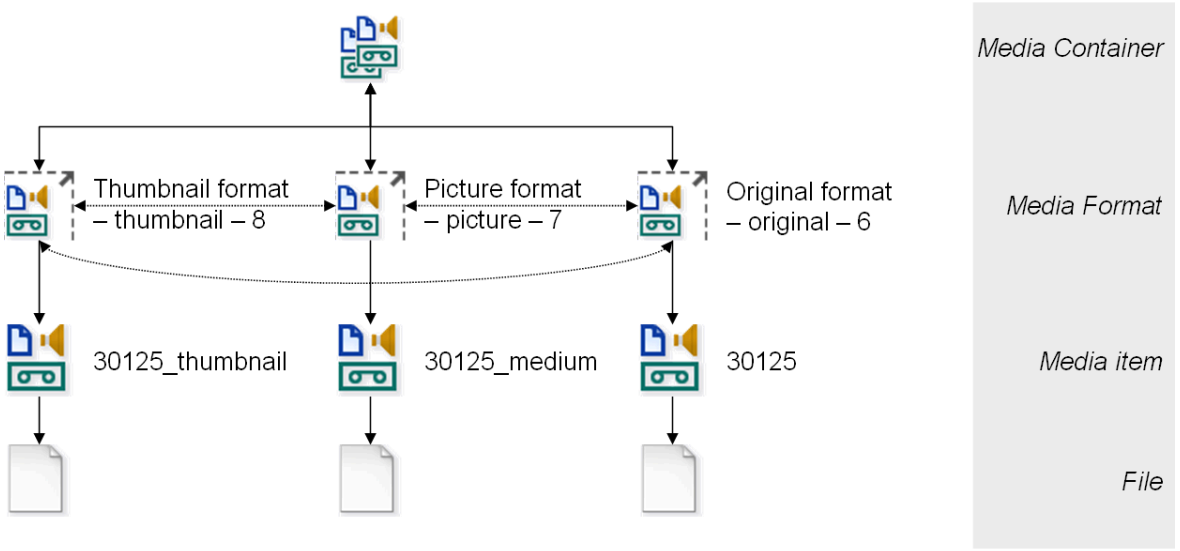
In this example, the **30125_medium** media item is in **Picture Format - picture - 7**, the **30125_thumbnail** media item is in **Thumbnail Format - thumbnail - 8**, and the **30125** media item is in **Original Media Format - original - 6**.



This graphic shows the basic structure. The media **30125_thumbnail**, and **30125_medium** are different versions (that is, media formats) of one single media: **30125**. Each of the media has a media format. The media container reflects the fact that these media are related by holding the media and their respective media format. The media container can look up which of its media is available in a given media format. That way, it's possible to get a media in a certain media format using the media container.

Using the media container, you do not have to explicitly retrieve the media in a certain media format. You can use the `MediaService` method, **`getMediaByFormat(MediaModel, MediaFormatModel)`** to get the media representation in a given format. The media service will look for the media container and return media in the specified format, if it is available. This is fully transparent and you do not have to use media container explicitly. See the example:

```
final MediaModel mediaModel = ...
final MediaFormatModel formatModel = ...
final MediaModel mediaInRequiredFormat = mediaService.getMediaByFormat(mediaModel, formatModel);
```



In the example above, if the media item, **30125** were asked to return its representation in the **Picture Format - picture - 7**, the media container would return the **30125_medium** media item.

For more information, see Handle Media Using the MediaContainer

Grouping Media Formats Using Media Contexts

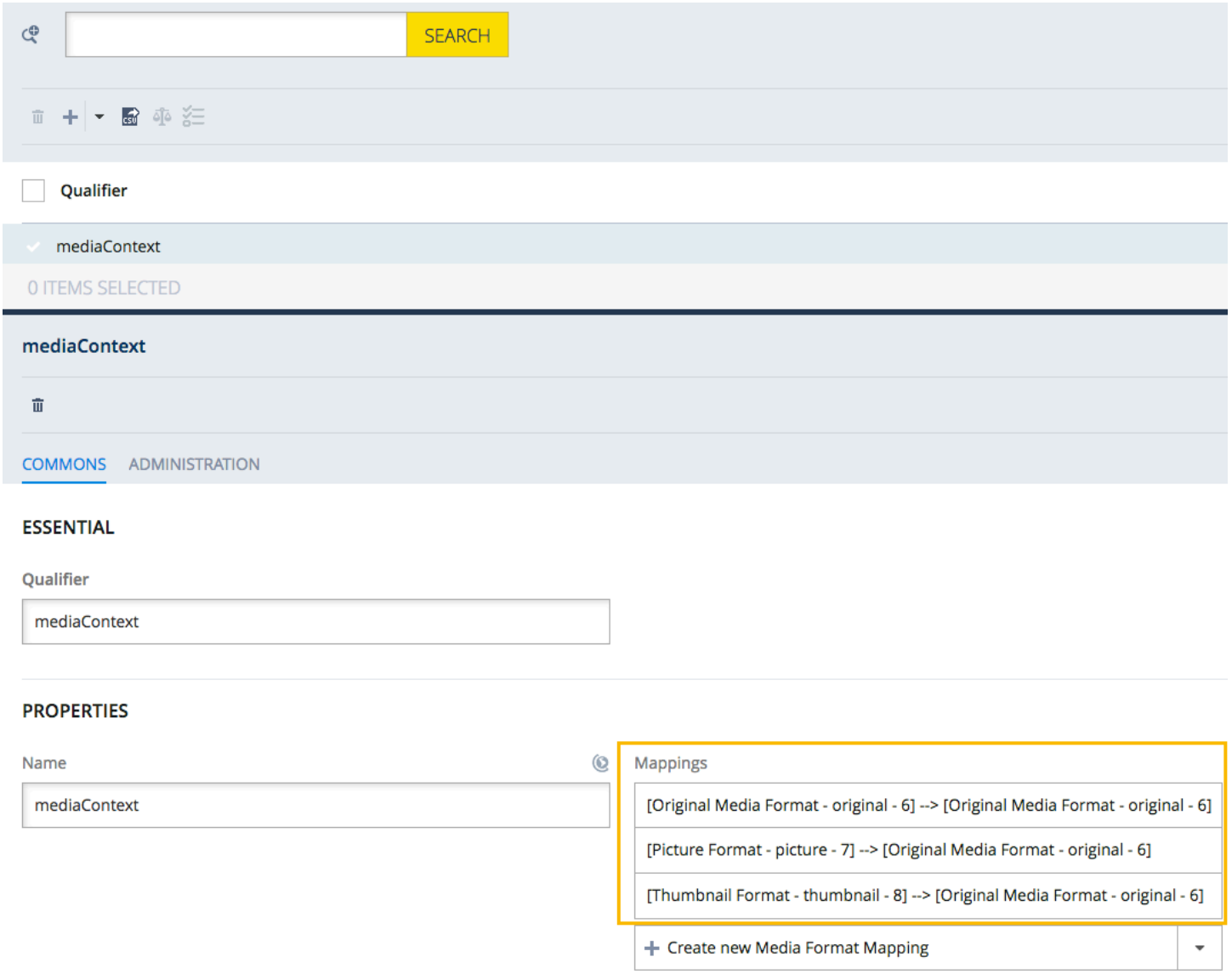
Media Contexts are to **Media Formats** what **Media Containers** are for **Media items**. It's a rule on which media formats to use instead of others in any given context. In essence, a **Media Context** tells the user, "if you run across media items in this media format, replace them with the media items in the other media format".

A **Media Context** therefore defines a mapping of media formats. One entry holds the media formats to replace, and the other entry holds the media format to use as a replacement.

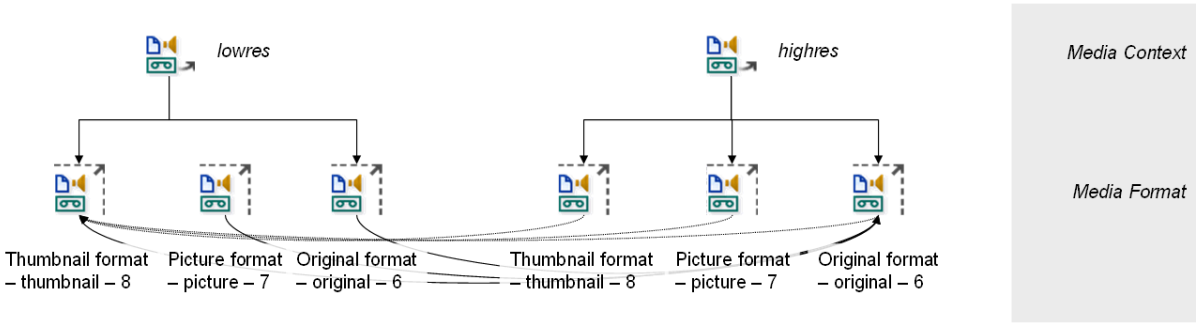
Media Context **high-res** in the screenshot defines these replacement rules:

Source Media Format	To be Replaced by Media Format
Original Media Format - original - 6	**Original Media Format - original - 6**
Picture Format - picture - 7	**Original Media Format - original - 6**
Thumbnail Format - thumbnail - 8	**Original Media Format - original - 6**

This assures that for all images, the media in the **Original Media Format - original - 6** media format is selected.



The following diagram shows the replacement table of the **high-res** Media Context and of a **low-res** Media Context in a graphical way:



ℹ Note

By factory default, media formats are unaware of **Media Contexts** and do not use them automatically. If you want to make sure that they are automatically used to replace Media Formats, you need to implement this functionality explicitly.

Referencing Files of Media Items

Learn how to reference media files stored by the mediaweb Extension or by a different system.

SAP Commerce differentiates between the following means of referencing files for media items:

- The file referenced by the media item is managed by the mediaweb Extension. Where the extension actually stores your files - in the mediaweb Extension itself, on the local computer, or on a network share, isn't important.
- Using a URL: Media items need to reference their files using a URL if the file is outside the mediaweb Extension, such as:
    - in an image database
    - in a media asset management system
    - in the classpath of a JAR file

Examples of Media Items

See examples of media items in SAP Commerce.

SAP Commerce Aspect	Usage of Media items
SAP Media Conversion	Integrated with an open source software ImageMagick to create, edit, compose, or convert almost all available bitmap image formats. Extraction of the metadata like IPTC and Exif metadata of bitmap images.
SAP WCMS Module	Images in paragraphs
CronJob	CronJob logs
ImpEx	ImpEx logs, (temporary) dump files
Product	Images

Media Folders

SAP Commerce allows you to create folders for organizing and managing your media. You can use the web server or application server to limit access to media files by media folder.

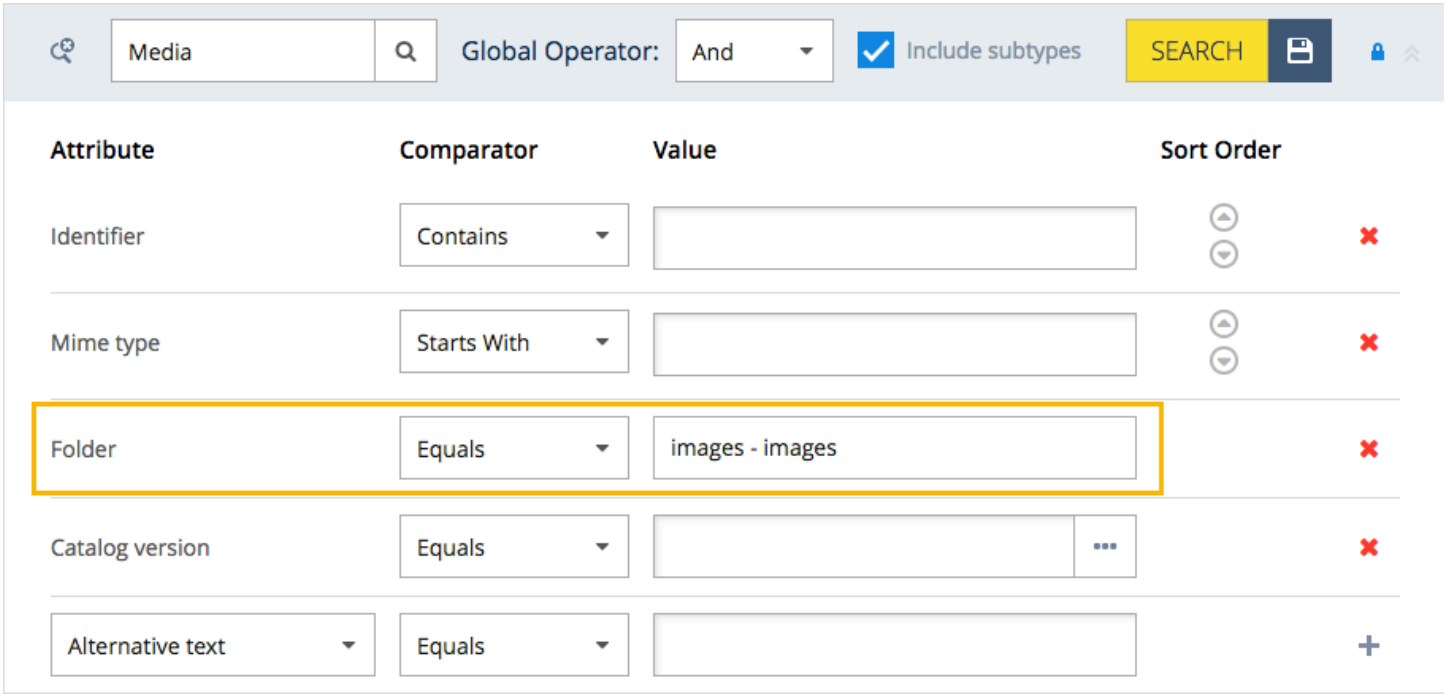
Media files for a web application can potentially contain security-sensitive data. Some files such as product images or animated banners, for example, should be available to all. For other files, such as internal presentations, ensure that access is granted only to authorized users. Directory-based authentication is a simple and effective means of access control.

Media items in SAP Commerce can be just about any file that can be referenced. Unlike other items such as products or users, which are pure Java objects, media items have a physical representation somewhere on the file system. They can include CSV files, images, Flash animations, ZIP files, or MP3 files.

Any media item in SAP Commerce consists of the following:

- It is a physical file located somewhere on the application server file system
- It is an SAP Commerce object of the `Media` type that holds a reference to the physical file

You can search for media by media folder in Backoffice.



Functional Principle of Media Folders

You can assign media files to media folders. Every media folder is an individual directory on the application server, and media assigned to a media folder are stored in that directory.

The physical location where media files reside depends on the application server in use. Refer to the documentation of your application server. In the SAP Commerce Server embedded Tomcat instance, the deployed media files are located in *<HYBRIS_DATA_DIR>*/media. The system generates the following default folders during initialization:

- **cronjob**
- **impex**
- **catalogsync**

You may also notice the existence of the `root` folder. It is not a physical folder on the application server file system. It is a name associated with the main media folder of the currently activated tenant. In other words, if you are working on the `master` tenant, the `root` folder is the same as the `sys_master` folder.

The system prevents you from removing the root media folder as it could cause severe issues impacting various aspects of SAP Commerce, including your storefront. Use the following property to disable this behavior:

```
media.is.root.folder.removable=true
```

Media folders in an SAP Commerce installation are tenant-specific; every tenant has its own media folders. For each tenant, the system creates a directory named `sys\_ <tenant>` in the deployment directory of the application server. For example, if you have the media folders `cronjob` and `impex`, and the tenants: `master` and `junit`, then the deployment directory of your application server will have the following directories:

- media
    - sys_master
        - cronjob
        - impex
    - sys_junit

- `cronjob`
- `impex`

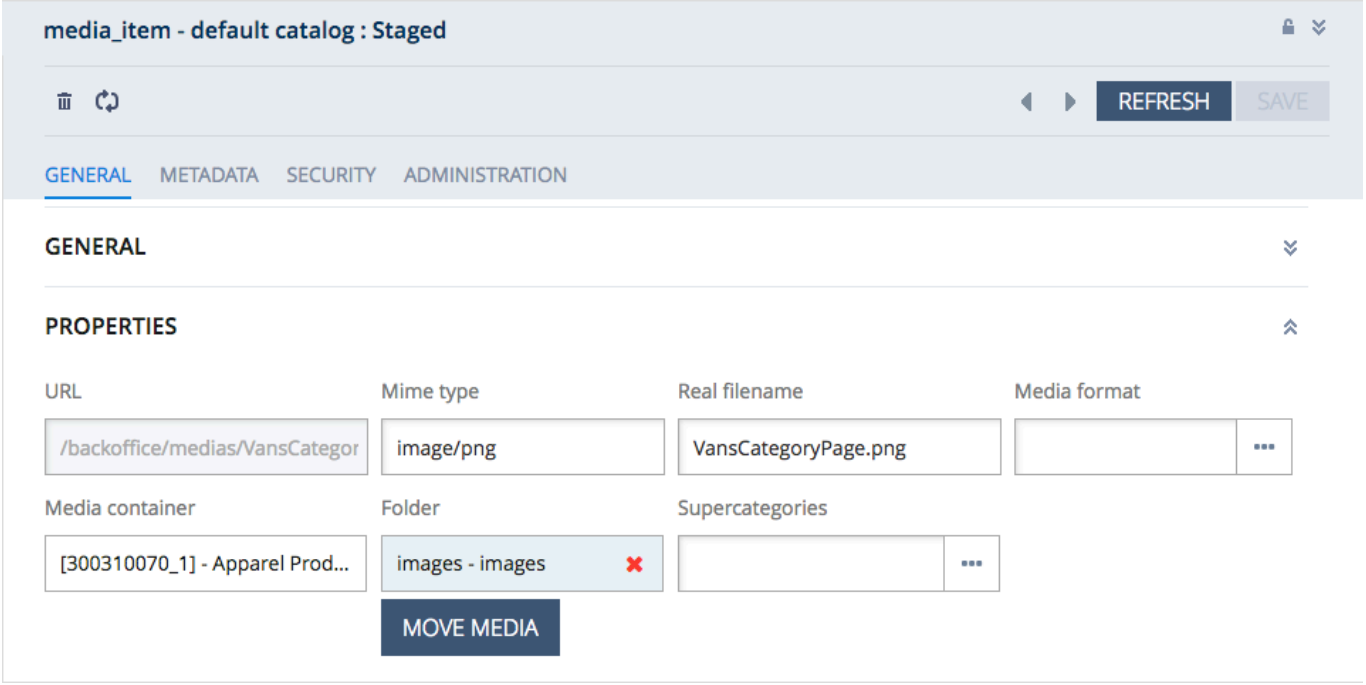
The SAP Commerce getter and setter methods are not affected by the MediaFolder a media is in. Calling the `getMedia(String)` method of `MediaService`, for example, returns every media item that matches the search pattern, regardless of whether the item is in a media folder or not.

Moving Media Items in Backoffice

Backoffice Administration Cockpit provides a Move Media button in the General tab of the media item details.

Procedure

1. Log into Backoffice and search for the media item you wish to move.
2. Select the media item to show the item details in the lower panel.
3. Remove the old folder location from the Folder field under Properties in the General tab.



4. Select a new folder location from the Folder menu.
5. Click Move Media.

Storing Many Files for One Media Item

Normally, a single **Media** item holds reference to only one file. However, you may need to have a **Media** item that refers to more than one file. For example, you have many pictures of one product and you want to have only one **Media** item for all these pictures.

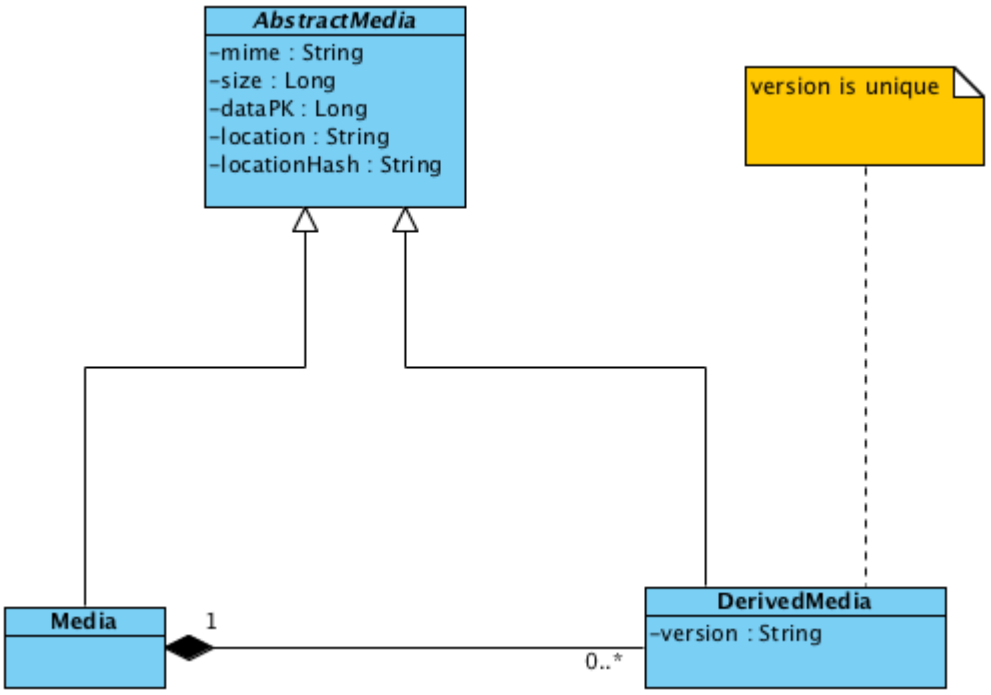
It is possible with **DerivedMedia** item and so called derived media functionality.

ℹ Note

Derived media functionality is only accessible from ServiceLayer.

Overview

DerivedMedia item is related to **Media** item. It is one to many relation, it means that basically one **Media** object can hold a collection of **DerivedMedia** objects. Each **DerivedMedia** holds its own binary data (file) and unique **version** name. **Version** is just a string identifier, it could be for instance name of data format, like **thumbnail** or **web**. Below you can see a diagram that describes the architecture:



Using API

To use derived medias functionality through the API, you have to use **MediaService**.

Add Additional Stream to Media

Example:

```
// Imagine that you have Media model already created and MediaService injected by Spring
final MediaModel media;
final MediaService mediaService;

// And you have a stream of data
final InputStream myData;

// You can upload new version of binary data
final String versionId = "myFancyVersion";
mediaService.addVersionStreamForMedia(media, versionId, myData);
```

Retrieve Stream for Media in a Specific Version:

Example:

```
// Imagine that you have Media model already created and MediaService injected by Spring
final MediaModel media;
final MediaService mediaService;

final String versionId = "myFancyVersion";
final InputStream stream = mediaService.getStreamForMediaVersion(media, versionId);
```

Get Direct URL Link for a Specific Version:

Example:

```
// Imagine that you have Media model already created and MediaService injected by Spring
final MediaModel media;
final MediaService mediaService;

final String versionId = "myFancyVersion";
final String mediaURL = mediaService.getUrlForMediaVersion(media, versionId)
```

Remove Specific Version from Media:

Example:

```
// Suppose that we have Media model already created and MediaService injected by Spring
final MediaModel media;
final MediaService mediaService;

final String versionId = "myFancyVersion";
mediaService.removeVersionForMedia(media, versionId);
```

Media URLs

Media URLs have different structures depending if the access to a media item is secured or not.

See structures of secured and unsecured URL patterns:

- Not secured URL pattern: ${server_address}:${port}/localMediaWebRoot/realFileName?context=encodedMediaContext, for example: `http://localhost:9001/medias/someFile.jpg?context=NAYDCL3IGAZC6ZTPN4XGU4DHHI5DU4LXMVZHI6JRGIZTINI`,

  > **Note**
  >
  > For details on how the context string **`encodedMediaContext`** is created, see the **`LocalMediaWebURLStrategy.java`** file.

- Secured URL pattern: URLs that use unique PKs of media items in URLs. This ensures that the correct item is used during a permission check. You can change the default `securemedias` prefix. For more details, see SecureMediaFilter in a Custom Web Application.

Pretty URLs

Use Pretty URLs to create SEO-friendly URL addresses.

Pretty URLs are disabled by default and need to be activated with the following key:

```
media.legacy.prettyURL=true
```

When the `media.legacy.prettyURL` property is set to `true` and the `realFileName` media attribute is set to a value with an extension name, the URL is generated in the Pretty URLs' format. If these conditions aren't met, the standard URL (with the `context` attribute) is generated. As a result, the Pretty and the standard URLs can be served from the media filters. The standard URL takes precedence over the Pretty URL when both are matching the requested URL.

> **Note**
>
> Extension names of media aren't required for the `LocalFileMediaStorageStrategy` strategy.

See an example of URLs:

- When you enable Pretty URLs: `/medias/sys_master/folder/h78/hd0/8796125986846/fileName.jpg`
- When you disable Pretty URLs: `/medias/someFileName.jpg?context=NAYDCL3IGAZC6ZTPN4XGU4DHHI5DU4LXMVZHI6JRGIZTINI`

Media Storage

The concept of media storage means how and where binary data of any Media item is stored and read.

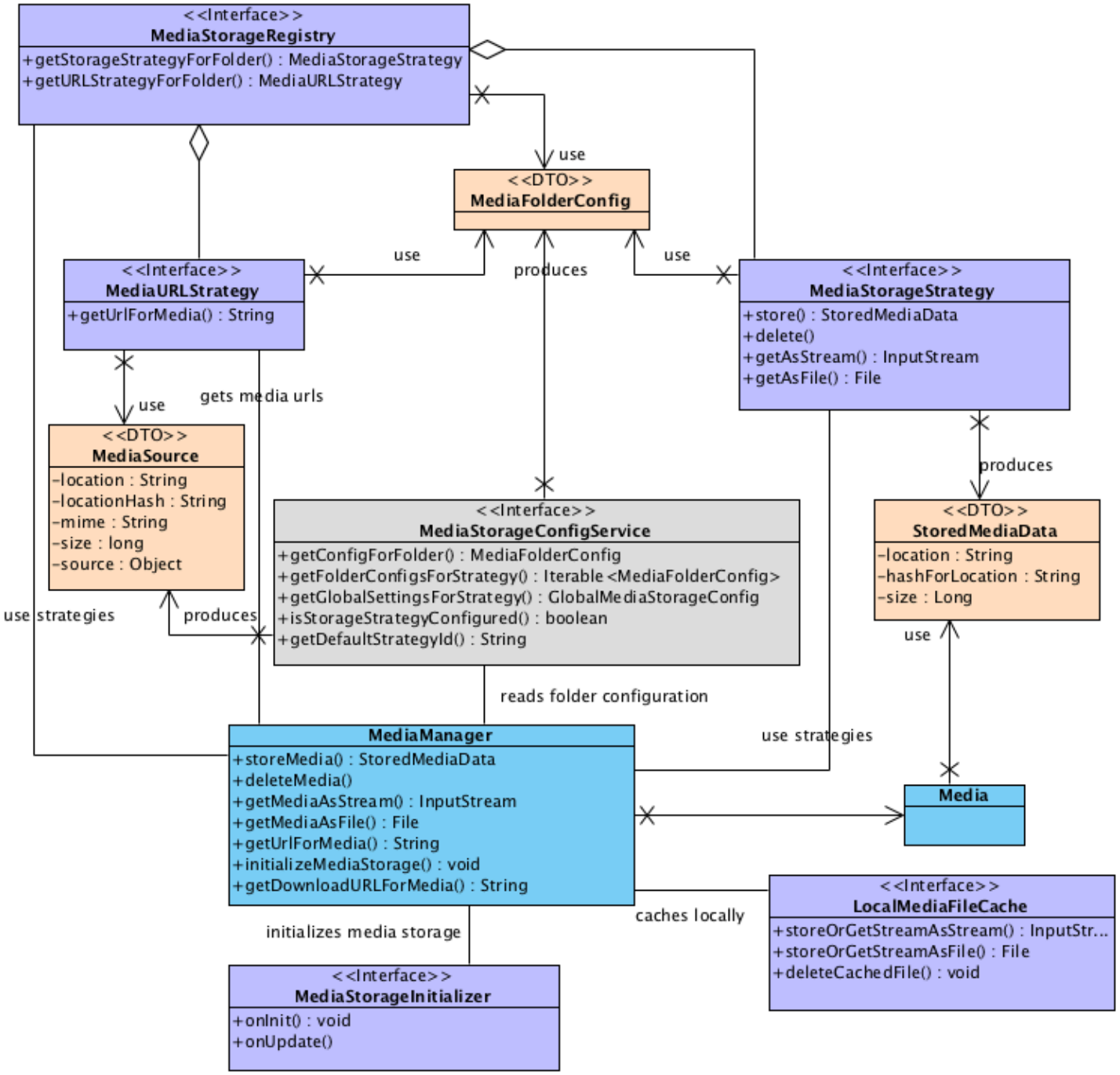
Storage can be local or can use remote services. Local storage means that binary data is stored as files in the same place where you deployed your SAP Commerce installation. With remote service storage, you can choose between three ready-to-use implementations for Amazon S3, Windows Azure Blob, and MongoDB GridFS. Different external storage services require different ways of communication and usage of different API or external libraries. Therefore, the architecture of media storage implementation allows you also to create your custom media storage strategy for a variety of external services.

Overview

The UML diagram below depicts the architecture of the media storage concept. It shows all important interfaces and interdependencies:

- **MediaStorageStrategy**: Performs actions on media data streams, like store, read and delete.
- **MediaURLStrategy**: Renders public web URLs.
- **MediaStorageRegistry**: Aggregates and serves all configured **MediaStorageStrategy** and **MediaURLStrategy** objects.
- **MediaStorageConfigService**: Keeps the configuration of each MediaFolder up to date.
- **LocalFileMediaCache**: Caches files locally.
- **MediaStorageInitializer**: Provides method for cleaning out media storage on fresh initialization of SAP Commerce.

MediaManager is the only connection between Media item and media storage.



Available Storing Strategies

Currently you can choose between the following storing strategies:

- **LocalFileMediaStorageStrategy**: Responsible for storing binary data in local file system storage. It is used by default. For more details, read Using Local Media Storage Strategy.
- **S3MediaStorageStrategy**: Responsible for storing binary data in Amazon S3. For more details, read Using Amazon S3 Media Storage Strategy.
- **WindowsAzureBlobStorageStrategy**: Responsible for storing binary data in Windows Azure Blobl Store. For more details, read Using Windows Azure Blob Media Storage Strategy.
- **GridFSMediaStorageStrategy**: Responsible for storing binary data in MongoDB. For more details read Using MongoDB GridFS Media Storage Strategy.

Configuration

Configuration for whole media storage infrastructure is based on properties and a special naming convention, which all media related properties must follow. Properties can be divided into three priority levels, whereby a level with a higher priority overrides a setting with a lower priority level:

- Default settings - priority 3
- Global settings (for specific strategy) - priority 2
- Folder settings - priority 1

Naming Convention

All property keys must follow certain naming convention rules, which are described in the sections below. Keys contain: a fixed, level-specific prefix, a variable part, and a suffix: a so-called **subKey**. Some keys also need to contain a Spring bean id of the specific storage strategy or a folder qualifier (the variable part of key, not always required). For better readability, parts of key will be colored as follows:

- prefix
- variable part
- subKey

Default Settings

It is possible to use the **`local.properties`** file to configure default settings for the media infrastructure. These settings will be read in if, for example, a particular folder does not have any media storage strategy attached: In this case the necessary strategy will be picked up from default settings. Another example is a situation where a folder is not configured at all: again, default settings will be used. You can see the default Platform configuration with its description in the main **project.properties** file.

> **i Note**
>
> **Do Not Modify Default Settings**
>
> Do not modify the default settings unless you really know what you are doing. Default settings provide tried and tested values for the media storage infrastructure.

For more information, see Configuration Properties.

All keys in this level follow naming convention:

```
media.default. subKey
```

An example of valid default settings for Local Media Storage Strategy is:

local.properties

```
media.default.storage.strategy=localFileMediaStorageStrategy
media.default.secured=false
media.default.hashing.depth=2
```

Global Settings

Sometimes, you can use specific global settings to activate certain strategies, for instance credentials. All keys at this level must conform to the following naming convention:

media.globalSettings. springBeanIdOfParticularStorageStrategy. subKey

This is an example of valid global settings for the S3 Media Storage Strategy:

local.properties

```
media.globalSettings.s3MediaStorageStrategy.accessKeyId=123456
media.globalSettings.s3MediaStorageStrategy.secretAccessKey=foobarbaz
media.globalSettings.s3MediaStorageStrategy.hashing.depth=3
media.globalSettings.s3MediaStorageStrategy.secured=true
```

In the example above, the media storage strategy identified by the id **s3MediaStorageStrategy** defines two additional subKeys: **accessKeyId** and **secretAccessKey**, overrides two subKeys from default settings: **hashing.depth** with the value **3**, and **secured** with the value **true**. With this configuration, all folders that are configured to use this strategy will be secured by default and will store data in a three-sub-folders structure.

Folder Settings

Each media folder can be configured separately, which makes the whole configuration very flexible. Bear in mind that folder-specific settings have precedence over default or global settings and that subKeys from default and global settings may be used in folder settings. All keys in this level must follow the naming convention:

media.folder. folderQualifier. subKey

An example of valid settings for a folder with the qualifier **foo** is:

local.properties

```
media.folder.foo.storage.strategy=s3MediaStorageStrategy
media.folder.foo.accessKeyId=78900
media.folder.foo.secretAccessKey=bazbarfoo
media.folder.foo.hashing.depth=4
media.folder.foo.secured=false
```

In the example above, the folder **foo** is configured to use **s3MediaStorageStrategy**, thus it can override its global settings. It overrides both **accessKeyId** and **secretAccessKey** (so it may use different credentials to S3 storage). It also overrides default settings for **hashing.depth** with value **4** and **secured** with the value **false**.

Validating Media Folder Names for Configuration

The following properties allow you to decide whether you want to validate your media folder names according to default, global, and folder settings:

```
media.default.mediaFolderName.validation=
media.globalSettings.<mediaStrategyName>.mediaFolderName.validation=
media.folder.<folderName>.mediaFolderName.validation=
```

The `media.default.mediaFolderName.validation` strategy is enabled by default. If you want to disable it, navigate to your `local.properties` file and use it with the value `false`:

```
media.default.mediaFolderName.validation=false
```

You can use these properties to enable configuration validation of media folder names configured with a given media strategy, for example the Amazon S3 media storage strategy:

```
media.default.mediaFolderName.validation=false
media.globalSettings.s3MediaStorageStrategy.mediaFolderName.validation=true
```

As a result, only names of media folders configured with the Amazon S3 media storage strategy are validated when the folders are being created. If they don't conform to the rules of acceptable folder names, the folders cannot be created.

You can also disable validation for folder settings for a given folder:

```
media.default.mediaFolderName.validation=true
media.folder.<mediaFolderName>.mediaFolderName.validation=false
```

Configure with MediaStorageConfigService

For easy management of the media storage configuration, the Platform is shipped with the **MediaStorageConfigService** interface and its default implementation **DefaultMediaStorageConfigService**, which can be taken from the Spring context by using **mediaStorageConfigService** Spring bean id. This class is the central point for reading in the whole configuration for the media storage infrastructure.

This service produces two DTO objects that can be used for reading the configuration:

- **MediaFolderConfig**: Used in most cases, keeps all required settings for a particular folder. All folder-specific overrides are taken into account here.
- **GlobalMediaStorageConfig**: This object keeps global settings for the globally configured storage strategy with the **media.default.storage.strategy** property.

In most cases, if you want to implement your own storage strategy, you will use the **MediaFolderConfig** DTO config object, which is passed all the time as the first parameter to every method in the two main interfaces: **MediaStorageStrategy** and **MediaURLStrategy**.

`MediaFolderConfig` DTO object itself provides a set of direct methods to read values specific to the default level of configuration:

```
// let's assume config object is ready to use
MediaFolderConfig config;

boolean secured = config.isSecured();
boolean cacheEnabled = config.isLocalCacheEnabled();
int hashingDepth = config.getHashingDepth();
```

as well as three generic methods to read any configuration subKey for its value:

```
// lets assume config object is ready to use
MediaFolderConfig config;

// Read plain String value
String strategyId = config.getValue("storage.strategy");
// Read Boolean value
Boolean secured = config.getValue("secured", Boolean.class);
// Read Integer value and return 10 as default when subKey has no value
Integer myValue = config.getValue("some.integer.subKey", Integer.class, Integer.valueOf(10));
```

and methods to read information about folder like its qualifier, storage strategy bean id which is used and configured url strategy bean Ids:

```
// lets assume config object is ready to use
MediaFolderConfig config;

String folderQualifier = config.getFolderQualifier();
String strategyId = config.getStorageStrategyId();
Iterable<String> urlStrategies = config.getURLStrategyIds();
```

ℹ Note

Keep in mind that all keys stored in configuration DTO objects are subKeys

All values for configuration subKeys stored in DTO objects are valid Java types. Conversion from String is done by converters which are registered for the following predefined subKeys:

- url.strategy
- local.cache
- secured
- hashing.depth
- cleanOnInit

System by default provides three converters:

- **IntegerValueConverter** - Spring bean ID: integerConfigValueConverter. Converts string representation of integer into Integer object.
- **BooleanValueConverter** - Spring bean ID: booleanConfigValueConverter. Converts string representation of boolean into Boolean object.
- **IterableValueConverter** - Spring bean ID: iterableConfigValueConverter. Converts comma separated string of values into Iterable object containing all values as strings.

By default all custom settings are stored without conversion. However, the user may want to use converters. It is just matter of registering bean of class **ConfigValueMappingRegistrar** as follows:

```
<bean id="someCustomSubKeyConversionMapping" class="de.hybris.platform.media.storage.impl.ConfigVal
        p:value-ref="integerConfigValueConverter" />
```

Assuming that my.fancy.subKey points to an Integer value, this mapping will register it to use **IntegerValueConverter**.

It is also possible to write your own converters by implementing **de.hybris.platform.media.storage.ConfigValueConverter** interface and registering it to any subKey as was shown in example above. All registered mappings are scanned and used automatically by **MediaStorageConfigService**.

Storing Media Files in a Structure of Subfolders

Each of the provided storage strategies may store medias in a structure of subfolders. The name of each subfolder consists of the following:

- **h** prefix.
- Hex representation of a hash code extracted from computation of 11111111 value and each byte from hash code representation of **Media** 's **dataPK**. Logical AND operation is used in computing.

For example, **dataPK** of a selected media item has the following byte representation: 00100001 00001000 00011101 10101100. By default two levels depth subfolders structure will be created. This means that two separate AND operations are performed on first two bytes and value 11111111:

- 10101100 AND 11111111 gives 10101100. Hex representation of extracted byte is AC
- 00011101 AND 11111111 gives 00011101. Hex representation of extracted byte is 1D

Then **h** prefix is added to each hex representation and in this way you get the names of each subfolder. Your media data will be stored in the following location: ***${HYBRIS_DATA_DIR}*****`media/sys_master/hac/h1d`**. When the file is stored in its physical

location, the relative location is cached in **location** property of the Media item and is used when read request for a Media comes next time. This location may look like in the following example:

```
sys_master/root/h44/h9a/8796157444126.jpg
```

This solution gives you a possibility to have 256 folders at each level. This means 65.536 folders can be created in total. If you assume that in each folder you store 1000 files, it gives us over 65 million files in total.

To use this algorithm in a custom storage service you have to use **HierarchicalMediaPathBuilder** class. Here is an example:

```
HierarchicalMediaPathBuilder pathBuilder = HierarchicalMediaPathBuilder.forDepth(2);

final String mainFolderPath = "sys_master/root/";
final String yourMediaId = "12345678.jpg";

final String path = pathBuilder.buildPathForMedia(mainFolderPath, yourMediaId, subDirDepth);

// Now returning path looks like this: sys_master/root/hd1/hc2/h00/
```

Migrate to Structure of Subfolders

It is possible to migrate any folder structure after changing **MediaFolder** configuration. You have to do the following:

1. Create an instance of the **MediaFolderStructureMigrationCronJob** cron job
2. Assign it to the **mediaFolderStructureMigration** job
3. Provide the name of the **MediaFolder** that have to be migrated
4. Execute the cron job

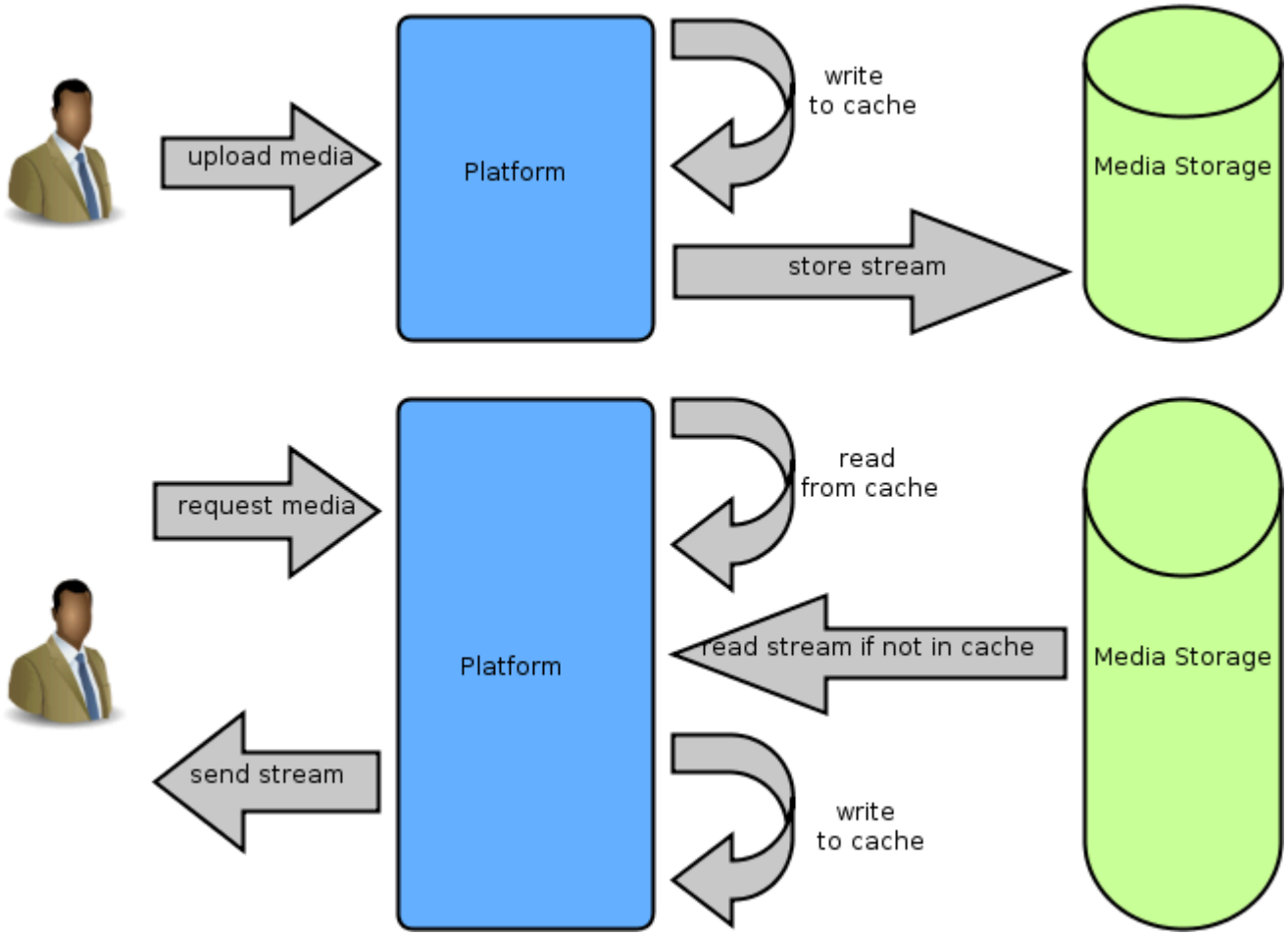
You can do it in Backoffice or using code like in the following example:

```
// Assume you have already injected services
final ModelService modelService;
final MediaService mediaService;
final CronJobService cronJobService;

// You want to migrate root folder
final MediaFolderStructureMigrationCronJobModel cj = modelService.create(MediaFolderStructureMigrat
cj.setMediaFolder(mediaService.getRootFolder());
// Set mediaFolderStructureMigration job which is already defined in the initialized system
cj.setJob(cronJobService.getJob("mediaFolderStructureMigration"));
modelService.save(cj);

// Perform configured CronJob
cronJobService.performCronJob(cj);
```

The **mediaFolderStructureMigration** job looks for all **Media** files related to your **MediaFolder** and moves them to proper subfolder structure. The whole process uses the default paging strategy (100 items on each page) so even folders with a large number of files can be migrated safely.

Local File Caching

External storages may affect performance when requesting files, thus it is recommended to use a local file caching mechanism. Each storage strategy allows enabling of local file caching. A general overview of the caching mechanism is shown in the picture below:



The idea of caching files locally is described below:

1. User is uploading data:
    a. Data stream is locally cached as file on the local file system
    b. Data stream is sent to configured storage through storage strategy
2. User is requesting for data:
    a. Local cache is requested for the data
        - If file exist, then stream is sent to the user
        - If file does not exist, then external storage is requested for the data that is cached locally and sent as stream to the user

Configure Local File Caching

You can provide the following changes in the default configuration by adding the following properties to the `local.properties` file:

- Control subdirectory name:

  ```
  media.default.local.cache.rootCacheFolder=localCache
  ```

- If you like to store cache files separately for each media folder (for example when you want to store cache files on separate disks or partitions), you can do it as follows:

  ```
  media.folder.myFolder1.local.cache.rootCacheFolder=myFolder1-cache
  media.folder.myFolder2.local.cache.rootCacheFolder=myFolder2-cache
  ```

- Disallow storing files with names longer than 255 characters in local cache:

  ```
  prevent.longfilenames.localcache=false
  ```

ℹ Note

Enabling local file caching makes sense only for storing strategies that use **LocalMediaWebURLStrategy**.

⚠ Caution

Declaring max cache size via **media.default.local.cache.maxSize** requires to declare it higher then an expected file size. Uploading file with size greater than declared max cache size will cause an exception.

Evictions

Evictions to the cache will occur automatically but you can also evict any cached file by hand by calling:

```
// assuming that localMediaFileCacheService and mediaStorageConfigService are taken from Spring cor

MediaFolderConfig folderConfig = mediaStorageConfigService.getConfigForFolder("folderQualifier");
String locationInStorage = "some/location/foo.jpg";
localMediaFileCacheService.removeFromCache(folderConfig, locationInStorage);
```

When eviction is done automatically underlying logic will take care about cached file removal. File will be removed only when last client will release the usage by closing stream. One exception to this rule is when client will ask for the real File object to cached file instead of InputStream. In this case such cached file when evicted will not be removed from the cache but special zero-bytes marker will be created in the cache directory. Such marked files will stay in the cache to next restart of the Platform and then removed on Platform startup. It is also save to delete such files (together with marker files) directly using operating system commands. Below is example of cached file and its eviction marker:

```
/h0f/h72/ODc5NjE4NzM2MTMxMC8zMDgzMS5qcGc=__H__c9690cd1-10e0-436b-9ce0-b509a5b0c721.bin
/h0f/h72/ODc5NjE4NzM2MTMxMC8zMDgzMS5qcGc=__H__c9690cd1-10e0-436b-9ce0-b509a5b0c721.bin.EVICTED
```

Media Location Verification

All media storage strategies provided by SAP Commerce compute the hash from the name of the MediaFolder, location of the Media item and a special **salt** string configured in **`project.properties`** file under **media.storage.location.hash.salt** property.

⚠ Caution

Necessary Configuration for Security Reason

It is highly recommended for security reason to change hash **salt** property for each installation of SAP Commerce to some very unique value.

The computed hash is stored in **locationHash** of the Media item and is then verified by the **MediaFilter** when serving medias by the **mediaweb** extension. It means that this mechanism works only when the **LocalMediaWebURLStrategy** is used for creating media URLs. If the **MediaFilter** does not verify the hash provided in request, then a **HttpServletResponse.SC_FORBIDDEN** response is send.

ℹ Note

Avoid Media Location Verification

You can avoid media location verification:

- If you do not want to have this option, then you can avoid it when using **S3MediaStorageStrategy** and **WindowsAzureBlobStorageStrategy**. You just need to use corresponding URL strategies instead of **LocalMediaWebURLStrategy**:

- **s3MediaURLStrategy** with **S3MediaStorageStrategy**
- **windowsAzureBlobURLStrategy** with **WindowsAzureBlobStorageStrategy**

- If you do not want to have this option while using your custom storage strategy, but you want to use **LocalMediaWebURLStrategy** as there is no need to implement a custom URL strategy, then just do not implement a hash computation in your custom storage strategy. In this case the **LocalMediaWebURLStrategy** constructs URLs with a special marker, which will bypass the verification procedure.

Detecting Invalid Salt Values

The following property allows you to define the behavior of SAP Commerce when an invalid value of salt is detected during system initialization:

```
media.default.storage.location.hash.illegalValueBehavior=ignore
```

There are three supported behaviours:

- `error`: an exception is thrown and initialization fails before making any changes to the existing type system
- `warn`: a warning is printed in logs, but initialization is not stopped
- `ignore`: nothing happens (default)

Initializing Media Storage on Initialization or Update

Each time user is executing a full initialization or update of the SAP Commerce, media storage may be initialized in a special way - for instance it can be cleaned out. Because different kind of storages may provide different ways of initialization, mostly because of different APIs, you can use the **MediaStorageInitializer** interface, which contains two methods: **onInit()** and **onUpdate()**. Any implementation of the **MediaStorageInitializer** is used during the initialization or update process automatically.

Creating Custom Media Storage and URL Strategy

You can create your own media storage strategy. In the simplest scenario you can just implement the **MediaStorageStrategy** and use the existing **LocalMediaWebURLStrategy** for rendering URLs. If specially crafted URLs are needed, then the **MediaURLStrategy** must be implemented as well.

You can also use the following interfaces and implementations that help developing the media storage related logic:

- **MediaStorageInitializer**: If you want to execute any logic during a fresh initialization or update, implement this interface and register in the Spring context.
- Media location verification: Inject **MediaLocationHashService** into your custom **MediaStorageStrategy** and then use it in the **storeMedia** method as follows:

```
@Override
public StoredMediaData storeMedia(final String folderQualifier, final String mediaId, final M
{
  // Storage logic goes here. Here locationOfMedia and size should be accessible
  // locationHashService should be injected by Spring
  return new StoredMediaData(locationOfMedia, locationHashService.createHashForLocation(folde
}
```

Related Information

Media

Using Local Media Storage Strategy

In this strategy, all data are stored locally in local file system storage defined by `media.read.dir` and `media.replication.dirs` properties. The default configuration is that all files for all MediaFolders use Local Storage strategy.

Configure Local Storage Strategy

You can change default configuration by putting in `local.properties` file custom values for the following properties:

`local.properties`

```
media.default.storage.strategy=localFileMediaStorageStrategy
media.default.hashing.depth=2
// Where media files are written. Instead of below location you can also use '/usr/var/media' or '\
media.replication.dirs=c:\\media
media.read.dir=c:\\media
```

It is possible to mix different strategies per MediaFolder. To configure your custom MediaFolder to store binary data of a Media item on the local file system, provide custom values for set of properties in your `local.properties` file. Below you have an example of configuration provided for folder `localMedias1`:

`local.properties`

```
media.folder.localMedias1.storage.strategy=localFileMediaStorageStrategy
media.folder.localMedias1.url.strategy=localMediaWebURLStrategy
media.folder.localMedias1.hashing.depth=1
```

File Location Verification in LocalFileMediaStorageStrategy

It is now impossible to return media from different folders by tampering their URL. When media file location doesn't match their folder, `MediaInvalidLocationException` is thrown. If you wish to turn off this behavior, add the following property to your `local.properties` file:

```
local.file.media.storage.folder.integrity.verification.enabled=false
```

Secure Media Access

Secure media access means giving access to media after checking permissions. In other words, you cannot access specific media by just guessing its URL. Permission check while accessing medias is ensured by a special filter, which you can add to your custom Web application.

Overview

Secure access to a media is ensured by checking the access rights, which means that you can't access the media by just guessing its URL. Secured media follows a different URL pattern than unsecured media. This URL pattern can be customized to some extent. Checking access rights is ensured by a special filter that is responsible for checking with the media permission service if access is granted and:

- If access is granted, you receive the `Media` item.

- If access is denied, you receive a response that access is forbidden.

It is possible because each `Media` item has its own access control list and can store access rights for any principal.

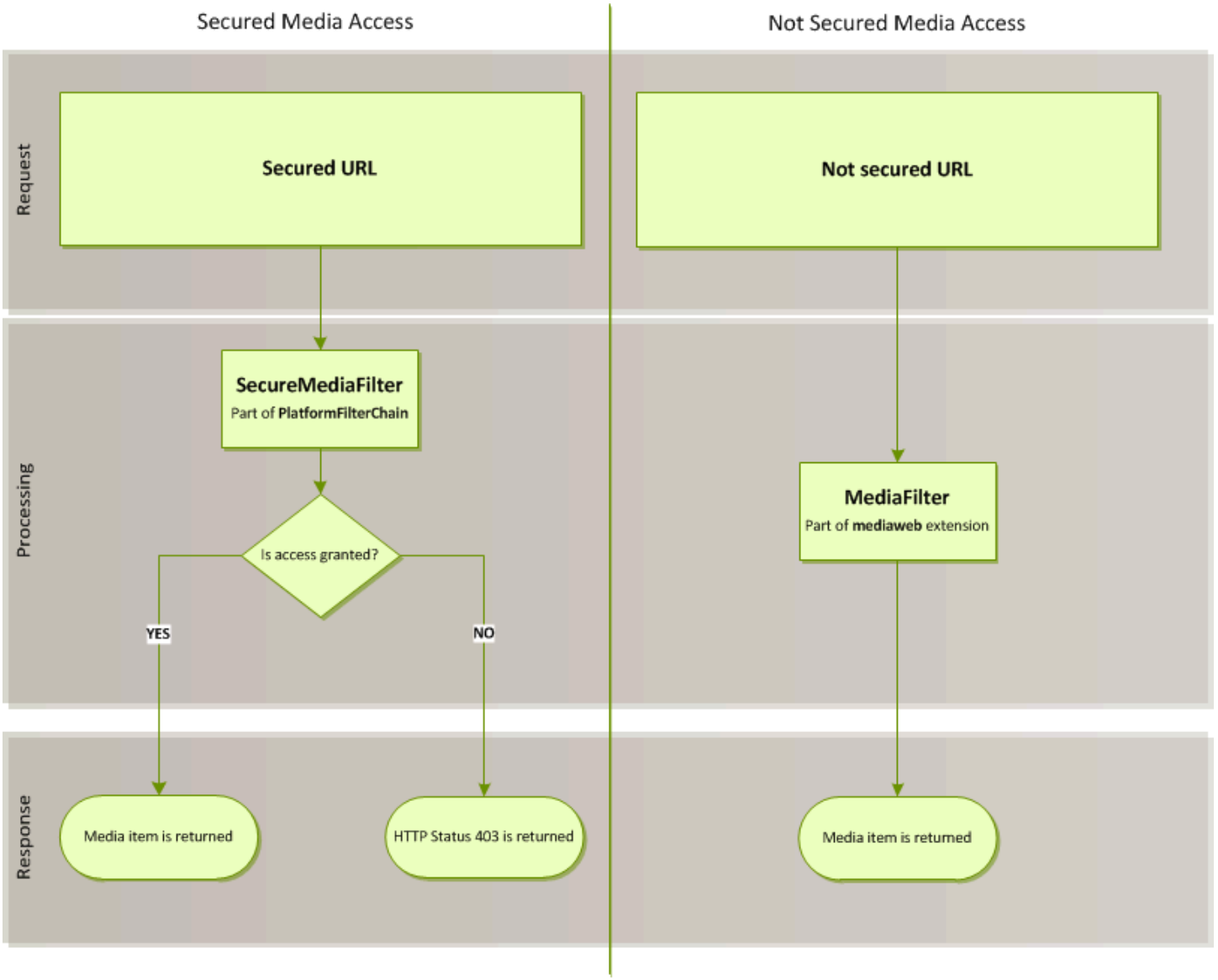


Figure: Comparison between secured and not secured access to a `Media` item.

ℹ Note

Platform also provides the `WebAppMediaFilter` filter that unifies the way media is served. It handles both secure and non-secure types of media as opposed to `MediaFilter` and `SecureMediaFilter`.

Keep in mind that the `SecureMediaFilter` is in every Web application in SAP Commerce and you can also add it to your custom Web application as part of the `PlatformFilterChain`. Similarly, the `MediaFilter` is always a part of a separate Web application that is a part of the `mediaweb` extension.

It is ensured that the right filter starts processing the request for a `Media` item by:

- Having different URL patterns for secured and unsecured `Media` items. In other words, `MediaFilter` will not react to secured URLs.
- Checking if the `Media` item belongs to a folder that is enabled for secure media access.

Enabling Secure Media Access

You can enable secure media access for specific Media folder by putting in your `local.properties` file the following property set to `true`:

`local.properties`

```
media.folder.<folderName>.secured=true
```

It means that only secure URL will be rendered for each `Media` item stored in these folders. It also means that access to these medias will be filtered only by the `SecureMediaFilter`.

Related Information

Media Guide
hybris Platform Filters
Users in the hybris Platform
Managing and Checking Access Rights

SecureMediaFilter in a Custom Web Application

The `SecureMediaFilter` should be used as a part of `PlatformFilterChain`. See the example showing how to add `SecureMediaFilter` to an existing filter chain.

To start using this filter, create a filter chain in your Web application. For instructions on how to do it, read hybris Platform Filters, section **Configuring Existing Filters**. Below you can find an example of how the `SecureMediaFilter` is added to the existing filter chain it the `mam` extension:

`mam-web-spring.xml`

```
<bean id="mamFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterChain">
  <constructor-arg>
   <list>
    <ref bean="mamRedirectFilter"/>
    <ref bean="sessionFilter"/>
    <ref bean="mamDataSourceSwitchingFilter"/>
    <ref bean="mamCatalogVersionActivationFilter"/>
    <!-- Add reference to SecureMediaFilter -->
    <ref bean="mamSecureMediaFilter"/>
     </list>
  </constructor-arg>
</bean>

<!-- Add bean declaration for SecureMediaFilter -->
<bean id="mamSecureMediaFilter" class="de.hybris.platform.servicelayer.web.SecureMediaFilter">
    <property name="mediaPermissionService" ref="mediaPermissionService"/>
    <property name="modelService" ref="modelService"/>
    <property name="userService" ref="userService"/>
    <property name="mediaService" ref="mediaService"/>
    <!-- <property name="secureMediaToken" value="securemedias"/>  Change the value and remove
    comment to have different prefix for secure URL -->
    <aop:scoped-proxy/>
</bean>
```

Managing Permissions

Managing permissions for `Media` items is based on the existing permission services for managing and checking access rights. It is possible to set permission not only for a type but also for a specific item. If no permission is set for a specific item, the item's type

level is checked instead.

Using API

Use the `MediaPermissionService` for the following tasks:

- Checking if the access to a `Media` item is granted or denied for a given principal:

```
// Assume you already have the following objects
MediaModel media;
UserModel user; // Can be for example fetched with userService.getCurrentUser()

//Check if given user is permitted to read the given *Media* item
boolean isGranted = mediaPermissionService.isReadAccessGranted(media, user);
```

- Granting or denying access to a `Media` item for a given principal.

```
// Assume you already have the following objects
MediaModel media;
UserModel user; // Can be for example fetched with userService.getCurrentUser()

//Grant the read permission for a user on a given *Media* item
mediaPermissionService.grantReadPermission(media, user);

//Deny the read permission for a user on a given *Media* item
mediaPermissionService.denyReadPermission(media, user);
```

ℹ Note

Grant and Deny Permission for a User on a Given Media Item

The principal cannot have grant and deny permissions assigned to the same `Media` item at the same time. The deny permission overrides the grant permission.

Using Backoffice

You can grant or deny access to a `Media` item for a give principal by opening specific `Media` item and going to **Security** tab.

Using ImpEx

Below you can find the example of an ImpEx import script for granting access to a `Media` item with code `1017895.jpg` for the `editor` principal:

```
INSERT_UPDATE media; code[unique=true]; catalogVersion(catalog(id),version)[unique=true]; permitte
;1017895.jpg; clothescatalog:Staged;editor;
```

ℹ Note

Following the example above, if some other principal already had access to `1017895.jpg` `Media` item, then this import would replace him with `editor` principal. In other words, the list of principals provided in the script is not a list of principals that will be added to existing ones, but a target list of principals that should have access.

Securing MediaFilter

The `media.allowed.extensions.for.ClassLoader` property enables you to explicitly specify extensions of files that you allow MediaFilter to serve from the Platform classpath. Use this property to prevent MediaFilter from serving files you think users shouldn't have access to.

MediaFilter can serve files from the Platform classpath. The resource paths that point to such files start with the `/fromjar/` url segment and are loaded by the classloader. While useful in certain scenarios, this feature could possibly allow users to download files that they shouldn't have access to.

You can specify the types of files that can be loaded from the classpath by setting the `media.allowed.extensions.for.ClassLoader` property. This property expects a coma separated list of file extensions that MediaFilter will be able to load (and serve) with the classloader:

```
# default settings
media.allowed.extensions.for.ClassLoader=jpeg,jpg,gif,bmp,tiff,vcard,templ,tif,csv,eps,pdf,png

# completely disabled
media.allowed.extensions.for.ClassLoader=
```

To make sure that, for example, no `essentialdata.csv` can be downloaded, exclude the csv extension from the list:

```
media.allowed.extensions.for.ClassLoader=jpeg,jpg,gif,bmp,tiff,vcard,templ,tif,eps,pdf,png
```

Now no csv files can be served from the classpath by MediaFilter.

WebAppMediaFilter

The `WebAppMediaFilter` filter unifies the way media is served. It handles both secure and non-secure types of media as opposed to `MediaFilter` and `SecureMediaFilter`.

Platform comes with a web application called `mediaweb`. `mediaweb` is fully responsible for all local media serving. Any request for media always hits `MediaFilter` that assures media is served to you properly. `MediaFilter` is registered in the `mediaweb` `web.xml` file under the `/medias` url pattern (which is also a webroot of the `mediaweb` extension). As a result, any URL that starts with `/medias` hits `MediaFilter` in the `mediaweb` extension. That means that your own web application is not responsible for media serving.

However, Platform uses the concept of **secure** media. Such media should only be served to authenticated and authorised users. For this to work, Platform provides another filter called `SecureMediaFilter`. This filter must be added to the common Spring `PlatformFilterChain`. This leads to inconsistency in handling media: non-secure media is served via `MediaFilter` located in the `mediaweb` extension while secure media is served inside your web application via `SecureMediaFilter`. To remove this inconsistency, Platform now provides `WebAppMediaFilter` that serves both secure and non-secure types of media.

Using WebAppMediaFilter

To use `WebAppMediaFilter`, first register it as a regular Spring bean in your configuration. Use `de.hybris.platform.servicelayer.web.WebAppMediaFilter` as a class implementation:

```
<bean id="yourAppMediaFilter" class="de.hybris.platform.servicelayer.web.WebAppMediaFilter">
    <property name="mediaPermissionService" ref="mediaPermissionService"/>
    <property name="modelService" ref="modelService"/>
    <property name="userService" ref="userService"/>
```

```
    <property name="mediaService" ref="mediaService"/>
</bean>
```

Note

Remember to register it under a unique ID. There may be other web applications in the same SAP Commerce installation that are also using `WebAppMediaFilter`.

Next, add your Bean into the existing `PlatformFilterChain` configuration, or create a new configuration:

```
<bean id="yourAppPlatformFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterCha
   <constructor-arg>
      <list>
         <ref bean="log4jFilter"/>
         <ref bean="sessionFilter"/>
         <!-- New WebAppMediaFilter Bean ref -->
         <ref bean="yourAppMediaFilter"/>
      </list>
   </constructor-arg>
</bean>
```

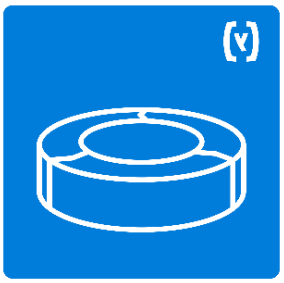
If you haven't already registered `PlatformFilterChain` in your web application, add the following code snippet to your `web.xml`:

```
<filter>
    <filter-name>yourAppPlatformFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
    <filter-name>yourAppPlatformFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Since `PlatformFilterChain` is registered under the `/*` url pattern, it will be hit for each request. As a result, both non-secure and secure media will be served by `WebAppMediaFilter`.

yempty Template

We have improved the `yempty` template and now it contains `WebAppMediaFilter` already registered for you. Whenever you create a new extension using `yempty`, you get `WebAppMediaFilter` configured by default. Additionally, `yempty` contains a fully configured Spring MVC application, sample Controller, Service and Spring Security configuration. The following screenshot shows a newly generated extension using `yempty`:



Welcome to "foobar" extension

Getting started

This extension was generated using yEmpty template. It contains basic Spring MVC with sample Controller as well as WebAppMediaFilter enabled. Now you should go through the crucial configuration files and adjust them to your needs. Feel free to remove default controller and jsp pages.

- **resources/foobar-items.xml** - here you can model your items
- **resources/foobar-spring.xml** - here you can define your services
- **web/webroot/WEB-INF/config/foobar-spring-mvc-config.xml** - here is a Spring MVC related configuration
- **web/webroot/WEB-INF/config/foobar-web-app-config.xml** - here you can define web related services, facades etc.
- **web/webroot/WEB-INF/config/foobar-spring-security-config.xml** - here you can configure your Spring Security settings
- **web/webroot/WEB-INF/web.xml** - here you can configure filters, servlets etc.
- **web/webroot/WEB-INF/views** - here you can keep your jsp pages
- **web/webroot/static** - here you can keep your static files, javascripts, css etc.

This extension comes with basic Spring Security configuration which is disabled by default. If you want to enable it go to the *web/webroot/WEB-INF/web.xml* file and uncomment filter named **springSecurityFilter**, its filter mapping and restart application.

Determining Content Type

It is possible to determine if the content type returned by Media filters is taken from the database or the context. The behavior depends on the value of the following property:

```
media.filter.contentType.fromDB=true
```

The following values are supported:

- `true` - the content type is taken from the database. It requires a query to the database, which can have a negative impact on performance. It is the default value of the property.
- `false` - content type is taken from context provided in the request.

This setting is, however, respected only if the request context does not contain the hash that verifies the provided content type. If the hash verifies the content type in the context, this value is used regardless of the value of the `media.filter.contentType.fromDB` property.

Enabling mediaweb.webroot Property for WebAppMediaFilter

To make `WebAppMediaFilter` conscious of the `mediaweb.webroot` property and, as a result, make `WebAppMediaFilter` handle media's inbound requests, navigate to your `local.properties` file and add the following property with the value `true`:

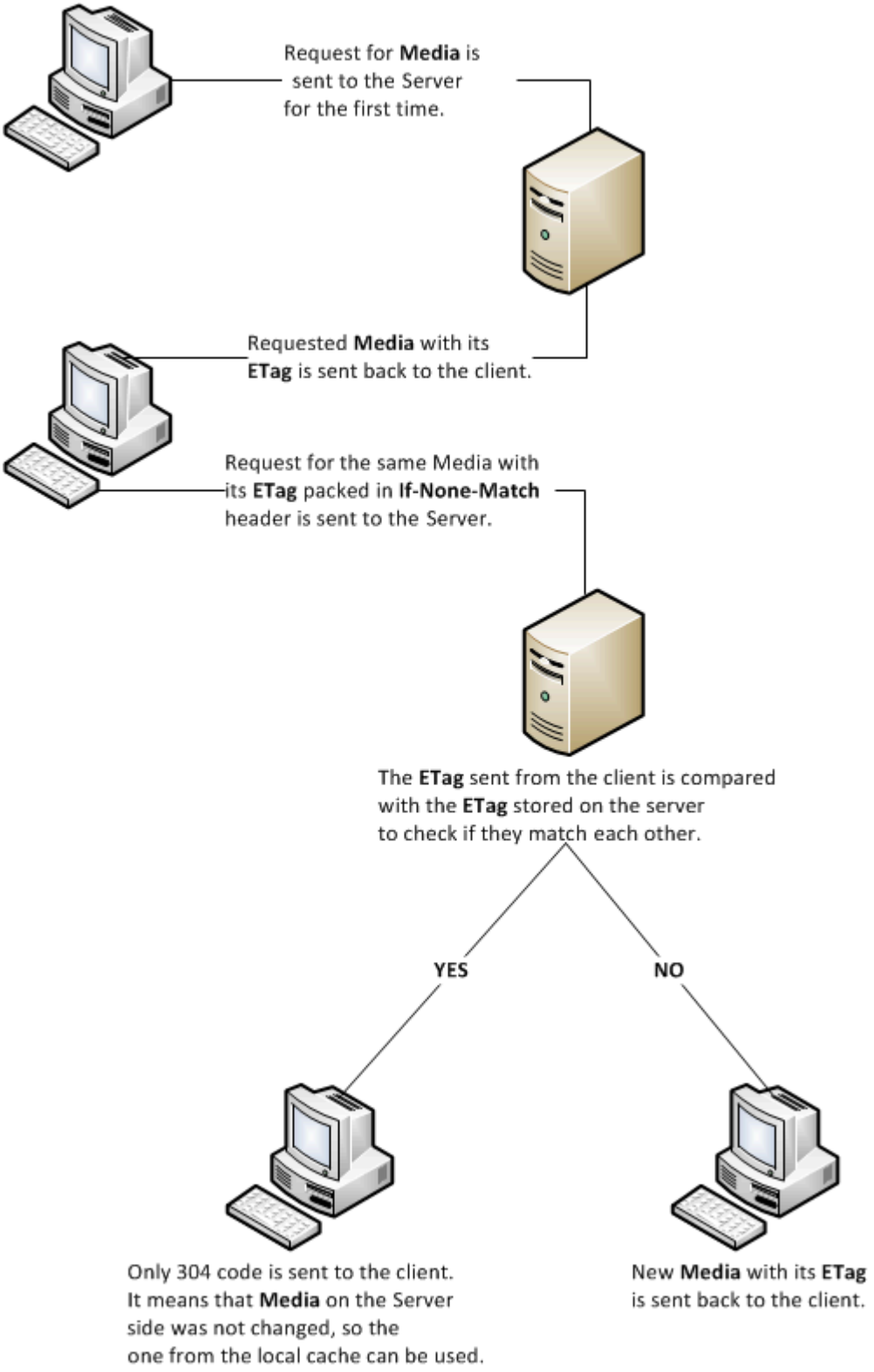
```
allow.custom.mediaweb.webroot=true
```

Using ETag for Web Cache Validation

SAP Commerce uses HTTP ETags in web cache validation.

Whenever a media file is changed, the old file is removed and the new one is saved. Therefore, a new **`dataPK`** attribute is generated for the media. This solution allows the use of a **`dataPK`** attribute as a unique ETag value that is used for web cache validation.

See how web cache validation works in SAP Commerce:



Customer Response Headers for Media Filters

Set custom response headers to improve security of uploaded media.

To set customer response headers for the media filter, navigate to the `local.properties` and use the `media.set.header.<header_name>=<value>` property, for example:

```
media.set.header.superheader=supervalue
```

Use the `media.set.header.<header_name>=disable` property to disable any preconfigured values, for example:

```
media.set.header.Content-Security-Policy=disable
```

The filters respect the values with the following priority, listed from the most to the least important:

1. Blocked properties' values or properties with the `disable` value
2. Properties' values
3. Hardcoded preconfigured values

To set headers based on the media's mime type or folder, use the following properties:

```
media.set.mime.header.<mime>.<headername>=<headervalue>
```

```
media.set.folder.header.<folder>.<headername>=<headervalue>
```

Both properties can override generic headers that are set through the `media.set.header.<headername>=<headervalue>` property. If the mime and folder configuration interfere with each other, the property based on the mime type is considered. In the following example, the headers have the following values:

- `header1` has the `value1` value,
- `header2` has the `value4` value,
- `content-security-policy` and `header3` have no values as they're disabled.

```
media.set.folder.header.root.header1=value1
media.set.folder.header.root.header2=value2
media.set.folder.header.root.header3=value3

media.set.mime.header.text/plain.header2=value4
media.set.mime.header.text/plain.header3=disable
media.set.mime.header.text/plain.content-security-policy=disable
```

Extension Library Management Using Maven

To keep the source repository footprint small and make library updates simple and reliable, each Platform extension is able to use Maven to manage them.

This affects any kind of library required by an extension regardless of it being a 3rd party one or from SAP Commerce. As soon as you find yourself committing a JAR file to your extension, you're actually doing it wrong. Instead, libraries created by SAP Commerce are to be released properly to our Artifactory and fetched the way we describe here.

Ideally, we can establish a 'no JAR' policy for all our source code repositories.

Switching On

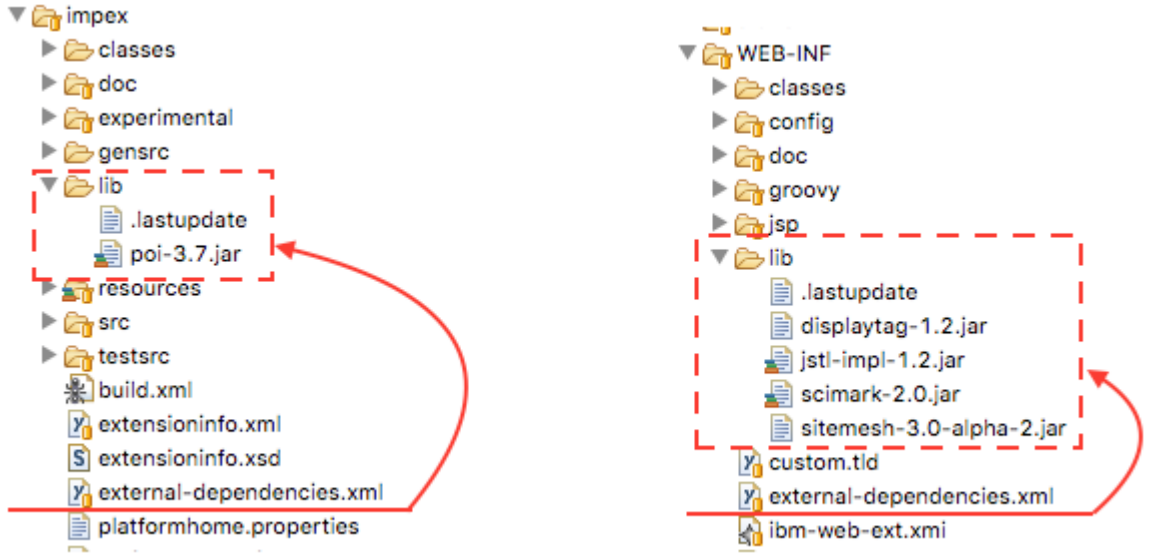
To make your extension use Maven, you need to adjust your extension `info.xml` file and add `usemaven="true"`:

```
<extensioninfo>
  <extension name="foobar" usemaven="true" .. >

..
  </extension>
</extensioninfo>
```

`external-dependencies.xml`

To keep things simple, we actually use the Maven `pom.xml` format to declare dependencies, but for technical reasons the file is actually called `external-dependencies.xml`.

Due to extensions having multiple locations where libraries can be put, there are also multiple `external-dependencies.xml` files to be added.



Managing Global Libraries

The most common place for an extension to ship libraries is `<extension folder>/lib`. At runtime, these libraries will be added to the global class path of Platform (which is why we need to align with other extensions shipping the same).

To use Maven for managing `/lib`, you add an `external-dependencies.xml` file to your extension `root` folder:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-inst
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0
   <modelVersion>4.0.0</modelVersion>

   <groupId>de.hybris.platform</groupId>

   <artifactId>acceleratorservices</artifactId>

   <version>5.0.0.0-SNAPSHOT</version>

   <packaging>jar</packaging>

   <dependencies>


      <dependency>
         <groupId>commons-net</groupId>
         <artifactId>commons-net</artifactId>
         <version>3.0.1</version>
      </dependency>
      <dependency>
         <groupId>org.projectreactor</groupId>
         <artifactId>reactor-core</artifactId>
         <version>1.1.5.RELEASE</version>
      </dependency>
   </dependencies>
</project>
```

With that in place, the ant build process automatically downloads the specified dependencies and place them in `/lib`. To avoid duplicate downloads, a marker file `.lastupdate` is also placed inside `/lib`.

> ⚠ **Caution**
>
> The build process doesn't download transitive dependencies. This means that all required dependencies must be explicitly defined.

Managing Web Module Libraries

In addition to libraries for the global class path, an extension may also require libraries for its web module. These are located in `<extension folder>/web/webroot/WEB-INF/lib`. At runtime, they are loaded into the web application class loader only (which is why you don't need to align with other extensions).

To have them managed by Maven, add another `external-dependendencies.xml` file into `<extension folder>/web/webroot/WEB-INF`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-inst
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xs
   <modelVersion>4.0.0</modelVersion>
   <groupId>de.hybris.platform.hac</groupId>
   <artifactId>hac.web</artifactId>
   <version>5.0.0.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   <dependencies>
      <dependency>
        <groupId>displaytag</groupId>
        <artifactId>displaytag</artifactId>
        <version>1.2</version>
      </dependency>
      <dependency>
         <groupId>org.glassfish.web</groupId>
         <artifactId>jstl-impl</artifactId>
         <version>1.2</version>
      </dependency>
      <dependency>
         <groupId>org.sitemesh</groupId>
         <artifactId>sitemesh</artifactId>
         <version>3.0-alpha-2</version>
      </dependency>
      <dependency>
         <groupId>gov.nist.math</groupId>
         <artifactId>scimark</artifactId>
         <version>2.0</version>
      </dependency>
   </dependencies>
</project>
```

Again, the build process will download the libraries and put them in the `WEB-INF/lib` folder together with the `.lastupdate` marker file. Also, we do not download transitive dependencies here.

Releasing Extensions with Libraries

The ant-based release process automatically includes the library binaries in the assembled artifact so that you are able to use the released extension without Maven. For that reason the release process is also changing `extensioninfo.xml` by `useMaven="true"` back to `useMaven="false"`.

This applies to all types of release: binary, source and development.

HTTP Session Failover

Session failover prevents losing session data by backing up session content in the database. This mechanism provides high availability in a clustered environment.

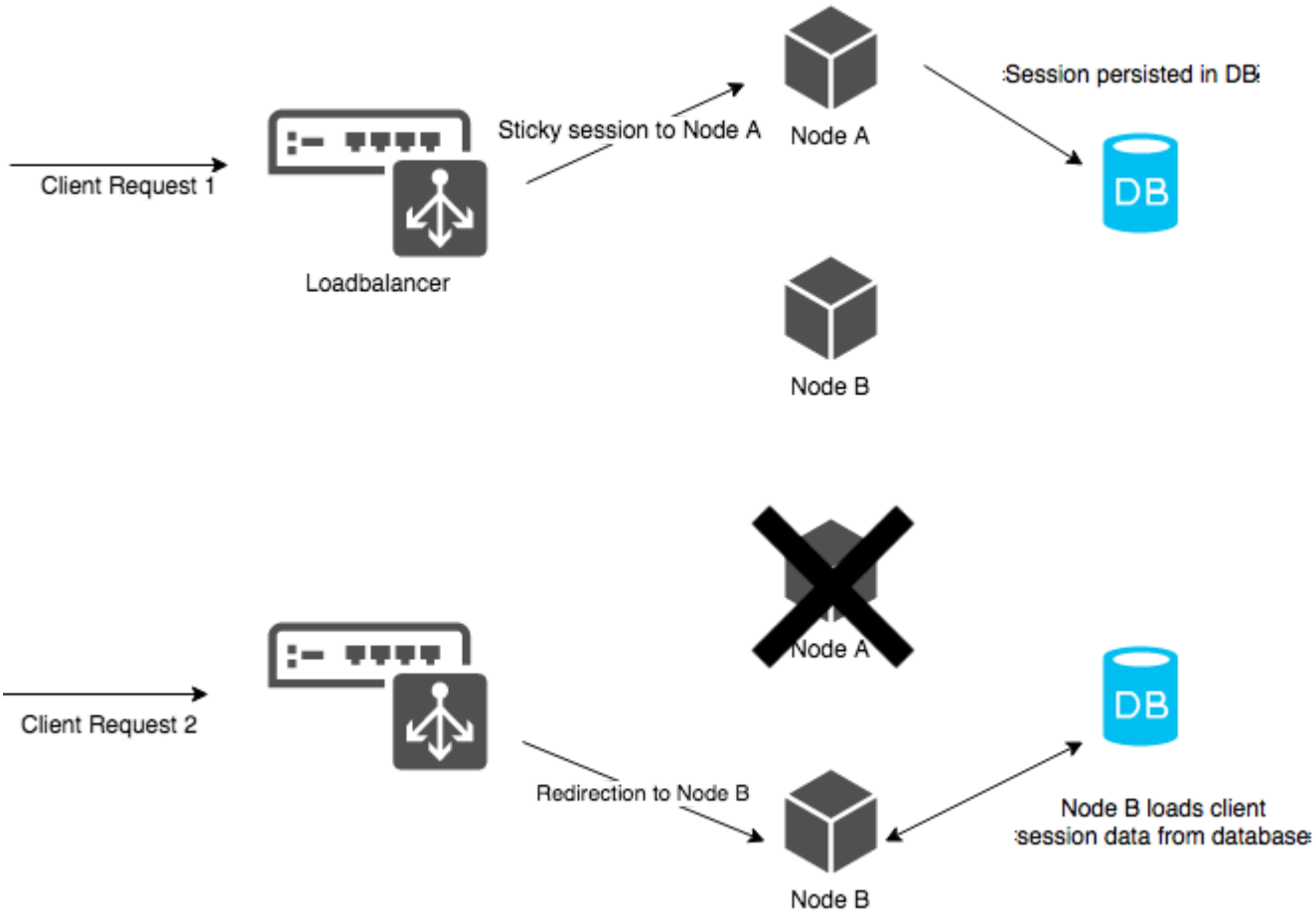
Loss of session data can be a problem. Imagine a customer adding items to a shopping cart and the server crashes before the customer manages to check out. The session has been lost, and the cart is empty. The customer has to add the items to the cart again.

If session failover is in place, cluster nodes can restore sessions persisted in the database. After a node shuts down, the remaining nodes can restore its sessions from the database. They continue to serve requests of these sessions without interruption or data loss. From the client point of view the same session state is maintained by the server. The customer shouldn't notice any problems either.

Session Failover Support

The SAP Commerce session failover implementation is based on the Spring Session project and is application-server agnostic. It uses the SAP Commerce persistence to store session data as items in the relational database.

SAP Commerce session failover is designed with sticky-session load balancing in mind. A load balancer with configured sticky sessions assigns clients to specific cluster nodes. It always forwards requests from the same client to the same server. The server holds the client session cached in memory. A session is a single up-to-date object and there is no need to read it from the database. When the node stops, the load balancer redirects the client requests to another server. That server doesn't have the respective session object in the cache, so it loads it from the database. Lack of session invalidation makes this approach unsuitable for round-robin session load balancing.



SAP Commerce session failover operates in two modes. In **synchronous** mode, a session is saved to the database after each request has been processed before sending a response. In **asynchronous** mode, a session is saved by a separate thread with a configurable time interval for the save operation. The session is added to the to-be-saved queue before sending an HTTP response, but the save action is done later.

Note

HTTP session failover is not supported by Backoffice. For more information, see Backoffice Framework - Technical Overview.

Implementation Details and Customization

`HybrisSpringSessionFilter` is the main class of the session failover feature. Set it as the first element of the `PlatformFilterChain` so it can wrap session-related objects before session is created or used. `HybrisSpringSessionFilter` uses `SessionRepositoryFactory` to create `SessionRepository` for each extension that has session failover enabled. Each extension needs its own `SessionRepository` so that a proper class loader is set for object deserialization. You can replace `SessionRepositoryFactory` with a custom implementation that provides different session storage strategies.

Session Cache Region

All sessions are stored in the separate `sessionCacheRegion` cache region that is defined by the `defaultSessionsCacheRegion` Spring bean.

Session Invalidation

Regular HttpSessions provided by Tomcat are proactively monitored for invalidation (`StandardManager.processExpires`). As soon as a session is a candidate for invalidation, the session is destroyed, and a `sessionDestroyed` event is triggered. Such mechanism is not provided with the OOTB Spring session implementation. Consequently `StoredHttpSession` items have no end of life and remain in the database. You can remedy this by setting up a cronjob deleting the `StoredHttpSession` items where the current time - modified time > 3600 seconds.

Configuring Session Failover

Session failover provides properties you can set up to precisely meet the requirements of your clustered environment.

All the default property values are set in the `platform/project.properties` file in the **Session failover settings** section. You can customize or overwrite them by using standard mechanisms, for example, through `local.properties`.

Global Configuration Properties

The **global** session failover properties include:

Property	Default Value	Values	Description
`spring.session.enabled`	false	true/false	Enables spring session support.
`spring.session.save.async.interval`	5000	int	Interval between session persistence attempts (in milliseconds)
`spring.session.save.async.queue.size`	10000	int	Size of a queue of sessions to persist
`spring.session.save.async.max.items`	2000	int	Max number of session items being saved in one

Property	Default Value	Values	Description
			attempt
`session.serialization.check.response.error`	false	true/false	Returns the 500 error code if a session is not serializable.

Extension Specific Properties

You can specify these properties for each extension individually:

Property	Default Value	Values	Description
`spring.session.[extension_name].save`	no default value is set	sync/async	Selects the session failover operation mode for a target extension. If the property isn't set, then session failover isn't enabled for this extension.
`spring.session.[extension_name].cookie.name`	JSESSIONID	string	Session cookie name
`spring.session.[extension_name].cookie.path`	*<context path>*	string	Session cookie path
`spring.session.[extension_name].secure`	not set	true/false/no value	If not set otherwise, the cookie secure attribute is set to `true` if the request is secure. As the request is by rule always secure, the behavior is effectively always set to `true`.
`spring.session.[extension_name].cookie.httponly`	not set	true/false/no value	If not set otherwise, the cookie httpOnly attribute is set to `true` if the request is secure. As the request is by rule always secure, the behavior is effectively always set to `true`.

⚠ Caution

Set `spring.session.[extension_name].save` for each extension using session failover. Simply setting `spring.session.enabled` to `true` isn't sufficient.

Enabling Session Failover for an Extension

Enabling session failover boils down to setting up a few properties and adding one filter to the filter chain.

Context

An HTTP session of a web extension has to be serializable to work with session failover. For more information, see Checking If a Session Is Serializable.

Procedure

1. Set the `spring.session.enabled` property to `true`.

   ```
   spring.session.enabled=true
   ```

   This enables the `HybrisSpringSessionFilter` global filter. If the property is set to `false`, no session failover can be performed no matter what settings you set for specific web extensions.

2. Select the sync or async session failover operation mode for the target extension.

   ```
   spring.session.<extension_name>.save=sync
   spring.session.<extension_name>.save=async
   ```

   The async configuration is faster but choosing it can result in session data loss if a node fails before the pending session is saved to the database.

3. Configure the session cookie name and path. We recommend configuring the name and path of the cookie to be the same as the `JSESSIONID` cookie, which is generated without the session failover feature enabled. You can, however, set a different cookie ID. See the examples:

   ```
   //Configuration with JSESSIONID cookie name:
   spring.session.<extension_name>.cookie.name=JSESSIONID
   spring.session.<extension_name>.cookie.path=/<extension_webApp_contextPath>

   //Configuration with a different cookie:
   spring.session.<extension_name>.cookie.name=different_cookie_ID
   spring.session.<extension_name>.cookie.path=/<extension_webApp_contextPath>
   ```

4. Make sure `hybrisSpringSessionFilter` is first in the filter chain bean for the target web extension. You can find the filter chain bean inside a *<your_extension_name>*`-web-app-config.xml` file located in a *<your_extension_name>*`/web/webroot/WEB-INF/config/` directory. This code snippet shows how to modify the bean for an example extension called `sessionfailovertest` so that the bean includes `hybrisSpringSessionFilter`:

   ```
   <bean id="sessionfailovertestPlatformFilterChain" class="de.hybris.platform.servicelayer.web.
      <constructor-arg>
         <list>
            <ref bean="hybrisSpringSessionFilter"/>
            <ref bean="log4jFilter"/>
            <ref bean="sessionFilter"/>
            <ref bean="sessionfailovertestMediaFilter"/>
         </list>
      </constructor-arg>
   </bean>
   ```

   Place the filter in the filter chain before a session gets created or accessed by the application. It wraps `HttpRequest` and `HttpResponse`. Otherwise two conflicting session cookies could be created - one from the application server and another by the session failover feature.

Browsing Persisted Sessions

A list of persisted sessions is available in Backoffice.

Procedure

1. In the Explorer tree, click System > Types.
2. Search for the `StoredHttpSession` type.
3. Click the type to open the editor area.
4. Use the Search by type action.

Results

You can now see all sessions that are backed up in the database.

Checking If a Session Is Serializable

Session failover requires that sessions be serializable, and SAP Commerce enables you to check whether they are or not.

Due to the extensible nature of SAP Commerce, it is difficult to assure the session is serializable before a project is implemented. Using the serialization checking mechanism during the development phase helps to find common mistakes such as storing non-serializable objects in an http session.

Enabling Session Serialization Checking

To enable checking for session failover, set `session.serialization.check` to `true`:

```
session.serialization.check=true
```

This enables session serialization checking for all web extensions unless `session.serialization.check.extensions` is set.

To enable checking for specific extensions, set names of the extensions for which session checking should be done as a value for the `session.serialization.check.extensions` property:

```
session.serialization.check.extensions=hac,sampleextension
```

Separate extension names with the comma.

> ⚠ **Caution**
> Session serialization checking is a development feature. **Disable it in production environments**.

When a non-serializable object is added to an HTTP session, an error and stacktrace are logged:

```
ERROR [hybrisHTTP10] [SessionFilter] Failed to serialize attribute: jalosession
java.io.NotSerializableException: de.hybris.sessionfailovertest.NotSerializable at
java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184) ...
```

Configuration Properties

Use these properties to configure session serialization checking:

Property	Sample Value	Values	Description
`session.serialization.check`	false	true/false	Enables serialization checking feature o each request.
`session.serialization.check.extensions`	hac,yacceleratorstorefront	comma-separated names of extensions	Specifies a list of extensions for which session serialization checking should be done. If this property is not set while `session.serialization.check=t` then session serialization checking will b performed for all web extensions.

Transparent HTTP Session Failover

Session failover mechanism prevents HTTP session loss after the node this session was bound to goes down. SAP Commerce includes the so-called transparent HTTP session failover mechanism.

HTTP is a stateless protocol, that is, the server is unaware whether any requests are related to one another. A common way to allow some sort of session management is by using session cookies.

However, HTTP sessions are typically bound to one node in a web server cluster. If the node goes down, the HTTP sessions on that node are lost. To avoid HTTP session loss, the web server cluster needs to have some session failover mechanism. A common session failover mechanism is session replication, which makes sure that HTTP sessions are replicated across the entire cluster. One typical technical means of such replication is serialization.

SAP Commerce includes a transparent HTTP session failover mechanism.

- **Transparent** means that the functionality is automatic and the application (in this case, SAP Commerce) doesn't have to handle any of the session managements.
- **HTTP Session Failover** means that the HTTP sessions that are assigned to web requests can be replicated onto every individual web server cluster node. By consequence, even if a web server node goes down, the HTTP sessions can still be recovered and be processed on another web server node.

Note that the transparent HTTP session failover takes place on the web application level only. On the web application level, SAP Commerce uses other modifications for sessions than on the ServiceLayer. Therefore, the transparent HTTP session failover is no replacement for the SAP Commerce Cluster. For example, CronJobs don't run on the web application level and therefore can't use this transparent HTTP session failover.

How Does Transparent HTTP Session Failover Work?

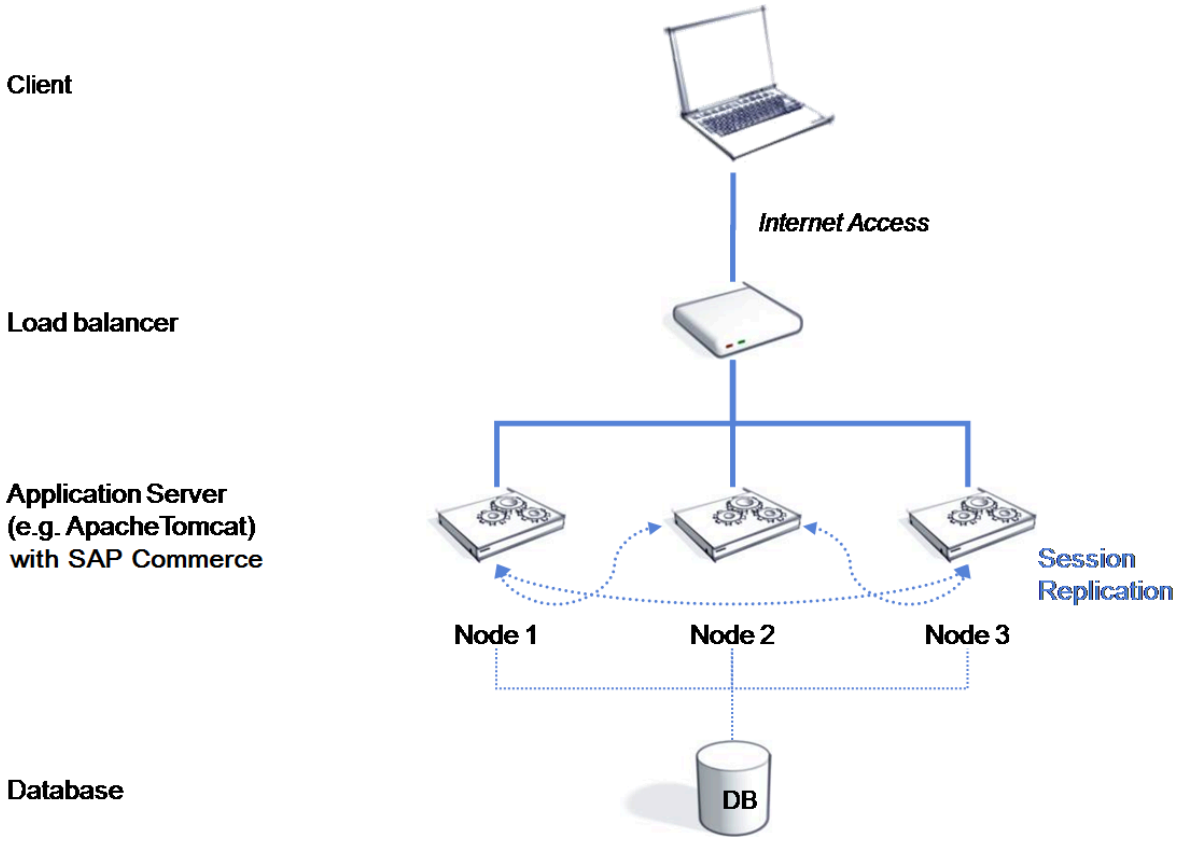


Figure: Typical Productive System Infrastructure, reduced to the components that are relevant for transparent HTTP session failover.

The figure shows an excerpt from a typical production system architecture:

1. The client opens a connection to the system such as `http://www.myshop.com`, thus creating a web request.
2. The load balancer dispatches the web request onto one of the nodes, depending on the load-balancing strategy.
3. The transparent HTTP session failover makes sure that the sessions are replicated across the cluster. Session replication is indicated by dotted blue lines in the figure.

You can set up your system to use either sticky sessions or non-sticky sessions.

- Sticky Sessions

  Here, all web requests of one HTTP session are served from the same cluster node.

  If a request of an HTTP session was first dispatched to **Node 1** such as `http://www1.myshop.com`, all subsequent requests are dispatched to **Node 1**, such as `http://www1.myshop.com`.

- Non-Sticky Sessions

  Here, web requests are dispatched to random nodes across the cluster, depending on the load-balancing strategy.

  If a request of an HTTP session was first dispatched to **Node 1** such as `http://www1.myshop.com`, subsequent requests might be dispatched to any nodes, such as `http://www3.myshop.com`.

Without a session replication mechanism, you need to use sticky sessions: on a non-sticky session system, every individual node that has processed the session once, has an individual copy of the session. Take a session cart, for example: on a non-sticky session system, each time the session is processed by a different node, that node holds a representation of the cart at the time. This might result in 25 different versions of the cart in a 25 node system.

Possible Scenarios of Setup

In SAP Commerce you can use sticky sessions, non-sticky sessions, and transparent HTTP session failover, resulting in three possible scenarios:

Scenario	Description and Recommendation
Sticky sessions without session replication	Consider this scenario if maximum performance is important to you and if you can accept that sessions are lost if the respective cluster node goes down. If session loss is not acceptable to you, consider the approaches with session failover.
Sticky sessions, plus transparent HTTP session failover for session replication	Sticky sessions provide better performance over non-sticky sessions.
Non-sticky sessions, plus transparent HTTP session failover for session replication	Not recommended because sticky sessions perform better than Non-Sticky Sessions: the replication of sessions between nodes takes some time and also costs some performance. Consider using sticky sessions plus transparent HTTP session failover instead.

Setting Up Transparent HTTP Session Failover

These steps need to be completed only once. As soon as the infrastructure for load balancing and transparent HTTP session failover is set up, you can use the infrastructure for any number of extensions.

1. Choose and configure the technology used for session replication. You can use either the Apache Tomcat session replication mechanism or Oracle Coherence.

   The Apache Tomcat is available for free and less complex to set up. Oracle Coherence is better in performance than the Apache Tomcat, but must be licensed and is more complex to set up.

   - The Apache Tomcat default session replication mechanism

     See Using Apache Tomcat Session Replication.

   - Oracle Coherence

     See Using Oracle Coherence with an Extension.

2. Choose and configure a load balancer.

   You can use any load balancer. Typically, in production environments, a hardware load balancer is used. However, for testing in project use, for example, a hardware load balancer might be too expensive or not be available. Instead of a hardware load balancer, you can use software load balancers, such as:

   - Apache http Server.

     For details on setup, refer to Using Apache HTTP Server as Load Balancer.

   - Inlab Balancer

     For details on setup, refer to Load Balancer's Example Configuration - Inlab Balance.

Related Information

Jalo Session
Platform Filters
About Extensions
HTTP Session Failover

Using Apache Tomcat Session Replication

SAP Commerce supports session replication in SAP Commerce Server.

Context

To enable HTTP session replication (transparent HTTP session failover), modify the configuration of the pre-bundled Apache Tomcat. For additional detail, see Tomcat 7.0 Cluster Howto .

For the Apache Tomcat to use session replication, you need to make some modifications. These setup steps are required once per Apache Tomcat instance. In the case of SAP Commerce Server, this means that you need to make these modifications on every SAP Commerce Server instance.

⚠ Caution

Not all extensions support session replication.

Procedure

1. Put the following Tomcat session replication configuration inside the `Engine` tag of the *<HYBRIS_CONFIG_DIR>*`/tomcat/conf/server.xml` file.

```
<Engine name="Catalina" defaultHost="localhost">
        ...


<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
       channelSendOptions="8">

   <Manager className="org.apache.catalina.ha.session.DeltaManager"
          expireSessionsOnShutdown="false"
          notifyListenersOnReplication="true"/>

   <Channel className="org.apache.catalina.tribes.group.GroupChannel">
      <Membership className="org.apache.catalina.tribes.membership.McastService"
                address="228.0.0.4"
                port="45564"
                frequency="500"
                dropTime="3000"/>
      <Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"
              address="auto"
              port="4000"
              autoBind="100"
              selectorTimeout="5000"
              maxThreads="6"/>

      <Sender className="org.apache.catalina.tribes.transport.ReplicationTransmitter">
         <Transport className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"
      </Sender>
      <Interceptor className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetecto
      <Interceptor className="org.apache.catalina.tribes.group.interceptors.MessageDispatchIn
   </Channel>

   <Valve className="org.apache.catalina.ha.tcp.ReplicationValve" filter=""/>
   <Valve className="org.apache.catalina.ha.session.JvmRouteBinderValve"/>

   <ClusterListener className="org.apache.catalina.ha.session.ClusterSessionListener"/>
</Cluster>
```

   Each invocation of `ant all` replaces the *<HYBRIS_BIN_DIR>*`/platform/tomcat/conf/server.xml` file (used by Tomcat) with new one generated from this template so make any permanent changes in the *<HYBRIS_CONFIG_DIR>*`/tomcat/conf/server.xml` file

2. Make sure that the IP address and the port values in the `Membership` element in the `server.xml` file are identical for all Apache Tomcat instances. It has to be so because an Apache Tomcat cluster is identified by the combination of an IP address and a port, such as:

```
<Membership
   className="org.apache.catalina.tribes.membership.McastService"
   address="228.0.0.4"
   port="45564" />
```

Note

Under Microsoft Windows, Tomcat clustering doesn't work when cluster nodes are on the same physical machine.

3. Invoke **ant all** and make sure that the *<HYBRIS_BIN_DIR>*/platform/tomcat/conf/server.xml file contains cluster configuration from step 1.
4. Configure Session Manager for all the contexts (extensions) that should have session replication enabled. To do this, replace <Manager pathname="" /> with <Manager className="org.apache.catalina.ha.session.DeltaManager" expireSessionsOnShutdown="false" notifyListenersOnReplication="true" /> for chosen extensions in the *<HYBRIS_BIN_DIR>*/platform/tomcat/conf/server.xml file.

   For example, if you wish to enable session replication for storefront (yacceleratorstorefront extension) replace:

```
<!-- 'yacceleratorstorefront' extension's context for tenant 'master' -->
<Context path="/yacceleratorstorefront" docBase="/Users/i310983/session-replication-test/sapc
   <Manager pathname="" />
   <Loader platformHome="/Users/i310983/projects/dev-platform-core/bin/platform" className="d
</Context>
```

   with

```
<Context path="/yacceleratorstorefront" docBase="/Users/i310983/session-replication-test/sapc
   <Manager className="org.apache.catalina.ha.session.DeltaManager" expireSessionsOnShutdown=
   <Loader platformHome="/Users/i310983/projects/dev-platform-core/bin/platform" className="d
</Context>
```

   If this element is not commented out, an exception will occur during start up of the Apache Tomcat which is part of SAP Commerce Server, and the clustering will not work:

```
WARNING: Manager [ org.apache.catalina.session.StandardManager@ed5d9d] does not implement Clu
```

Note

This file can be overwritten by build process, so make sure that you have correct Manager set before starting a cluster node!

5. If you configured everything properly, check whether you can see org.apache.catalina.tribes logs after startup, for example:

```
Sep 15, 2015 4:36:11 PM org.apache.catalina.ha.session.DeltaManager startInternal
INFO: Register manager localhost#/yacceleratorstorefront to cluster element Engine with name
Sep 15, 2015 4:36:11 PM org.apache.catalina.ha.session.DeltaManager startInternal
INFO: Starting clustering manager at localhost#/yacceleratorstorefront
Sep 15, 2015 4:36:11 PM org.apache.catalina.ha.session.DeltaManager getAllClusterSessions
INFO: Manager [localhost#/yacceleratorstorefront], requesting session state from org.apache.c
Sep 15, 2015 4:36:11 PM org.apache.catalina.ha.session.DeltaManager waitForSendAllSessions
INFO: Manager [localhost#/yacceleratorstorefront]; session state send at 9/15/15 4:36 PM rece
```

6. Test whether session replication works:
   a. Log in to yacceleratostorefront on the first node - you should have two JSESSIONID cookies created.
   b. Add some products to the cart.
   c. Log in with a different browser that supports cookie manipulation to another node - you should also have two JSESSIONID cookies created.
   d. Edit these two JESESSIONID cookies content so that they match the JSESSIONID of the first node.
   e. Refresh the second browser - you should have the same session as on the other node and browser - all the changes you do in one browser will be visible in the other - this is a proof that session is replicated.

> **Note**
>
> When testing, pay attention to `CartRestorationFilter`. It stores additional cookie with a unique id that matches cart id that is stored in database. When SAP Commerce detects that this cookie exists but there's no session if fetches cart data from database. This can give an impression that session replication is working when in fact it is not.

Related Information

Running SAP Commerce
https://tomcat.apache.org/tomcat-7.0-doc/cluster-howto.html

Extensions Using Session Replication

Follow these steps to use the transparent HTTP session failover with your custom extension.

Context

These steps are not needed for extensions written by SAP Commerce. If your custom extension doesn't contain a web module, you don't need to follow these steps either.

To configure an extension to support session failover:

Procedure

1. Add the `<distributable />` directive to your extension's `web/webroot/WEB-INF/web.xml` file:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<web-app id="mywebapp">

        <distributable/>
```

2. Set the following property in your `local.properties` file:

```
session.replication.support=true;
```

   The property enables the session replication support globally for all extensions. The `SessionFilter` that needs to be included in your Platform filter chain handles the session failover.

3. **Optional:** If you use the deprecated `HybrisInitFilter`, modify the `<filter>` section:
   - Make sure that `HybrisInitFilter` is enabled.
   - Add the `touch.httpSession` init filter parameter inside `web/webroot/WEB-INF/web.xml`, and set the value to `true`:

```
<filter>
        <filter-name>InitFilter</filter-name>
        <filter-class>de.hybris.platform.util.HybrisInitFilter</filter-class>

        <!-- touch httpSession every time jaloSession changes (for session failover) (default
        <init-param>
                <param-name>touch.httpSession</param-name>
                <param-value>true</param-value>
        </init-param>
[...]
</filter>
```

Setting `touch.httpSession` to `true` allows session replication through serialization of the JaloSession.

ℹ Note

Find Out More

The HttpSession object is modified by SAP Commerce by adding an attribute, `jalosession`. However, session replication systems cannot detect changes in the JaloSession object assigned to the HttpSession. Therefore, changes in the JaloSession are not replicated onto other web server cluster nodes by factory default.

The `touch.httpSession` parameter affects the `RootRequestFilter` and, by consequence, the `HybrisInitFilter`. If set to `true`, the parameter causes the filters to "touch" the `HttpSession` object by re-setting the same JaloSession to the `HttpSession` object. This re-setting of the `jalosession` attribute causes the `HttpSession` object to appear modified to session replication mechanisms, and the `HttpSession` object will be replicated. By consequence, changes in the JaloSession are also replicated. This allows SAP Commerce to use common session replication systems.

`web/webroot/WEB-INF/web.xml`

```
<?xml version="1.0" encoding="iso-8859-1"?>

<web-app id="springmvc" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/

        <distributable/>
        <display-name>myapp</display-name>

        <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
[...]
        <filter>
                <filter-name>InitFilter</filter-name>
                <filter-class>de.hybris.platform.util.HybrisInitFilter</filter-class>

                <!-- touch httpSession every time jaloSession changes (for session failover)
                <init-param>
                        <param-name>touch.httpSession</param-name>
                        <param-value>true</param-value>
                </init-param>
[...]
        </filter>
```

Related Information

Platform Filters

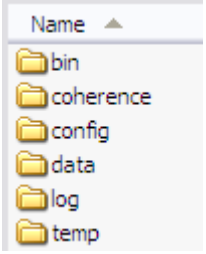
Using Oracle Coherence with an Extension

To use session replication with Oracle Coherence for web applications, all JSP pages and servlets in that web application must be wrapped in Oracle Coherence classes. This process is automatically done by Coherence*Web Installer.

Setting Up Oracle Coherence for Use with SAP Commerce

Oracle Coherence is available as a ZIP file for download:

1. Download Oracle Coherence: http://www.oracle.com/technetwork/middleware/coherence/downloads/index.html
2. Extract the ZIP file.
3. Place the `coherence` directory on the same level as the *<HYBRIS_BIN_DIR>* directory, such as:



Test Scenarios

Follow the steps to execute an example test scenario.

Prerequisites

The setup consists of two SAP Commerce Cluster nodes (using different ports) and InLab Balance as Loadblanacer running on the same machine (Mac OS-X).

Procedure

1. Make sure that only one node is running.
2. Enter the store address: `http://localhost:8080/springmvcstore/`.

   8080 is the configured loadbalancer port here.
3. Log in as user **demo** and use the password you defined for this account.
4. Fill the cart.
5. Start the second node.
6. **Optional:** Shut down the node that was running first, when the second node is up and running.
7. Verify the cart to make sure that cart entries are the same.
8. **Optional:** Verify whether Coherence sends the session to the other node in a serialized way, such as:

```
INFO  [Coherence] 2009-10-30 09:53:24.645/580.052 Oracle Coherence EE 3.4.2/411
<Info> (thread=DistributedCache:DistributedSessions, member=3): Restored from backup 85 parti
```

Using Apache HTTP Server as Load Balancer

The Apache HTTP Server is a commonly used, open source http server and is available for many common operating systems. The Apache HTTP Server can be extended in functionality by integrating modules. By using certain modules, the Apache HTTP Server can be used as a software load balancer, for example.

The Apache HTTP Server can by extended to integrate additional functionality by using modules (also called "mods"). The Transparent HTTP Session Failover functionality relies on these mods:

1. **mod_proxy_balancer**

   Required. Provides load balancing functionality. For details, see the Apache website: http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html .
2. **mod_headers**

   Optional. Required for sticky sessions. For details, see the Apache website: http://httpd.apache.org/docs/2.2/mod/mod_headers.html .

To use such mods, two steps are necessary:

1. The mods must be made available to the Apache HTTP Server.
2. The Apache HTTP Server must be configured to use the mods.

Making Mods Available to the Apache HTTP Server

There are two ways of how the Apache HTTP Server can integrate mods:

- Standalone.

  Modules are compiled independently of the Apache HTTP Server executable. In this case, you can disable unused mods to increase performance.

  This is recommended if you use the Apache HTTP Server both as a web server and as a load balancer.

  To enable standalone mods, you need to modify the **`${APACHE_INSTALL}/conf/httpd.conf`** file and uncomment the lines that reference mods. For example, from

`${APACHE_INSTALL}/conf/httpd.conf (mods disabled)`

```
#LoadModule headers_module modules/mod_headers.so
#LoadModule proxy_module modules/mod_proxy.so
#LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
#LoadModule proxy_connect_module modules/mod_proxy_connect.so
#LoadModule proxy_http_module modules/mod_proxy_http.so
```

to **`${APACHE_INSTALL}/conf/httpd.conf (mods enabled)`**

```
LoadModule headers_module modules/mod_headers.so
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_connect_module modules/mod_proxy_connect.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

- Built-In

  Modules are compiled into the Apache HTTP Server executable. In this case all modules are compiled and loaded every time the Apache HTTP Server is started.

  This is recommended if you use the Apache HTTP Server only as a load balancer and not as a web server as well.

Configuring Apache HTTP Server

Once you have set up the Apache HTTP Server as a load balancer, you need to configure the load balancing:

- Sample configuration with sticky sessions: Example Configuration with Sticky Sessions
- Sample configuration without sticky sessions: Example Configuration Without Sticky Sessions

Example Configuration with Sticky Sessions

This document provides a load balancer example configuration for Inlab Balance.

Let's imagine that we want to set up our environment in a following manner:

- We want application to be accessed by 192.168.0.1 IP address on port 80
- We want application to run on two cluster nodes:

- First on port 8001 and with 192.168.0.2 IP address
- Second on port 9001 and with 192.168.0.3 IP address

Apache HTTP Server's Configuration

Apache HTTP Server can be configured using `httpd.conf` file.

Append the following code sample to the existing HTTP Server's configuration file:

`${APACHE_INSTALL}/conf/httpd.conf`

```
Listen 80
Header add Set-Cookie: "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; path=/" env=BALANCER_ROUTE_CHANGED
<Proxy balancer://mycluster>
BalancerMember http://192.168.0.2:8001 route=node1
BalancerMember http://192.168.0.3:9001 route=node2
</Proxy>

ProxyPass / balancer://mycluster/ stickysession=ROUTEID
ProxyPassReverse / balancer://mycluster/

<Location /balancer-manager>
SetHandler balancer-manager

Order Deny,Allow
Deny from all
Allow from all
</Location>
```

Thanks to this line:

`${APACHE_INSTALL}/conf/httpd.conf`

```
ProxyPass /myapp balancer://mycluster/myapp stickysession=ROUTEID
```

it's completely transparent to the user, which cluster's member he is connected to (address bar always shows url of load balancer server, not cluster's members url).

Load balancer port is set to 80 and application is visible as `http://192.168.0.1:80/myapp`.

Example Configuration Without Sticky Sessions

This document provides a load balancer example configuration without sticky sessions.

Let's imagine that we want to set up our environment in the following manner:

- We want application to be accessed by 192.168.0.1 IP address on port 80.
- We want application to run on two cluster nodes:
  - First on port 8001 and with 192.168.0.2 IP address
  - Second on port 9001 and with 192.168.0.3 IP address

Apache HTTP Server's Configuration

Apache HTTP Server can be configured using `httpd.conf` file.

Append the following code sample to the existing HTTP Server's configuration file:

`${APACHE_INSTALL}/conf/httpd.conf`

```
Listen 80
<Proxy balancer://mycluster>
BalancerMember http://192.168.0.2:8001/ loadfactor=2
BalancerMember http://192.168.0.3:9001/ loadfactor=1
</Proxy>

ProxyPass /myapp balancer://mycluster/myapp
ProxyPassReverse /myapp balancer://mycluster/myapp

<Location /balancer-manager>
SetHandler balancer-manager

Order Deny,Allow
Deny from all
Allow from all
</Location>
```

In provided example ratio is 2:1 and sticky sessions are off, so every third **request** goes to 192.168.0.3:9001.

Thanks to this line:

`${APACHE_INSTALL}/conf/httpd.conf`

```
ProxyPass /myapp balancer://mycluster/myapp
```

it is completely transparent to user which cluster's member he is connected to (address bar always shows url of load balancer server, not cluster's members url).

Load balancer port is set to 80 and application is visible as `http://192.168.0.1:80/myapp`.

Load Balancer's Example Configuration - Inlab Balance

This document includes a load balancer example configuration for Inlab Balance.

Let's imagine that we want to setup our environment in a following manner:

- We want application to be accessed by 192.168.0.1 IP address on port 80.
- We want application to run on two cluster nodes:
  - First on port 8001 and with 192.168.0.2 IP address.
  - Second on port 9001 and with 192.168.0.3 IP address.

Inlab Balance Configuration

Balance can be configured by passing command line parameters.

```
balance 80 -H -b 192.168.0.1 192.168.0.2:8001 192.168.0.3:9001 %
```

Parameter	Description
80	port used to access Inlab Balance
H	failover to next host even if sticky sessions are on
b 192.168.0.1	IP address used to access Inlab Balance

Parameter	Description
192.168.0.2:8001	IP address and port of the first node
192.168.0.3:9001	IP address and port of the second node
%	turns on sticky sessions mode

For more details, consult Inlab Balance documentation on the Inlab website.

Mac OS-X Related Modifications

Makefile

```
# $Id: Makefile,v 1.45 2008/04/08 17:39:08 tommy Exp $

#CFLAGS=-g -I.
CFLAGS=-O2 -Wall -Wstrict-prototypes -Wuninitialized

# uncomment for any OS other than Cygwin
BALANCE=balance
ROOT=root
INSTALL=install
BINDIR=/sbin
MANDIR=/opt/local/share/man/man1

# uncomment for Solaris
# LIBRARIES=-lsocket -lnsl
# INSTALL=/usr/ucb/install
# BINDIR=/usr/local/libexec

# uncomment for Cygwin
# LIBRARIES=-L/usr/local/lib -lcygipc
# BALANCE=balance.exe
# ROOT=Administrators

CC=gcc
RELEASE=3.42

all: balance

balance: balance.o butils.o
        $(CC) $(CFLAGS) -I. -o balance balance.o butils.o $(LIBRARIES)

balance.o: balance.c balance.h
        $(CC) $(CFLAGS) -I. -c balance.c

butils.o: butils.c balance.h
        $(CC) $(CFLAGS) -I. -c butils.c

balance.pdf: balance.ps
        ps2pdf balance.ps balance.pdf

balance.ps: balance.1
        troff -Tpost -man balance.1 | /usr/lib/lp/postscript/dpost > balance.ps
        # groff -f H -man balance.1 > balance.ps

ci:
        ci -l *.c *.h Makefile balance.1 README balance.spec

clean:
        rm -f $(BALANCE) *.o balance.ps balance.pdf

install:
        $(INSTALL) -o $(ROOT) -g admin -m 755  $(BALANCE) \
                $(DESTDIR)$(BINDIR)/$(BALANCE)
        $(INSTALL) -o $(ROOT) -g admin -m 755  balance.1 \
                $(DESTDIR)$(MANDIR)
        mkdir -p $(DESTDIR)/var/run/balance
        chmod 1777 $(DESTDIR)/var/run/balance
```

```
release: balance.pdf
        rm -rf ./releases/balance-$(RELEASE)
        mkdir ./releases/balance-$(RELEASE)
        cp balance.1 balance.pdf balance.c balance.h butils.c COPYING Makefile README ./releases/ba
        cp balance.spec ./releases/balance-$(RELEASE)/balance.spec
        cd releases; tar -cvf balance-$(RELEASE).tar ./balance-$(RELEASE)
        cd releases; gzip balance-$(RELEASE).tar

rpm:    ever
        cp releases/balance-$(RELEASE).tar.gz /usr/src/redhat/SOURCES/
        rpmbuild -ba balance.spec
        cp /usr/src/redhat/SRPMS/balance-$(RELEASE)-1.src.rpm ./releases
        cp /usr/src/redhat/RPMS/i386/balance-$(RELEASE)-1.i386.rpm ./releases

ever:
```

Sample: Connections to the local port 8080 are forwarded alternating to 192.168.146.7:9001 and 192.168.146.7:9004.

```
sudo /sbin/balance -f 8080 192.168.146.7:9001 % 192.168.146.7:9004 %
```

ⓘ Note

The parameter **%** allows connecting one client always to the same server, for example balancing http sessions to a single server.

Setting Up Your Extensions to Use Oracle Coherence

Follow the steps to set up your extensions to use Oracle Coherence.

The preparatory steps are required for every extension that you write and in which you wish to use transparent HTTP session failover. For extensions written by SAP, these preparatory steps are not needed. For extensions written by yourself, you need these preparatory steps only if you wish to use the transparent HTTP session failover. If your extension does not contain a web module, you do not need these preparatory steps.

Setting Up the Web Module of Your Extensions

Follow the steps to set up the web module of your extensions.

Prerequisites

To configure an extension to support session failover, modify the extension's `web/webroot/WEB-INF/web.xml` file.

Procedure

1. Add the `<distributable/>` directive, such as:

   `web/webroot/WEB-INF/web.xml`

   ```
   <?xml version="1.0" encoding="iso-8859-1"?>
   <web-app id="mywebapp">
           <distributable/>
   ```

2. Set the following property in your `local.properties` file:

```
session.replication.support=true;
```

This enables the session replication support globally for all extensions. The `SessionFilter` that needs to be included in your Platform. Filters chain handles the session failover. For more details, read hybris Platform Filters.

If you are still using the deprecated `HybrisInitFilter` modify the `<filter>` section by:

- Making sure that the `HybrisInitFilter` is enabled.
- Adding the `touch.httpSession` init filter parameter, and setting the value to **true**.

`web/webroot/WEB-INF/web.xml`

```
<filter>
        <filter-name>InitFilter</filter-name>
        <filter-class>de.hybris.platform.util.HybrisInitFilter</filter-class>

        <!-- touch httpSession every time jaloSession changes (for session failover) (de
        <init-param>
                <param-name>touch.httpSession</param-name>
                <param-value>true</param-value>
        </init-param>
[...]
</filter>
```

Setting this parameter to **true** allows session replication via serialization of the JaloSession.

Note

Find Out More

The HttpSession object is modified by SAP Commerce by adding an attribute `jalosession`. However, session replication systems can't detect changes in the JaloSession object assigned to the HttpSession. Therefore, changes in the JaloSession are not replicated onto other web server cluster nodes by factory default.

The `touch.httpSession` parameter affects the `RootRequestFilter` and, by consequence, the `HybrisInitFilter`. If set to **true**, the parameter causes the filters to "touch" the HttpSession object by resetting the same JaloSession to the HttpSession object. This resetting of the `jalosession` attribute causes the HttpSession object to appear modified to session replication mechanisms, and the HttpSession object will be replicated. By consequence, changes in the JaloSession are also replicated. This allows SAP Commerce to use common session replication systems.

`web/webroot/WEB-INF/web.xml`

```
<?xml version="1.0" encoding="iso-8859-1"?>

<web-app id="springmvc" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.cor

        <distributable/>
        <display-name>myapp</display-name>

        <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
[...]
        <filter>
                <filter-name>InitFilter</filter-name>
                <filter-class>de.hybris.platform.util.HybrisInitFilter</filter-class>

                <!-- touch httpSession every time jaloSession changes (for session failo
                <init-param>
                        <param-name>touch.httpSession</param-name>
                        <param-value>true</param-value>
                </init-param>
```

```
            [...]
                    </filter>
```

Modifying the Build Framework for Your Extension

To make use of Oracle Session Replication, an extension has to be wrapped by Oracle Coherence. You can make SAP Commerce do this wrapping automatically by hooking into the SAP Commerce Build Framework.

This example assumes that the extension which is wrapped by Oracle Coherence is named `extensionName`. Replace `extensionName` with your actual extension's name.

Add the following code to the `buildcallbacks.xml` file in the *<HYBRIS_BIN_DIR>*`/extensionName` directory. This code implements a new Apache Ant build target: `buildCoherence`:

```
<property name="coherence.dir" location="${HYBRIS_BIN_DIR}/../coherence" />
<import file="${coherence.dir}/ant-coherence-web-handling.xml" />

<target name="buildCoherence">
        <!-- create war at temp folder -->

        <!-- MODIFICATION: you have to replace 'extensionName.war' by the real one-->
        <echo>Creating war file at ${HYBRIS_TEMP_DIR}/coherence_war/extensionName.war</echo>

        <!-- MODIFICATION: you have to replace 'extensionName' by the real one-->
        <buildwar destdir="coherence_war" extname="extensionName" />

        <!-- apply Coherence installation -->

        <!-- MODIFICATION: you have to replace 'extensionName.war' by the real one-->
        <echo>Installing Coherence on ${HYBRIS_TEMP_DIR}/coherence_war/extensionName.war
                to ${HYBRIS_TEMP_DIR}/coherence_war/coherence_temp/extensionName.war</echo>

        <!-- MODIFICATION: you have to replace 'extensionName' by the real one-->
        <coherence-install
                dir="${HYBRIS_TEMP_DIR}/coherence_war"
                coherence.dir="${coherence.dir}"
                app.name="extensionName" />

        <!-- delete web folder at target extension -->

        <!-- MODIFICATION: you have to replace 'extensionName' by the real one-->
        <echo>Deleting web folder of target extension extensionName</echo>

        <!-- MODIFICATION: you have to replace 'ext.extensionName.path' by the real one-->
        <delete dir="${ext.extensionName.path}/web/webroot" />

        <!--  and unzip modified war to it -->

        <!-- MODIFICATION: you have to replace 'extensionName' by the real one-->
        <echo>Unzipping Coherence war to web folder of target extension extensionName</echo>

        <!-- MODIFICATION: you have to replace 'extensionName.war' by the real one-->
        <unzip
                dest="${ext.extensionName.path}/web/webroot"
                src="${HYBRIS_TEMP_DIR}/coherence_war/coherence_temp/extensionName.war" />
</target>
```

Building Extensions Using Oracle Coherence

Follow the steps to build extensions using Oracle Coherence.

Procedure

1. Call `ant buildCoherence` in the *<HYBRIS_BIN_DIR>*`/platform` directory to wrap the `extensionName` extension with Oracle Coherence.
2. Wait for the build to complete.

   Calling `ant clean` requires calling `ant buildCoherence`.

   The wrapper process of Oracle Coherence patches the files which are compiled by the SAP Commerce Build Framework. During `ant clean`, the compiled files are removed. By consequence, the wrapper files for Oracle Coherence are removed, too. Therefore, to make sure that the wrapper files are available, you need to call `ant buildCoherence` after every call of `ant clean`.
3. Start SAP Commerce Server
   - Normal operation mode:
     a. Navigate to the *<HYBRIS_BIN_DIR>*`/platform` directory.
     b. To start the SAP Commerce Server:
     - On Windows systems, call the `hybrisserver.bat` file.
     - On Unix systems call the `hybrisserver.sh` file, such as: `./hybrisserver.sh`.
   - Debug operation mode, requiring **develop** configuration template:
     a. Navigate to the *<HYBRIS_BIN_DIR>*`/platform` directory.
     b. To start the SAP Commerce Server:
     - On Windows systems run the `hybrisserver.bat` file with the debug parameter, such as `hybrisserver.bat debug`.
     - On Unix systems call the `hybrisserver.sh` file with the debug parameter, such as `./hybrisserver.sh debug`.

   For more information, see Configuration Templates.

   If you got this line in console during the start

   ```
   INFO   | jvm 1    | main    | 2009/10/29 15:33:25.449 | 2009-10-29 15:33:25.199/54.703 Oracle
   ```

   a. Rename `WEB-INF/classes/coherence-override-local.xml` to `tangosol-coherence-override.xml`.
   b. Rebuild SAP Commerce.
4. Verify whether Oracle Coherence has been set up correctly.
   - Check the log whether a second node is started and has notified that there is already Oracle Coherence cluster and it has been attached to it:

     ```
     INFO  [Coherence] 2009-10-30 09:45:24.280/99.687 Oracle Coherence EE 3.4.2/411 <
     Info> (thread=Cluster, member=n/a): This Member(Id=3, Timestamp=2009-10-30 09:45
     :23.85, Address=192.168.146.15:8089, MachineId=64015, Location=site:Local-Deskto
     p,machine:testmachine1,process:3228, Edition=Enterprise Edition, Mode=Development,
      CpuCount=2, SocketCount=2) joined cluster "Local-Dev" with senior Member(Id=1,
     Timestamp=2009-10-29 16:33:33.933, Address=192.168.146.160:8088, MachineId=64160
     , Location=site:Local-Desktop,machine:testmachine2,process:1148, Edition=Enterpr
     ise Edition, Mode=Development, CpuCount=4, SocketCount=4)
     ```

   - Verify the correct setup of Oracle Coherence via JMX.

ImpEx API

The `ImpEx` API allows you to interact with the `ImpEx` extension. You can use the API to import and export data, and extend or adapt it to your needs.

For details of the ImpEx user interface in Backoffice or SAP Commerce Administration Console, see Using ImpEx with Backoffice or SAP Commerce Administration Console.

Import API

You can trigger an import using the import API in a number of ways. These include using the back end management interfaces, as well as triggering it programmatically

Export API

You can trigger an export using the export API in a number of ways. These include using the back end management interfaces, as well as triggering it programmatically.

Validation Modes

The validation mode controls validation checks on ImpEx. By default, strict mode is enabled meaning all checks are run.

Customization

You can extend your import or export process with custom logic. Customization allows you to addresses requirements that cannot be achieved completely with the ImpEx extension.

Scripting

You can use Beanshell, Groovy, or JavaScript as scripting languages within ImpEx. In addition, ImpEx has special control markers that determine simple actions such as **beforeEach**, **afterEach**, **if**.

User Rights

The ImpEx extension allows you to modify access rights for users and user groups.

Translator

A translator class is a converter between ImpEx-related CSV files and values of attributes of SAP Commerce items

Import API

You can trigger an import using the import API in a number of ways. These include using the back end management interfaces, as well as triggering it programmatically

There are four basic ways of triggering an import of data for the `ImpEx` extension:

1. Using the Backoffice ImpEx Import Wizard. For details, see Import Wizard.
2. Creating an Import CronJob in Backoffice. For details, see Import CronJob.
3. Using `ImpEx Import` page in SAP Commerce Administration Console. For details, see Import Through Administration Console.
4. Using the Import API, which is described here.

To support the systems with large numbers of products, SAP Commerce has a capability of multithreaded import operations. For details, see Multithreaded Import.

You have several possibilities to perform an import programmatically. The decision depends mainly on the specialized configuration needs. The basic kind of processing is the instantiation and configuration of the **Importer** class. For detailed information, Using an Importer Instance. Here you have the full range of configuration possibilities. The instantiation and configuration of an **Importer** class triggers an import cronjob too, but it additionally provides the features of a cronjob, that is, all settings, results, and logs are stored as persistent, which is strongly preferred. The third convenient alternative is the usage of the API methods of the **ImpExManager** method. For detailed information, see Using a Method of ImpExManager. They also use an import cronjob, but you do not have to create and configure it on your own.

Using an Importer Instance

The **Importer** class is the central class for processing an import. The process for importing by directly using this class has 3 steps.

Procedure

1. Instantiate the class.

   While instantiating the **Importer** class, this CSV-stream is given using a **CSVReader** or an **ImpExImportReader**. If you only want to specify the input stream, use an **CSVReader** class, the **Importer** instantiates a corresponding **ImpExImportReader**. The usage of an **ImpExImportReader** is only needed, if special settings while instantiation are needed (settings after instantiation can be done using the **getReader()** method of the **Importer** instance).

   Example:

   ```
   CSVReader reader  = new CSVReader( "input.csv", "utf-8" );
   Importer importer = new Importer( reader );
   ```

   or:

   ```
   Importer importer = new Importer( new ImpExImportReader( reader, new MyImportProcessor() ) );
   ```

2. Configure the import process.

   You have several possibilities for configuring the import process.

   - You can configure the used **ImpExImportReader** using the **getReader** method. Here you can configure several things about the reading of the input (**skipValueLines**, **enableCodeExecution**, and so on) or item processing (**setRelaxedMode**, and so on).
   - You can set a **DumpHandler** for specifying the dump file handling. Thirdly, you can set an **ErrorHandler** to specify the process in case of an error. Fourthly, you can set the maximal amount of passes for resolving dumped value lines.

3. Trigger the import.

   You can perform the import using the **importNext**, which processes the input stream until an item was processed (that means **inserted**, **updated** or **removed**). It returns the processed item. Another possibility is the usage of the **importAll** method, which calls the **importNext** method until finishing of the input stream. While and after the import process, you have several possibilities to get information about the state, for example: current pass, processed items, and so on.

   Example:

   ```
   Item item = null;
   do
   {
    item = importer.importNext();
    System.out.println( "Processed items: " + getProcessedItemsCountOverall() );
   }
   while( item != null );
   ```

   or:

   ```
   importer.importAll();
   System.out.println( "Processed items: " + getProcessedItemsCountOverall() );
   ```

Using ImpExImportCronJob

When using `ImpExImportCronjob`, you have the advantage of persistent logging, as well as persistent result and settings holding.

The possible settings you can find in the API of the cron job class. The following sample shows an example configuration:

```
try
{
  // Creating import media
  ImpExMedia jobMedia = createImpExMedia( "myImportScript", "UTF-8" );
  jobMedia.setFieldSeparator( ';' );
  jobMedia.setQuoteCharacter( "\"" );
  jobMedia.setData( new DataInputStream( ImpExManager.class.getResourceAsStream("myScript.impex")),
              jobMedia.getCode() + "." + ImpExConstants.File.EXTENSION_CSV, ImpExConstants.File.M

  // create cronjob
  ImpExImportCronJob cronJob = ImpExManager.getInstance().createDefaultImpExImportCronJob();
  cronJob.setEnableCodeExecution( codeexecution );
  cronJob.setJobMedia( jobMedia );

  // process import
  cronJob.getJob().perform( cronJob, true );
}
catch( UnsupportedEncodingException e )
{
  log.error( "Given encoding is not supported", e );
}
```

Using a Method of ImpExManager

The `ImpExManager` class provides methods with the qualifier **importData**. These methods all use a cron job for performing an import from a given source and help you to simplify the import call.

Important parameters for the import methods are as follows:

- **synchronous** - sets the created cronjob to be performed synchronous or asynchronous.
- **removeOnSuccess** - sets the cronJob and created medias to be removed if finished successfully.
- **codeExecution** - sets the execution of BeanShell code to be enabled.

If the **removeOnSuccess** flag is set, the resulting cronjob may not be valid anymore after import, because it is already removed.

```
InputStream is = ImpExManager.class.getResourceAsStream( csv );
 ImpExManager.getInstance().importData( is, "windows-1252",
        CSVConstants.HYBRIS_FIELD_SEPARATOR, CSVConstants.HYBRIS_QUOTE_CHARACTER, true );
```

Another set of methods for import are the **importDataLight** methods. These are lightweight methods, note that they have prefix **light**, using no cronjob, and so no persistent and logging offset.

```
InputStream is = ImpExManager.class.getResourceAsStream( csv );
ImpExManager.getInstance().importDataLight( is, "windows-1252", true );
```

Data Inclusion

Often you want to separate your ImpEx script logic from the data for example if the data is provided by a foreign system, and you do not want to touch this file by adding a ImpEx header. This is possible by using the `include` methods of the `ImpExReader` instance registered at the BeanShell context.

Storing both header definitions and data in one single file is a possible approach for smaller files. However, if you want to separate your **ImpEx** script logic from the data, for example, if you need to process externally supplied data (from a customer or supplier), or if your file exceeds several hundred lines of data, you probably want to split it. The **ImpEx** extension allows you to include other files within the parse process using the **include** methods of the **ImpExReader** class.

Basically there are three possibilities for inclusion, first the reference of input streams outside of the platform like files from file system, second the referencing of medias from the platform, and the third alternative is the direct inclusion of data from a database.

Inclusion of Data using Input Streams

The inclusion of input streams within the parse process can be achieved by using the **includeExternalData** methods (marked with the **BeanShell** annotation) of the **ImpExReader** instance registered as **ImpEx** variable at the BeanShell context. Just add such a call to the script and the stream is included directly at the position of the call within the parse process. So be sure that there is a header written at the main script before calling an include containing only raw data.

```
INSERT_UPDATE Product;code[unique=true];...
                                       "#% impex.includeExternalData(ImpExManager.class.getResourc
```

The common parameters used at most method signatures are:

Name	Type	Default value	Description
linesToSkip	int	0	Amount of lines that are skipped when start reading from external data.
columnOffset	int	-1	The column offset compared to **ImpEx** standard: if the first column already contains data use -1.
encoding	string	UTF-8	The encoding of the external CSV data.
delimiter	char	;	Field separator used in external data.

An other method signature alternative allows to set a **CSVReader** instance instead of an input stream where you can configure some additional parameters like the maximal buffer size for reading a cell, which is spread over many lines (By default a **ImpEx** CSV cell can only consist of 10000 lines).

```
INSERT_UPDATE Product;code[unique=true];...
                                       "#% CSVReader reader = new CSVReader( ImpExManager.class.ge
                                       "#% reader.setMaxBufferLines(100000l);"
                                       "#% impex.includeExternalData( reader, 1, -1 );"
```

ℹ Note

You can chain includes of data, so you can include data within an already included data. Please do not use the **ImpExManager.importData()** methods within BeanShell code, because it starts a completely separate import and with that a completely different context as for example the BeanShell interpreter and the resolving of dumped lines.

Inclusion of Data using Media

If you already have your data file added to the platform as an instance of a **ImpExMedia** type, you can simply include the media using the following BeanShell call:

```
INSERT_UPDATE Product;code[unique=true];...
                                        "#% impex.includeExternalDataMedia( ""MyDataMediaCode"" );'
```

The advantage of using the media type is you can do the configuration using the properties of the media instance, so you just give the **code** of the media as parameter.

Unfortunately there can be medias with the same code, so it is necessary to provide the import process with a collection of all possibly included medias. You can only do it by using an import procedure where an import cronjob takes part, because the collection of medias is stored at the cronjob instance.

Summarized, to include data using a media, you have to use an import proceeding with cronjob, and you have to set the collection of all included medias at this cronjob (the attribute is called **externalDataCollection**).

Inclusion of Data Through Database Statements

Using the BeanShell, you can also include external data directly from a database connection. Therefore you first have to open a JDBC connection using the **initDatabase** method. Secondly, you can include the data giving a SQL-query whose result set fits the defined header line.

```
#  Initialization
                                        INSERT_UPDATE XYType; $code[unique=true]; $mandant; $typ; b
                                        #% impex.initDatabase( <dburl>, <user>, <password>, <driver
                                        "#%
                                        impex.includeSQLData(
                                        "" SELECT ""+
                                        ""   myProduct.ProductID, myProduct.Tenant, Variant.myVaria
                                        myProduct.ProductID + '-base::' + CAST( myProduct.Tenant AS
                                        "" FROM DB.SpecialProduct as Product JOIN DB.SpecialProduct
                                        ""   ON myProduct.ProductID = Variant.ID and myProduct.Tena
                                        "" WHERE ""+
                                        ""   Variant.myVariantID > 0 AND Product.variant ='xytype''
                                        );
                                        "
```

Both methods and their signatures can be accessed by using the registered **ImpEx** variable of type **ImpExReader**.

Resolving

In contrast to export, ImpEx import provides a resolving mechanism to deal with value lines that cannot be imported by the time the parser runs across them. These lines are imported as far as possible, the unresolved lines are dumped into a temporary file and read in again when the current file has completed parsing.

The following code snippet gives an example of a case where a line may remain unresolved:

```
INSERT Address; appartment; owner(Principal.uid)
                ;testApp; testUser
INSERT Customer;uid[unique=true]
;testUser
```

Lines 1 and 2 try to create an address using the user **testUser** as owner. However, **testUser** is defined in lines 3 and 4. Therefore, **testUser** does not exist when the attempt to create the address is attempted, and **testUser** cannot be assigned to the address at this point of time. The parser then writes the address definition into a temporary file. When the main file has completed, that temporary file is read in again. Now **testUser** exists, and the address can be finished. Such a further run is called **pass** and an execution of the sample script can result in the following log:

```
Starting import synchronous using cronjob with PK=141025265286919088 and name=00000014-ImpEx-Import
INFO - Starting import with pass 1
INFO - Starting pass 2
INFO - Import finished successfully within 00:00:00:328 (hh:mm:ss:ms)
```

The dump file is stored by default at the used cronjob (attribute **unresolvedDataStore** at **Log** tab) and contains in the above example the following lines:

```
insert Address;appartment;owner(Principal.uid)
,,cannot create due to unresolved mandatory/initial columns;test;testUser
```

If all mandatory columns in the value line were given and successfully translated, an item is created despite the fact that maybe other non-mandatory attributes are not resolved. After creation, an update of the resting columns is made. The second line of the dumped line example shows the value line, which was not completed successfully. The first column of the value line holds three pieces of important information separated by commas. The first field contains the type of the item already created when mandatory attributes were resolved, but not all non-mandatory ones. The second holds the PK of this created item. The third contains an error message, which explains the reason not all attributes were resolved successfully.

If the item is already created but not all attributes were resolved, the already set attributes are ignored with the **<ignore>** flag. So just search for attributes not marked with that flag. The following content of such a dump file shows that a value line representing a Product could not be imported completely because the detail attribute was not resolved. You can see that the first and second attribute of the value line contain an **<ignore>** marker, which means that the product was created setting these two attributes. The third attribute was not resolved so there is no **<ignore>** marker.

```
INSERT Product;code;catalogVersion[allowNull=true];detail(code)
Product,288342436307920,, column 3: cannot resolve value 'testDetail' for attribute 'detail';<ignor
```

If a dump file cannot be imported completely, again, another dump file is written, and the next pass starts. If a dump file has the same size as a dump file of its following pass the import is aborted with following log:

```
ERROR (master) [ImpExImportJob] Can not resolve lines any more ... aborting further passes (at pass
 Finally could not import 1 lines!
```

In this case, you have to evaluate (as described above) the last dump file stored at the used cronjob.

Please be aware that only header and value lines are dumped, no BeanShell. So there is no chance to execute BeanShell code at a second pass.

Using Header Abbreviations

ImpEx provides a way to shorten length column declarations by using regexp patterns and replacements.

Although the ImpEx header definition language provides a most flexible way of using custom column translators, their declaration can grow long. You can shorten them using the ImpEx alias syntax.

The following example shows how to use this syntax to shorten classification columns declaration by giving them a new syntax: Put this into your SAP Commerce platform `local.properties` file:

```
impex.header.replacement.1 =
   C@(\\w+) ...
   @$1[ system='\\$systemName', version='\\$systemVersion',
        translator='de...ClassificationAttributeTranslator']
```

Note that the line has been wrapped to make it more readable - you have to leave it on one line, of course. Also note the Java string notation has to be used, that's why there are double '\'s.

All parameters starting with ***impex.header.replacement*** are parsed as **ImpEx** column replacement rules. The parameter has to end with a number that defines the priority of the rule. This way ambiguous rules can be sorted.

So what's this for? The first part of the property C@(\w+) defines the new abbreviation pattern to be used to declare classification attribute columns with. The second part is the replacement text including the attribute qualifer match group $1. In fact, it contains the original special column declaration. Both parts are to be separated by '...'.

So all that is needed now to declare classification attribute column is this:

```
$systemName=MySys
$systemVersion=1.0

INSERT_UPDATE Product; code[unique=true]; C@attr1; C@attr2 ; ...
```

Now the whole translator definition is hidden and you do not need to declare any helper alias for that.

Nevertheless both alias definitions are mandatory to tell the classification column which system and version it should take its attribute from.

Export API

You can trigger an export using the export API in a number of ways. These include using the back end management interfaces, as well as triggering it programmatically.

You can trigger an export of data for the **ImpEx** extension in the following ways:

1. In Backoffice using the ImpEx Export Wizard. See Export Wizard.
2. In Backoffice, using an ImpExExportCronjob. See Export Using an ImpexExportCronJob.
3. In SAP Commerce Administration Console, using the `ImpEx` extension page. See Export Through Administration Console.
4. Using the export API which is described here.

While at an import script, the data to be imported is specified via value lines, an export script has a different structure to define the set of items to be exported, as well as the export file format. Find more information on the structure of an export script, see Structure of an Export Script. The available validation modes for an export script are described in Validation Modes.

You have several possibilities to perform an export programmatically. The decision depends mainly on the specialized configuration needs.

The basic kind of processing is the instantiation and configuration of the `Exporter` class. Here you have the full range of configuration possibilities. The instantiation and configuration of an `Exporter` does an export cronjob too, but it additionally provides the features of a cronjob, that is: all settings, results, and logs are stored as persistent, which is strongly preferred. The third convenient alternative is the usage of the API methods of the `ImpExManager`. They also use an export cron job, but you do not have to create and configure it on your own.

Export Using an Exporter Instance

The `Exporter` class is the central class for processing an export. You can use this class directly for exporting data in three steps.

Procedure

1. Define your export configuration.

   All configuration is done by instantiation of an `ExportConfiguration`. This is a container for all configuration you can set for the exporter and is passed at instantiation of the Exporter. The constructor needs an `ImpExMedia` where the export script is managed by and the validation mode given as `EnumerationValue`.

   ```
   ExportConfiguration config = new ExportConfiguration( impexscript, ImpExManager.getExportOnly
   ```

   Configuration of additional settings can then be done via the setter methods of the object, for example for setting the result medias to use explicitly. Note that otherwise they are created automatically.

   ```
   config.setDataExportTarget(datatarget);
   config.setMediasExportTarget(mediastarget);
   ```

2. Create an instance of the class.

   With the created `ExportConfiguration` you can now create an instance of the **Exporter** class using:

   ```
   Exporter exporter = new Exporter( config );
   ```

3. Start the export.

   You can start the export process using a simple call of the `export` method:

   ```
   Export export = exporter.export();
   ```

   The returned `Result` item contains all result medias accessible via getter methods.

Export Using an ImpexExportCronJob

You can generate an export using the ImpEx export cron job.

When using the `ImpExExportCronjob`, you have the advantage of persistent logging, as well as persistent result and settings holding. You can create a cron job using the `createDefaultImpExExportCronJob` method of the `ImpExManager` class. Possible settings are provided in the API of the cron job class. For further details, see Using the ImpExImportCronJob.

A sample configuration looks like this:

```
// Creating export media
ImpExMedia jobMedia = createImpExMedia( "myExportScript", "UTF-8" );
jobMedia.setFieldSeparator( ';' );
jobMedia.setQuoteCharacter( "\"" );
jobMedia.setData( new DataInputStream( ImpExManager.class.getResourceAsStream("myScript.impex")),
        jobMedia.getCode() + "." + ImpExConstants.File.EXTENSION_CSV, ImpExConstants.File.MIME_TYPE

// Creating an ExportConfiguration
ExportConfiguration config = new ExportConfiguration( impexscript, ImpExManager.getExportOnlyMode(

// Creating export cronjob
ImpExExportCronJob cronJob=createDefaultExportCronJob( config );

// export
cronJob.getJob().perform( cronJob );
```

```
// Get result
Export export=cronJob.getExport();
```

Using an Export Method of the ImpEx Manager

You can export data using dedicated methods provided by the `ImpExManager` class.

The `ImpExManager` class has different methods with the qualifier `exportData`. These methods all use a cronjob for performing an export with given `ExportConfiguration` and help you to simplify the import call. The only important additional parameter for this methods is `synchronous`. This parameter defines if the created cronjob is performed synchronous or asynchronous.

```
// Creating an ExportConfiguration with an media containing the script
ExportConfiguration config = new ExportConfiguration( impexscript, ImpExManager.getExportOnlyMode()
// Export
Export export = ImpExManager.getInstance().exportData( config, true );
```

Another set of methods for export are the `exportDataLight` methods. These are lightweight methods using no cronjob and so no persistent and logging offset.

```
// Creating an ExportConfiguration with an media containing the script
ExportConfiguration config = new ExportConfiguration( impexscript, ImpExManager.getExportOnlyMode()
// Export
Export export = ImpExManager.getInstance().exportData( config );
```

Structure of an Export Script

An export script needs to specify the target file to export items to, a header line for defining how to export the items, and a statement specifying which items to export.

The structure of a typical export script using the export of all languages is shown in the following example.

```
"#% impex.setTargetFile( ""language.csv"", true, 1, -1 );" // 1. where to export
                        insert_update Language;active;fallbackLanguages(isocode);isocode[unique=tru
                        "#% impex.exportItems( ""Language"" , true );" // 3. what to export
```

The sample shows a typical script. Here all languages of the system are to be exported to a file called `language.csv` using the given header. The order of the three statements is important.

1. The first line is used to specify where to export the items followed by the next header. Using the given file name, a file is created at the resulting archive where all items are written to and which are exported using the next header line. The **setTargetFile** method is located at the **Exporter** class and comes in different signatures all covering a different combination of the parameters. The signature using all parameters is:

   ```
   public void setTargetFile( final String filename, boolean writeHeader, int linesToSkip, int o
   ```

   where the parameters are:

Name	Type	Default	Description
filename	String	Type code configured at next header line.	Name for target file within resulting archive where the items of the next header are written.

Name	Type	Default	Description
writeHeader	boolean	true	If set, the header is written as a comment to the first line at the target data file.
linesToSkip	int	0	Parameter is used at generated include statement at the generated import script. In the import case, it means the amount of lines that are skipped when start reading from external data.
columnOffset	int	-1	Specifies the column offset of the target data file compared to ImpEx standard: if the first column already contains data use -1 - is set at generated include statement at generated import script.

If you use a signature with less parameters, the default values are used for the missing ones. Be aware that you do not have to give such a statement, it is optional. In that case, all default values are used (a file is created using the name of the type configured at the header line).

2. The second line is a header line describing how to export items. This is analogous to the import case except for the header mode (INSERT, UPDATE, and so on), which is ignored at the export. However a header mode has to be given, furthermore the header is copied to the generated import script, and so it is useful to give a correct one.
3. The **exportItems** call gathers the set of items to export and exports them using the set header and target file. The method is located at the **Exporter** class and comes in different signatures of different nature.
    - **exportItems** by item set:

      ```
      public void exportItems( Collection<Item> items )
                                                                  public void exportItems( String
      ```

      These methods export given items where the items can be passed either as a list of PK's (String) or directly using a Collection of items.

    - **exportItems** by type code:

      ```
      public void exportItems( String typecode )
                                                                  public void exportItems( String
                                                                  public void exportItems( String
                                                                  public void exportItems( String
      ```

      It gathers items by selecting all items of a given type code and exports them. The parameters are:

Name	Type	Default	Description
typecode	String	-	The **typecode** where all items are gathered and exported.
count	int	1000	A range parameter - many FlexibleSearches are performed each covering **count** items of the type -

Name	Type	Default	Description
			does not limit the overall count.
inclSubTypes	boolean	false	If true, subtypes of given type are considered.

- **exportItems** by FlexibleSearch:

  > **Note**
  > **Paging of Search Result**
  >
  > Exporter API by default uses pagination of search results, therefore, to have accurate results, your FlexibleSearch queries must contain the **ORDER BY** clause, for example **ORDER BY {pk}**. If you disable pagination or you use some method that cares also for ordering of results, then the **ORDER BY** clause is not needed. For more details read FlexibleSearch, section **Paging of Search Result**.

```
// since 3.1-RC
                                                    public void exportItemsFlexible
                                                    public void exportItemsFlexible
                                                    final boolean dontNeedTotal, in
                                                    // since 3.1-u6
                                                    public void exportItemsFlexible
```

  It gathers items using a given FlexibleSearch query. Be aware that the resulting items have to be compatible with the current header.

You can generate a script for all types of your platform by using the **ScriptGenerator** in Backoffice. For details, see Script Generator.

Validation Modes

The validation mode controls validation checks on ImpEx. By default, strict mode is enabled meaning all checks are run.

There are five different modes available, where two are only applicable for import and three for export.

- **Import Strict** - Mode for import where all checks relevant for import are enabled. This is the preferred one for an import.
- **Import Relaxed** - Mode for import where several checks are disabled. Use this mode for modifying data not allowed by the data model like writing non-writable attributes. Please be aware of the fact that this mode only disables the checks by ImpEx, if there is any busines logic that prevents the modification, the import fails anyway.
- **Export Strict (Re)Import** - Mode for export where all checks relevant for a re-import of the exported data are enabled. This is the preferred mode for export if you want to re-import the data as in migration case.
- **Export Relaxed (Re)Import** - Mode for export where several checks relevant for a re-import of the exported data are disabled.
- **Export Only** - Mode for export where the exported data are not designated for a re-import. There are no checks enabled, so you can write for example a column twice, which cannot be re-imported in that way. Preferred export mode for an export without re-import capabilities.

The following table gives an overview of which checks can be disabled by using a different mode than the strict one.

Check Description	Thrown Exception	Import Strict	Import Relaxed	Export Only	Export Relaxed	Export Strict	Check location (for developers)
Clashing of columns (columns referencing same attribute descriptor)	Ambiguous columns ***columns-related code***.	✅	⛔	⛔	⛔	✅	**HeaderDescriptor.validate**
Missing unique modifier	Missing unique modifier inside ***header-related code*** - at least one attribute has to be declared as unique (attributename[internal:unique=true]).	✅	✅	⛔	✅	✅	**HeaderDescriptor.validate**
No abstract type	Type ***type-related code*** is abstract.	✅	⛔	⛔	⛔	✅	**HeaderDescriptor.calculatePermittedType**
No jalo only type	Type ***type-related code*** is jaloOnly.	✅	⛔	⛔	⛔	✅	**HeaderDescriptor.calculatePermittedType**
Missing mandatory columns	Type ***type-related code*** requires missing column(s) ***column-related code***.	✅	⛔	⛔	⛔	✅	**HeaderDescriptor.calculatePermittedType**
Missing unique column	Type ***type-related code*** has no unique columns.	✅	⛔	⛔	⛔	✅	**HeaderDescriptor.calculatePermittedType**
Read-only (write=false, initial=false) column with unique modifier.	Unique attribute ***attribute-related code*** is read-only which is not allowed.	✅	⛔	⛔	⛔	✅	**StandardColumnDescriptor.validate**
Mandatory columns without value (no default value, no allowNull modifier)	Value is NULL for mandatory attribute ***attribute-related code***.	✅	⛔	⛔	⛔	⛔	**DefaultValueLineTranslator.translateColumnV**
Dumping is disabled and a line has to be dumped	Line ***line-related code*** could not be imported completely	✅	⛔	⛔	⛔	⛔	**ImpExImportReader.readLine**
Empty column expression	No attributes specified for header	✅	⛔	⛔	⛔	✅	**AbstractTypeTranslator.translateColumnDesc**

Check Description	Thrown Exception	Import Strict	Import Relaxed	Export Only	Export Relaxed	Export Strict	Check location (for developers)
	with type ***type-related code***.						
Column within item expression not searchable or has no persistence representation	Attribute ***attribute-related code*** is not searchable - cannot use to resolve item reference via pattern.	✓	✗	✗	✗	✓	**ItemExpressionTranslator.checkResolvableAtt**
Column type within item expression is jalo-only.	Attribute ***attribute-related code*** is jalo-only - cannot use to resolve item reference.	✓	✗	✗	✗	✓	**ItemExpressionTranslator.checkResolvableAtt**

Customization

You can extend your import or export process with custom logic. Customization allows you to addresses requirements that cannot be achieved completely with the ImpEx extension.

Writing A Custom Cell Decorator

Using a cell decorator you can intercept the interpreting of a specific cell of a value line between parsing and translating of it. It means the cell value is parsed, then the cell decorator is called, which can manipulate the parsed string and then the translation of the string starts.

You can configure the usage of a cell decorator by adding the modifier **cellDecorator** to a header attribute specifying your decorator class.

```
INSERT MyType;...;myAttribute[cellDecorator=de.hybris.platform.catalog.jalo.classification.eclass.E
```

The specified class has to implement the **CSVCellDecorator** interface, which specifies a method:

```
String decorate( int position, Map<Integer, String> srcLine )
```

Here the whole parsed value line is passed (srcLine) where the line is a mapping of column position (Integer) to the parsed String. Furthermore the position of the column which has to be decorated is passed. As a result, the method estimates the decorated String, a decoration of the String inside the map has no effect.

So a typical implementation is:

```
public class MyDecorator implements CSVCellDecorator
                                                {
                                                public String decorate( int position, Map<Integer,
                                                {
                                                String parsedValue=srcLine.get(position);
                                                return parsedValue+"modified"; // some decoration s
                                                }
                                                }
```

For more convenience, the **ImpEx** extension provides the **AbstractImpExCSVCellDecorator** implementation that additionally adds a member to the decorator holding a reference to the column descriptor where the decorator instance is applied. This descriptor is set via an additional init-method, so it is not available at the constructor calling phase. With that member you have access to the whole column specification especially all configured modifiers and the whole related header descriptor.

Writing A Custom Translator

If you have to change the translation logic for a value, then a decorator is not enough for your needs. In such cases you can write your own translator and configure it for the translation of specific attributes. You can do the configuration by simply adding the **translator** modifier to the desired header attribute like:

```
INSERT MyType;...;myAttribute[translator=de.hybris.platform.impex.jalo.translators.ItemPKTranslator
```

In case the attribute is a real type attribute, the configured class has to extend the **AbstractValueTranslator** class in some way. This abstract class defines two methods:

1. `public abstract Object importValue( final String valueExpr, final Item toItem )`

   Implement the translation logic for the import case here. The given String is the parsed cell value (possible decorators already applied) and the passed item instance is the resolved item for example in case of an update or an insert where the current attribute is not mandatory. The result object represents the translated value and must be an instance of the expected attribute type. In case of an error (like a referenced item can not be resolved), you can call the protected ***setError*** method, which marks the cell as unresolved.

2. `public abstract String exportValue( final Object value )`

   Implement the translation logic for the export case here. The given object has to be translated to a String that has to be returned.

In most cases you want to use translation logic that already partly exists because ImpEx already provides a collection of translators. So please check first to see if you can extend an existing one.

Writing A Custom Special Translator

A special translator has to be written and configured in case you want to change the translation logic for special attributes, for example header attributes that have no counterpart at the type system. They are configured as normal translators:

```
INSERT MyType;...;@myAttribute[translator=de.hybris.platform.impex.jalo.media.MediaDataTranslator]
```

The configured translator has to implement the **SpecialValueTranslator** that specifies different methods. Thereby two methods are responsible for the real translation (import and export case), if you want to have default implementations for the other methods, just extend the **AbstractSpecialValueTranslator**.

The two methods are:

1. `public void performImport( final String cellValue, final Item processedItem )`

   where you have to implement the translation logic for import. As for normal translators, you get the parsed cell value and the already resolved item instance. You have to return the translated value.

2. `public String performExport( final Item item )`

   where you have to implement the logic for export, which means you have to translate given object to a String

Writing A Custom Script Modifier for Script Generator

The **ScriptGenerator** generates a default export script for all types of systems. Often you want to change the script generation for special cases where the **ScriptModifier** comes in play. The configured ScriptModifier instance of a ScriptGenerator instance is asked in several cases, for example which root types have to be considered for traversing the type system tree.

You can configure your ScriptModifier by adding an instance to the ScriptModifierEnum enumeration where the code of the new value has to represent your class name where dots are replaced by underscores, like:

```
de_hybris_platform_impex_jalo_exp_generator_MigrationScriptModifier
```

With that you can choose ScriptModifier at the ScriptGenerator wizard. The class specified at the enumeration has to implement the **ScriptModifier** interface.

The following methods are defined by this interface:

1. `void init( ScriptGenerator generator )`

   Add logic here that configures the related generator instance to your needs for example by calling

   ```
   generator.addIgnoreColumn( "Item", "allDocuments" );
   ```

   for excluding the **allDocuments** attribute in general. See the APIDoc of the **ScriptGenerator** interface for configuration possibilities.

2. `boolean filterTypeCompletely( ComposedType type )`

   Here you have the chance to filter a type completely from script generation. A static default implementation that filters jaloonly types and views can be found at .filterTypeCompletely

3. `Set<ComposedType> getExportableRootTypes( ScriptGenerator callback )`

   Here you can specify the root set of types for which an entry at the generated export script is created. In case you do not want to implement the method, you can use a call to **ExportUtils.getExportableRootTypes**.

Scripting

You can use Beanshell, Groovy, or JavaScript as scripting languages within ImpEx. In addition, ImpEx has special control markers that determine simple actions such as **beforeEach**, **afterEach**, **if**.

The following is an example of a simple ImpEx import script using Beanshell:

```
INSERT_UPDATE Title;code[unique=true]
                                    #% beforeEach: line.clear();
                                    ;foo
```

With the scripting engine support in SAP Commerce, you can set the value of the flag `impex.legacy.scripting` to `false` to benefit from new scripting features in ImpEx. You can then use not only Beanshell, but also Groovy and JavaScript.

The same example, but rewritten in Groovy, now looks like the following:

```
INSERT_UPDATE Title;code[unique=true]
                                    #%groovy% beforeEach: line.clear();
                                    ;foo
```

All you need to do is tell the console what language you use, for example `%groovy%`.

Standard Imports

By default, a number of standard imports are always provided to you by default in ImpEx scripting, so you do not need to call them yourself. Here is a list of those imports:

```
import de.hybris.platform.core.*
import de.hybris.platform.core.model.user.*
import de.hybris.platform.core.HybrisEnumValue
import de.hybris.platform.util.*
import de.hybris.platform.impex.jalo.*
import de.hybris.platform.jalo.*
import de.hybris.platform.jalo.c2l.Currency
import de.hybris.platform.jalo.c2l.*
import de.hybris.platform.jalo.user.*
import de.hybris.platform.jalo.flexiblesearch.*
import de.hybris.platform.jalo.product.ProductManag
```

Keep in mind that these imports are available only for Groovy and Beanshell languages. Javascript has a completely different concept of importing files so you need to use them in each script line.

Limitations

The previous Beanshell scripting implementation provided possibilites to write scripts like that:

```
#% import de.hybris.foo.bar.MyClass;
INSERT_UPDATE Title;code[unique=true]
#% beforeEach: MyClass.callStaticMethod();
;foo
```

In the provided example, we first import some class and in another line we refer to that class. In the new implementation this is not possible, because each line of the script is treated as a separate entity that does not know about others. So currently, we need to do it like this:

```
INSERT_UPDATE Title;code[unique=true]
"#%groovy% beforeEach:
import de.hybris.foo.bar.MyClass
MyClass.callStaticMethod()
";
;foo
```

Enabling ImpEx Scripting

To distinguish between the two modes of scripting in ImpEx, you can also use the property `impex.legacy.scripting`. This property is `true` by default, which means that the legacy behavior is the default. If you need new scripting capabilities, set the value of the property to `false`.

User Rights

The ImpEx extension allows you to modify access rights for users and user groups.

You can easily define user rights in ImpEx using the syntax demonstrated in the following example.

```
$START_USERRIGHTS
                        Type;UID;MemberOfGroups;Password;Target;read;change;create;delete;change_pe
                        UserGroup;impexgroup;employeegroup;
                        ;;;;Product;+;+;+;+;-
                        Customer;impex-demo;impexgroup;1234;
                        $END_USERRIGHTS
```

The following describes each line of the syntax:

- The **$START_USERRIGHTS** statement in line 1 indicates that every line until the **$END_USERRIGHTS** statement in line 6 isn't an ImpEx specific one, instead all lines between the two statements are redirected to the user rights management (**accessManager**).
- The second line is the header and defines the attributes that are to be set. This header is specific for the user rights management and doesn't support any features of ImpEx-like modifiers.
- The third line sets the **usergroup** with the name **impexgroup** that is a **MemberOfGroups** (subgroup) of **employeegroup** as an item where you modify given access rights. The item is the current one until the next item definition. If you forget to set a type (here UserGroup), the action will have no effect.
- The fourth line sets the permissions for **impexgroup**: + allows, - denies the respective access right. A skipped value specifies no override, that is, use the inherited setting.

Access right	Comment / Description
read	The user is allowed to read the value.
change	The user is allowed to change the value.
create	The user is allowed to create instances of this type.
delete	The user is allowed to delete instances of this type.
change_perm	The user is allowed to grant permissions to other users.

  If you need to define additional user rights for **impexgroup**, just add the respective lines after line four. Being a header, the **impexgroup** user right creation is active until the next user or group creation, and all permission definitions apply to it.
- The fifth line creates a user of type **Customer** named **impex-demo**. This user is a member of **impexgroup**, and their account password is **1234**.
- The sixth and last line ends the user right definitions section.

ℹ Note

You must specify the columns **Type**, **UID**, **MemberOfGroups**, and **Target** in the header even if you don't want to specify their values. Not specifying one of these columns in the header results in an error message like **This is not a CSV file with principal permissions! Aborting...**, and the import fails.

To set access rights to an **attribute** of a type, add the attribute identifier to the type identifier, separated by a full stop (.), as follows:

```
$START_USERRIGHTS
                        Type;UID;MemberOfGroups;Password;Target;read;change;create;delete;change_pe
                        UserGroup;impexgroup;employeegroup;
                        ;;;;Product.code;-;-;;;;
                        ;;;;Product.ean;-;-;;;;
                        $END_USERRIGHTS
```

You have to use all of the columns **Type**, **UID**, **MemberOfGroups**, and **Target** even if you don't want to specify their values. The permission columns at the header line don't matter in any case, but you can write them down to remember the meaning of the permission values but they aren't parsed. So if you change the permission columns at the header line, this has no effect.

In addition, you need to specify values for each of the potential access rights at the value line in exactly the order as in the example. The **ImpEx** framework implicitly expects an access rights line to contain values for all potential access rights in fixed order - even for attribute access rights, where only **Read** and **Change** are available.

Skipping one of these columns results in an error message like **This is not a CSV file with principal permissions! Aborting...**, and the import fails.

For example, the following **ImpEx** statements work, because they specify values for all the potential access rights:

- `$START_USERRIGHTS`

```
Type;UID;MemberOfGroups;Password;Target;read;
UserGroup;impexgroup;employeegroup;
;;;;Product.code;-;-;;;;
;;;;Product.ean;-;-;;;;
$END_USERRIGHTS
```

- `$START_USERRIGHTS`

```
Type;UID;MemberOfGroups;Password;Target;
UserGroup;impexgroup;employeegroup;
;;;;Product.code;-;-;;;;
;;;;Product.ean;-;-;;;;
$END_USERRIGHTS
```

The following ImpEx statements don't work, however, because they don't specify values for all the potential access rights:

- `$START_USERRIGHTS`

```
Type;UID;MemberOfGroups;Password;Target;read;
UserGroup;impexgroup;employeegroup;
;;;;Product.code;-;-
;;;;Product.ean;-;-
$END_USERRIGHTS
```

- `$START_USERRIGHTS`

```
Type;UID;MemberOfGroups;Password;Target;
UserGroup;impexgroup;employeegroup;
;;;;Product.code;-;-
;;;;Product.ean;-;-
$END_USERRIGHTS
```

Translator

A translator class is a converter between ImpEx-related CSV files and values of attributes of SAP Commerce items

A translator is one of the two ways SAP Commerce offers for using business logic when importing or exporting items.

On import, a translator converts an entry of a value line into a value of an SAP Commerce item. It writes the value from the CSV file into SAP Commerce.

On export, a translator converts a value of an attribute of a SAP Commerce item into a value line. It writes the value from SAP Commerce into a CSV file.

SAP Commerce comes with settings for translator classes for all default types. For all out-of-the-box types, there is a pre-set translator class. For custom type definitions, you can code a translator to handle custom attributes. To call a custom translator in ImpEx, you make a reference to its full classpath in the translator modifier in the header line of the ImpEx file, as follows:

```
INSERT_UPDATE myProduct;code;myAttribute[translator=my.company.org.translators.myCustomTranslator]
```

A translator is referenced for all value lines within a header. You cannot switch between translators within the scope of a header.

SAP Commerce comes with two kinds of translator classes: standard value translators, and special value translators. Each of these is dealt with separately in the following sections.

Standard Value Translators

Standard value translators are used for values that are mapped to standard type attributes, in contrast to special header attributes. If you do not specify a translator for a standard attribute, a fixed translator is chosen by default depending on the specified attribute type. Be aware that with the exception of the default translators, Standard Value Translators are not all part of the ImpEx extension itself. They can also be part of other modules as well, such as the **europe1** extension. As a consequence, a translator is not available if it is part of an extension that is not included in your installation.

The table below gives an overview on Standard Value Translators and their respective extensions.

Class	Applicable for Attribute Type	New Modifiers Introduced	Part of Extension
CollectionValueTranslator (used for Collections by default)	Collection	-	**impex**
Europe1ProductDiscountTranslator	Collection	price-format, date-format	**europe1**
Europe1UserDiscountsTranslator	Collection	price-format, date-format	**europe1**
Europe1PricesTranslator	Collection	price-format, date-format	**europe1**
ActiveDirectoryGroupCollectionTranslator	Collection	groupid	**ldap**
TaxValuesTranslator	Collection	-	**impex**
MapValueTranslator (used for Maps by default)	Map	-	**impex**
AtomicValueTranslator (used for AtomicTypes by default)	Atomic	-	**impex**
EClassAttributeTypeTranslator	ClassificationAttribute	-	**catalog**
EClassUnitTranslator	ClassificationAttributeUnit	systemName, systemVersion	**catalog**
ETIMAttributeTypeTranslator	ETIMAttribute	-	**catalog**
ItemExpressionTranslator (used for ComposedTypes by default)	ComposedType	-	**impex**
AlternativeExpressionTranslator (used for Alternative Patterns by default)	ComposedType	-	**impex**
ItemPKTranslator (used for PKs by default)	PK	-	**impex**
ProfiClassAttributeTypeTranslator	ProfiClassAttribute	-	**catalog**

Class	Applicable for Attribute Type	New Modifiers Introduced	Part of Extension
TaxValueTranslator	TaxValue	-	**impex**

Special Value Translators

Special Value Translators are used for values that have no exact attribute match, or for values that require complex business logic to resolve. Unlike Standard Value Translators, you have to explicitly enable Special Value Translators via the translator modifier. The following is a list of all Special Value Translators in an out-of-the-box SAP Commerce installation.

MediaDataTranslator

A media consists of several attributes, especially a URL where the described data can be found. Thereby the data is held locally at the **`mediaweb`** folder of the platform.

When inserting a media item, you can proceed like inserting a product by simply writing a header and defining the value lines. Problem is that there is no attribute that can be used for setting the real media data. This means you need a special column descriptor introduced by @ for defining a column without a mapped attribute. You have to specify the **de.hybris.platform.impex.jalo.media.MediaDataTranslator** as a translator. This translator holds an **MediaDataHandler,** which by default is the **DefaultMediaDataHandler**, when using a cronjob the **DefaultCronjobMediaDataHandler**.

The **MediaDataTranslator** translates the column values by calling the handler, which set the data to the already created/found media.

The **DefaultMediaDataHandler API** offers support for importing medias from different sources (ZIP files, classpath, URL, filesystem).

An example of using the DefaultMediaDataHandler is shown in the following ImpEx sample.

```
$catalogVersion=catalogVersion(catalog(id),version)[unique=true,default='mycatalog:Staged']

INSERT_UPDATE Media;code[unique=true];$catalogVersion; mime;realfilename;@media[translator=de.hybri

# ZIP Sample  Syntax: zip-prefix:path-to-zip&path-within-zip
; demo1; ; image/jpeg; demo1.jpg; zip:c:\demo.zip&demo1.jpg

# 1. CLASSPATH Sample  Syntax: jar-prefix:path-within-classpath
; demo2; ; image/jpeg; demo2.jpg; jar:/media/jeans/demo2.jpg

# 2. CLASSPATH Sample  Syntax: jar-prefix:any-class-within-classpath-for-getting-classloader&path-v
; demo2; ; image/jpeg; demo2.jpg; jar:de.hybris.platform.sampledata.jalo.SampleDataManager&/media/j

# URL Sample  Syntax: url-prefix:url
; demo3; ; image/jpeg; demo3.jpg; http:http://www.company.com/pictures/logo.gif

# FILE Sample Syntax: file-prefix:full-path
; demo4; ; image/jpeg; demo4.jpg; file:c:\demo4.jpg

# Exploded JAR Syntax: /medias/fromjar/file-name
; demo5; ; image/jpeg; demo5.jpg; /medias/fromjar/demo5.jpg
```

The **DefaultCronjobMediaDataHandler** extends the **DefaultMediaDataHandler** by additionally allowing you to specify a path without prefix and searches at the resource media attached at the cron job item. In that case, see Scripting.

Furthermore, you can set any **MediaDataHandler** at run-time by using the static method **setMediaDataHandler()** of the **MediaDataTranslator**.

UserPasswordTranslator

A special translator for importing/exporting user passwords. It stores passwords directly by specifying the desired encoding together with the encoded password separated by a colon (:). Refer to Password Storage Strategies to learn how to configure available password encodings. Use as follows:

```
INSERT Employee; uid[unique=true]; @password[translator=de.hybris.platform.impex.jalo.translators.l
; fritz ; md5:a7c15c415c37626de8fa648127ba1ae5
; max ; *:plainPassword
```

VelocityTranslator

The VelocityTranslator is used only for export. It can be used for exporting a value for an item using a velocity expression. With that, you can export a constant value or can aggregate different attributes.

Using the following header at an export of an iorder item, the code of the order is exported as well as the id and name of the user owning the order and its payment address, street name, and country.

```
INSERT_UPDATE Order; code[unique=true]; \
   @template1[translator=de.hybris.jakarta.ext.impex.jalo.translators.VelocityTranslator, expr='$it
   @template2[translator=de.hybris.jakarta.ext.impex.jalo.translators.VelocityTranslator, expr='$it
   "@template3[translator=de.hybris.jakarta.ext.impex.jalo.translators.VelocityTranslator, expr='$:
   @template4[translator=de.hybris.jakarta.ext.impex.jalo.translators.VelocityTranslator, expr='$it
```

ClassificationAttributeTranslator

Instead of importing each product feature one by one you can assign all features of a product with one value line using this translator. Therefore you have to declare a special attribute for each feature to import. Assuming you want to set a value for feature ***type*** at your product a suitable header could be as follows:

```
UPDATE Product; code[unique=true]; @type[system='SampleClassification',version='1.0',translator=de.
```

In this example, the modifiers **system** and **version**, which are both mandatory, specify the classification system version of the product feature.

Related Information

ImpEx
Importing LDAP Data Using ImpEx
JavaDocs

Installer Platform Plugin

The Platform Plugin is a standard Gradle plugin. It provides an API for managing Platform instances, as well as Tomcat instances in Gradle build files, in an easy, and readable way.

Enabling the Platform Plugin

You enable the Platform plugin in a Gradle build file. Once you have enabled the plugin, you can set up and configure any existing Platform instance. To enable the Platform plugin, use the following method:

```
apply plugin: 'platform-plugin'
```

Environment Variables

`DRIVER_JAR_PATH` is a path to the jdbc driver jar that must be copied into the Platform home `lib/dbdrivers` directory in some special cases when working with the installer.

Gradle Project Properties

The Platform plugin may use a project property passed to the build script to determine the Platform home:

```
-Pplatform_home=/path/to/bin/platform
```

This variable is useful in the development mode when testing plugin separately or if an installer project that is using the Platform plugin is outside of SAP Commerce. Please note that the installer install scripts use this property via the `-P` option - the `-Pplatform_home` switch may be used only when using gradle directly.

Setting Up the Platform Using the Platform Object

The following paragraph includes information about Platform Plugin variables and its methods.

Platform Plugin Variables

The Platform plugin enriches the standard Gradle **project** object with the following variables:

Variable	Description
`platformHome`	The absolute path to the Platform directory • Type: String • Examples: ◦ `/Users/foo/commerce-suite-5.4/hybris/bin/platform` ◦ `C:\foo\commerce-suite-5.4\hybris\bin\platform`
`installerHome`	The absolute path to the root directory of the installer project • Type: String
`installerWorkDir`	The work directory of the installer. • Type: String
`platformConfig`	The absolute path to the SAP Commerce config directory • Type: String • Examples: ◦ `/Users/foo/commerce-suite-5.4/hybris/config` ◦ `C:\foo\commerce-suite-5.4\hybris\config`
`suiteHome`	The absolute path to the SAP Commerce directory • Type: String • Examples: ◦ `/Users/foo/commerce-suite-5.4` ◦ `C:\foo\commerce-suite-5.4`

Variable	Description
`platformFactory`	The `platformFactory` object used to instantiate the `HybrisPlatform` object • Type: HybrisPlatformFactory
`tomcat`	The Tomcat builder object • Type: HybrisTomcat
`hsqldb`	The HSQLDB management object that makes it possible to run the HSQLDB as a separate process. • Type: HybrisHsqldb

The Platform plugin provides methods that are accessible directly from the build file so that they can be easily used with the `HybrisPlatform` object.

HybrisPlatformFactory, HybrisPlatform Objects

The standard Gradle project object includes a special object, the `HybrisPlatformFactory` object. It sets up and creates the `HybrisPlatform` object.

The `HybrisPlatform` object can be created in two ways:

- by using `platformFactory`:

```
def pl = platformFactory.createPlatform {
    // ...
}
```

- or via the `project` method:

```
def pl = platform {
    // ...
}
```

HybrisPlatform Object

The `HybrisPlatform` object provides methods to configure the Platform, such as the configuration of databases, extensions, local properties, and clusters. It also provides convenient methods to write configurations to the appropriate files, compile and initialize the Platform, and run the Platform on Tomcat.

To create a simple Platform object, use the following code:

createPlatform Method

```
def myPlatform = platformFactory.createPlatform()
```

The code creates a simple Platform object called `myPlatform`. The object is configured with the required default settings, runs on an HSQLDB database, and has Platform extensions set only.

The following table presents the methods provided with the `HybrisPlatform` object:

Method	Description
`myPlatform.setup()`	Writes the setup to a config folder.
`myPlatform.build()`	Initializes the Platform.
`myPlatform.runInBackground()`	Runs the Platform on a Tomcat instance in background mode.

The Platform plugin also provides the `platform` shortcut method, accessible directly from the build file. It creates an instance of the `HybrisPlatform` object, and allows you to configure it. The method takes *<Closure>* as a parameter, as shown in the following example:

dbSetup

```
def myPlatform = platform {

    dbSetup {
        dbType 'mysql'
        dbUrl 'jdbc:mysql://localhost/hybris?useConfigs=maxPerformance&characterEncoding=utf8'
        dbUser 'dbuser'
        dbPassword 'passwd'
    }

    extensions {
        scanPath '/Users/foo/backoffice-extensions'
        extensionNames 'backoffice'
    }

    localProperties {
        property 'media.legacy.prettyURL', 'true'
    }
}
```

The code example does the following:

- Creates an instance of the `HybrisPlatform` object that is configured to use a MySQL database.
- Configures the Platform to use the `backoffice` extension (using an additional scan path to search for the extensions in the `/Users/foo/backoffice-extensions` folder).
- Adds the *<media.legacy.prettyURL>* property to the `local.properties` file.

You can also configure the `HybrisPlatform` object to be a node in a cluster. The following code excerpt shows how to do it:

```
def myPlatform = platform {
    dbSetup {
        dbType 'mysql'
        dbUrl 'jdbc:mysql://localhost/hybris?useConfigs=maxPerformance&characterEncoding=utf8'
        dbUser 'dbuser'
        dbPassword 'passwd'
    }

    extensions {
        scanPath '/Users/foo/backoffice-extensions'
        extensionNames 'hmc', 'backoffice'
    }

    clusterSettings {
        enableAutodiscovery()
        udpMulticast()
```

```
    }
}
```

For more detailed information about cluster settings, see Advanced Cluster Settings API

HybrisPlatform API

In this paragraph you can find detailed information about the `HybrisPlatform` API.

- `dbSetup(Closure)` sets up the database connection. Note that the database must already exist. It takes *<Closure>* as a parameter in which you can set the required information using the following methods:
  - `dbType(String)`: The type of the database. The supported values are *<mysql>*, *<oracle>*, *<sqlserver>*, *<sap>*, and *<hsqldb>*.
  - `dbUrl(String)`: The URL to connect to the database.
  - `dbUser(String)`: Database user name.
  - `dbPassword(String)`: User database password.
  - `dbTablePrefix(String)`: Prefix for the database table.
  - `dbDriverJar(String)`: An optional path to the custom JDBC driver jar.

  dbSetup

  ```
  dbSetup {
      dbType 'mysql'
      dbUrl 'jdbc:mysql://localhost/hybris?useConfigs=maxPerformance&characterEncoding=utf8'
      dbUser 'myDbUser'
      dbPassword 'somePassword'
      dbTablePrefix 'foobar'
      dbDriverJar '/tmp/mysql-connector-java-5.1.31.jar'
  }
  ```

- `localProperties(Closure)` is used to specify custom properties. It takes *<Closure>* as a parameter in which you can set the required information using the following methods:
  - `property(String, String)`: Property key, property value.
  - `properties(Map<String, String>)`: Map of properties.
  - `customConfig(String)`: Name of the file used to store custom settings. All settings are merged.

    > ⚠ **Caution**
    >
    > The settings specified in this file overwrite the settings specified in a recipe.

  localProperties

  ```
  localProperties {
      property 'foo.key', 'foo.value'
      properties(
          'bar.key': 'bar.value',
          'baz.key': 'baz.value'
      )
      customConfig 'mySettings.properties'
  }
  ```

- `extensions(Closure)` is used to add a custom set of extensions to the configured Platform. It takes *<Closure>* as a parameter to set the required information using the following methods:
  - `extensionNames(String...)`: Extension names expressed as *<varargs>*.
  - `extName(String)`: Extension name.

- `extensionDirs(String...)`: Extension directories expressed as *<varargs>*.
- `extDir(String)`: Extension directory.
- `scanPath(String)`: Directory in which the Platform looks for the extensions provided by the `extensionNames` method or the `extName` method. Note that the *<autoload>* option must be set to *<false>*, that is, it must be disabled.
- `scanPath(String, Integer)`: Directory in which the Platform looks for extensions provided by the `extensionNames` method or the `extName` method. Note that the *<autoload>* option must be set to *<true>*, that is, it must be enabled and the `depth` option set to a specified scan depth.
- `scanPathWithAutoLoad(String)`: Directory in which the Platform looks for the extensions provided by the `extensionNames` method or the `extName` method. Note that the *<autoload>* option must be set to *<true>*, that is, it must be enabled.
- `scanPathWithAutoLoad(String, Integer)`: Directory in which the Platform looks for the extensions provided by the `extensionNames` method or the `extName` method. Note that the *<autoload>* option must be set to *<true>*, that is, it must be enabled).
- `disableDefaultScanPath()`: It disables the default scan path (*<$HYBRIS_BIN_DIR>*).
- `defaultScanPathWithAutoLoad()`: It forces the default scan path (*<$HYBRIS_BIN_DIR>*) to have the *<autload>* option set to *<true>*.
- `defaultScanPathWithAutoLoad(Integer)`: It forces the default scan path (*<$HYBRIS_BIN_DIR>*) to have the *<autload>* option set to *<true>* with a specified scan depth.
- `webApp(Closure)`: Deployment of external webapps. It takes *<Closure>* as a parameter to set the required information using the following methods:
  - `context(String)`: Path to `context.xml` file.
  - `contextRoot(String)`: Root context of the webapp.
  - `path(String)`: Path to the external `war` file.
- `customConfig(String)`: Name of the file used to store custom settings. All settings are merged.

  > **⚠ Caution**
  >
  > The settings specified in this file overwrite the settings specified in a recipe.

extensions

```
extensions {
    scanPath '/Users/foo/additional-extensions1'
    scanPathWithAutoLoad '/Users/foo/load-all-extensions'
    extensionNames 'foo', 'bar', 'baz'
    extName 'one'
    extName 'two'
    extensionDirs '/Users/foo/ext1', '/Users/foo/ext2'
    extDir '/Users/foo/ext3'
    extDir '/Users/foo/ext4'
    webApp {
        context '/foo/bar/context.xml'
        contextRoot 'foo'
        path '/one/two/three.war'
    }
    customConfig 'myExtensions.xml'
}
```

- `mediaStorageSettings(Closure)` is used to specify the media storage configuration. By default, the Platform is configured to use local media storage out-of-the-box, therefore no special configuration is required. This method takes *<Closure>* as a parameter to set the required information using the following methods:
  - `defaultHashingDepth(Integer)`: Specifies the number of subdirectories permitted in a storage. Permitted values are 0 to 4.
  - `disableLocalFileCache()`: disables local media file cache
  - `localStorageStrategy(Closure)`: Sets up the local media storage. This method takes *<Closure>* as a parameter to set the required information using the following methods:

- `defaultStorageStrategy()`: Specifies the storage strategy to be used as the default storage strategy for the entire SAP Commerce.
- `defaultUrlStrategy()`: Sets `localMediaWebURLStrategy` as the default URL storage strategy.
- `folders(String...)`: Specifies the storage strategy to be used as the strategy for the folders. This method takes the variable argument list of folder qualifiers.
- `foldersWithUrlStrategy(String...)`: Sets the storage strategy as a strategy, as well as the `localMediaWebURLStrategy` as the URL strategy for folders. This method takes the variable argument list of folder qualifiers.
- `dir(String)`: Sets the custom directory to be used as the default storage directory.

mediaStorageSetting

```
// Remaping media dir to /tmp/myMedias - useful for cluster setup
mediaStorageSettings {
    localStorageStrategy {
        dir '/tmp/myMedias'
    }
}

// Setting Local Storage strategy for two folders. This make sense in case when ot
mediaStorageSettings {
    localStorageStrategy {
        folders 'folderOne', 'folderTwo'
    }
}
```

- `s3StorageStrategy`: Sets up the s3 media storage. This method takes *<Closure>* as a parameter to set the required information using the following methods:
  - `defaultStorageStrategy()`: Sets the specified storage strategy as the default storage strategy for the entire SAP Commerce.
  - `defaultUrlStrategy()`: Sets the `s3MediaURLStrategy` media strategy as the default URL storage strategy. If you don't set this option, the `localMediaWebURLStrategy` are used by default.
  - `folders(String...)`: Sets this storage strategy as the strategy for the folders. This method takes the variable argument list of folder qualifiers.
  - `foldersWithUrlStrategy(String...)`: Sets the storage strategy as a strategy, as well as the `s3MediaURLStrategy` as the URL strategy for the folders. This method takes the variable argument list of the folder qualifiers.
  - `accessKey(String)`: Sets the access key for the S3 account.
  - `secretAccessKey(String)`: Sets the secret access key for the S3 account.
  - `endPoint(String)`: Sets the end point URL for the S3 storage.
  - `urlSigned(boolean)`: Specifies if the S3 URL strategy will produce signed URLs. This option is only applied when the `defaultUrlStrategy()` is used.
  - `timeToLiveInMinutes(int)`: Specifies the length of time, in minutes, for which the signed URL is valid. This option is only applied when `defaultUrlStrategy()` and `urlSigned(true)` are used.
  - `urlUnsignedHttps(boolean)`: Specifies if the unsigned URL uses HTTPS. This option is only applied when `defaultUrlStrategy()` and `urlSigned(false)` are used.
  - `urlUnsignedVirtualHost(boolean)`: Specifies if the unsigned URL uses virtual hosting. This option is only applied when `defaultUrlStrategy()` and `urlSigned(false)` are used.

s3StorageStrategy

```
// Setting S3 Storage Strategy as default (standard local URL strategy)
mediaStorageSettings {
    s3StorageStrategy {
        defaultStorageStrategy()
        accessKey 'SOME-KEY'
        secretAccessKey 'SOME-SECRET'
        endPoint 'http://somewhere.com/'
```

```
    }
}

// Setting S3 Storage Strategy for two folders
mediaStorageSettings {
    s3StorageStrategy {
        accessKey 'SOME-KEY'
        secretAccessKey 'SOME-SECRET'
        endPoint 'http://somewhere.com/'
        folders 'folderOne', 'folderTwo'
    }
}

// Setting S3 Storage Strategy as default (S3 Url strategy with default settings)
mediaStorageSettings {
    s3StorageStrategy {
        defaultStorageStrategy()
        defaultUrlStrategy()
        accessKey 'SOME-KEY'
        secretAccessKey 'SOME-SECRET'
        endPoint 'http://somewhere.com/'
    }
}
```

- `azureStorageStrategy(Closure)`: Sets up the Windows Azure Blob storage strategy. This method takes *<Closure>* as a parameter to set the required information using the following methods:
  - `defaultStorageStrategy()`: Sets the specified storage strategy as the default storage strategy for the entire system.
  - `defaultUrlStrategy()`: Sets the `windowsAzureBlobURLStrategy` as the default URL storage strategy. If you don't set this option, the `localMediaWebURLStrategy` is used as the default strategy.
  - `folders(String...)`: Sets this storage strategy as the strategy for the folders. This method takes the variable argument list of folder qualifiers.
  - `foldersWithUrlStrategy(String...)`: Sets the storage strategy as a strategy, as well as `windowsAzureBlobURLStrategy` as the URL strategy for the folders. This method takes the variable argument list of the folder qualifiers.
  - `connection(String)`: Sets up the connection string for the Azure account.
  - `publicBaseUrl(String)`: Sets up the public base URL for the Azure account.
  - `containerAddress(String)`: Specifies the container address in which all media will be stored.

azureStorageStrategy

```
// Setting Windows Azure Blob storage strategy as default (standard local URL strategy)
mediaStorageSettings {
    azureStorageStrategy {
        defaultStorageStrategy()
        connection 'CONNECTION-STRING'
        containerAddress 'MY-CONTAINER'
        publicBaseUrl 'some-account.blob.core.windows.net'
    }
}

// Setting Windows Azure Blob storage for two folders
mediaStorageSettings {
    azureStorageStrategy {
        connection 'CONNECTION-STRING'
        containerAddress 'MY-CONTAINER'
        publicBaseUrl 'some-account.blob.core.windows.net'
        folders 'folderOne', 'folderTwo'
    }
}

// Setting Windows Azure Blob storage strategy as default (Windows Azure Url strategy w
mediaStorageSettings {
    azureStorageStrategy {
        defaultStorageStrategy()
        defaultUrlStrategy()
        connection 'CONNECTION-STRING'
```

```
            containerAddress 'MY-CONTAINER'
            publicBaseUrl 'some-account.blob.core.windows.net'
        }
    }
```

- `gridFsStorageStrategy(Closure)`: Specifies the Mongo GridFS storage strategy. This method takes *<Closure>* as a parameter to set the required information using the following methods:
  - `defaultStorageStrategy()`: Sets this storage strategy as default one for entire system.
  - `defaultUrlStrategy()`: Sets `localMediaWebURLStrategy` as the default URL storage strategy. The GridFS storage doesn't provide a special URL strategy, therefore this method isn't mandatory.
  - `folders(String...)`: Sets this storage strategy as a strategy for folders. This method takes the variable argument list of the folder qualifiers.
  - `foldersWithUrlStrategy(String...)`: Sets the storage startegy as a strategy, as well as `localMediaWebURLStrategy` as the URL strategy for the folders. This method takes the variable argument list of the folder qualifiers.
  - `host(String)`: Sets the host for the MongoDB database. The default value is *<localhost>*.
  - `port(String)`: Specifies the port for the MongoDB database. The default value is *<27017>*.
  - `dbName(String)`: Specifies the name of the database. The default value is `hybris_storage`.
  - `userName(String)`: Specifies the user name for the MongoDB database. The default value is *<empty>*.
  - `password(String)`: Specifies the password for the MongoDB database. The default value is *<empty>*.

  gridFsStorageStrategy

```
// Setting Mongo GridFS storage strategy as default
mediaStorageSettings {
    gridFsStorageStrategy {
        defaultStorageStrategy()
        host '10.0.0.10'
        port '9090'
        userName 'some-user'
        password 'password'
    }
}

// Setting Mongo GridFS storage strategy for two folders
mediaStorageSettings {
    gridFsStorageStrategy {
        host '10.0.0.10'
        port '9090'
        userName 'some-user'
        password 'password'
        folders 'folderOne', 'folderTwo'
    }
}
```

- `clusterSetup(Closure)`: Used to specify the cluster node settings. This method takes *<Closure>* as a parameter to set the required information using the following methods:
  - `id(String)`: ID of a node.
  - `maxId(String)`: Max ID for the entire cluster.
  - `enableAutodiscovery()`:
  - `jGroups(Closure)`: Setting for the jGroups communication. This method takes a `Closure` as a parameter to set all necessary information by using the following methods:
    - `channelName(String)`: allows you to set up a JGroups channel name. Default is hybris-broadcast.
    - `configFile(String)`: the name of the configuration file.
    - `udpPing(Closure)`: allows you to set up JGroups UDP/PING communication. This method takes a `Closure` as a parameter to set all necessary information by using the following methods:
      - `mcastPort(int)`: allows you to set Multicast port for UDP/PING communication. Default is 45588.

- `udpPing()`: allows you to set JGroups UDP/PING communication with default settings.
- `tcpMping(Closure)`: allows you to set up JGroups TCP/MPING communication. This method takes a `Closure` as a parameter to set all necessary information by using the following methods:
  - `tcpAddress(String)`: allows you to set up a tcp bind address for TCP communication.
  - `tcpPort(int)`: allows you to set up a tcp bind port for TCP communication.
  - `multicastAddress(String)`: allows you to set up a multicast address for MPING communication.
  - `multicastPort(int)`: allows you to set up multicast port for MPING communication.
- `tcpMping()`: allows you to set up JGroups TCP/MPING communication with default settings.
- `tcpJdbcPing(Closure)`: allows to set up JGroups TCP/JDBC_PING communication. This method takes a `Closure` as a parameter to set all necessary information by using the following methods:
  - `tcpAddress(String)`: allows you to set up a tcp bind addres for TCP communication.
  - `tcpPort(int)`: allows you to set up a tcp bind port for TCP communication.
  - `clearTableOnViewChange()`: enabling this can help with removing crashed members that are still in the table. Default value is `false`
- `tcpJdbcPing()`: allows you to set up JGroups TCP/JDBC_PING communication with default settings.
- `tcpKubePing(Closure)`: allows you to set up JGroups TCP/KUBE_PING communication. This method takes a `Closure` as a parameter to set all necessary information by using the following methods:
  - `tcpAddress(String)`: allows you to set up a tcp bind addres for TCP communication.
  - `tcpPort(int)`: allows you to set up a tcp bind port for TCP communication.

- `jGroups()`: Default settings for the jGroups.
- `udpMulticast(Closure)`: Settings for the UDP multicast communication.
- `udpMulticast()`: Default settings for the UDP multicast.
- `udpUnicast(Closure)`: Settings for the UDP unicast communication.

- `clusterSettings(Closure)`: the same usage as `clusterSetup(Closure)`. This method is deprecated.
- `setup()`: Stores the properties and extensions settings in proper files, as well as optionally copies custom JDBC driver jar to the Platform `lib/dbdriver/` directory.
- `build()`: Executes the `ant clean all` command on the underlying Platform.
- `initialize()`: Executes the `ant initialize` command on the underlying Platform.
- `initialize(File)`: Executes `ant initialize` on underlying Platform and stores logs in a given file.
- `initializeTestSystem()`: Executes the `ant yunitinit` command on the underlying Platform.
- `update()`: Executes `ant updatesystem` on underlying Platform.

  `update(File)`: Executes `ant updatesystem` on underlying Platform and stores log in a given file
- `executeAntTarget(String)`: Enables the execution of any ant target supported by underlying `build.xml` file in the Platform.
- `executeAntTarget(String, String)`: Enables the execution of any ant target supported by underlying `build.xml` file in the Platform with additional `ant` options.

  ant targets

  ```
  executeAntTarget 'clean all initialize'
  executeAntTarget 'alltests -Dtestclasses.extensions=core'
  executeAntTarget 'initialize', '-Xmx2048m -XX:MaxPermSize=756M'
  ```

- `executeAntTarget(File, String)`: Allows you to execute any ant target and stores output in a given file.

- `executeAntTarget(File, String, String)`: Allows you to execute ant target with additional options and stores output in a given file.
- `start()`: Starts the configured Platform on a Tomcat in interactive mode.
- `startInDebug()`: Starts the configured Platform on a Tomcat in interactive and debug modes.
- `startInBackground()`: Starts the configured Platform on a Tomcat in background mode.
- `stopInBackground()`: Stops the Platform running on a Tomcat in background mode.
- `createProductionArtifacts()`: Creates production artifacts that are stored in a temp directory of the Platform.
- `createProductionArtifact(String args)`: Creates production artifacts that are stored in a temp directory of Platform; it allows you to pass additional command line arguments
- `importImpexFromContent(String content)`: Imports impex content kept in a string variable.

```
def importContent = '''
INSERT_UPDATE MediaFormat;qualifier[unique=true]
;foo;
;bar;
;baz;
'''
importImpexFromContent(importContent)
```

- `importImpexFromResource(String resourcePath)`: Imports impex content kept in a resource.

```
def resource = '/tmp/myImport.impex'
importImpexFromResource(resource)
```

- `executeScriptFromContent(String content, String scriptType)` : Executes the content of the script. Valid script types are **groovy**, **js**, **bsh**.

```
def script = '''
flexibleSearchService.search("SELECT {PK} FROM {MediaFormat}").getResult().forEach {
        println it.qualifier
}
''' executeScriptFromContent(content, 'groovy')
```

- `executeScriptFromResource(String resourcePath)`: Executes a script from the existing resource. Keep in mind that the resource file must have proper file extension (groovy, js or bsh).

```
def resource = '/tmp/myScript.groovy'
executeScriptFromResource(resource'
```

Before and After Setup/Build Hooks

You can schedule certain logic to be executed before and after the setup and build phases.

There are four methods that allow you to do that:

- `beforeSetup(Closure)`
- `afterSetup(Closure)`
- `beforeBuild(Closure)`
- `afterBuild(Closure)`

The following example shows how to provide logic to be executed after setup is done:

```
pl.afterSetup {
        println 'Called after setup'
    }
```

```
    pl.setup()
```

Testing Capabilities

The `HybrisPlatform` object includes an API for easy test execution.

Simply, inside your recipe, specify which tests and from which set of extensions you want to run:

```
def pl = platform {
    tests {
        extensions 'foo', 'bar'
        annotations 'UnitTest', 'IntegrationTest'
        packages 'foo.bar.baz.*'
        reportDir '/tmp/junit'
    }
}

pl.runTests()
```

The API includes the following methods:

- `tests(Closure)`: It allows you to configure test execution. This method takes *<Closure>* as a parameter to set the required information using the following methods:
    - `extensions(String...)`: Narrows test execution only to the names of provided extension.
    - `annotations(String...)`: Narrows test execution only to tests annotated with annotations. The options allowed are *<UnitTest>*, *<IntegrationTest>*, *<PerformanceTest>*, *<DemoTest>*
    - `packages(String...)`: Narrows test execution only to provided packages.
    - `excludedPackages(String...)`: Allows you to exclude packages from test execution.
    - `reportDir(String)`: Allows you to set different report directory. The default directory is `log/junit/` in the Platform main directory.
- `webTests(Closure)`: It allows you to configure execution of web tests. This method takes *<Closure>* as a parameter to set the required information using the following methods:
    - `extensions(String...)`: Narrows test execution only to the names of provided extension.
    - `annotations(String...)`: Narrows test execution only to tests annotated with annotations. The options allowed are *<UnitTest>*, *<IntegrationTest>*, *<PerformanceTest>*, *<DemoTest>*
    - `packages(String...)`: Narrows test execution only to provided packages.
    - `excludedPackages(String...)`: Allows you to exclude packages from test execution.
    - `reportDir(String)`: Allows you to set a different report directory. The default directory is a `log/junit/` in the Platform main directory.

Advanced Cluster Settings API

Setting up a cluster can be complicated, especially on a single-developer machine. The Platform Plugin API facilitates setting up a cluster.

To configure a cluster node, use the `cluster(Closure)` method; it is available in the `HybrisPlatform` and `PlatformInstance` objects.

Before configuring a cluster node, select the communication mechanism you want to use:

- jGroups
- UDP Multicast
- UDP Unicast

jGroups Configuration

To configure a jGroups-enabled node, use the following code:

```
clusterSettings {
    id '0'
    maxId '2'
    jGroups()
}
```

The code configures a cluster node with ID=0, max ID=2, and the following default jGroups communication settings:

- TCP address: `127.0.0.1`
- TCP port : `7800`
- Channel name: `hybris-broadcast`
- Configuration file: `jgroups-udp.xml`

You can configure the same cluster node, using a more verbose code:

```
clusterSettings {
    id '0'
    maxId '2'
    jGroups {
        defaultSettings()
    }
}
```

You can also create the cluster node by specifying each setting individually:

```
clusterSettings {
    id '0'
    maxId '2'
    jGroups {
        tcpAddress '10.0.0.1'
        tcpPort 6677
        channelName 'my-custom-channel'
        configFile '/Users/foo/my-custom-config.xml'
```

```
    }
}
```

UDP Multicast Configuration

To configure UDP multicast enabled-node, use the following code:

```
clusterSettings {
    id '0'
    maxId '2'
    udpMulticast()
}
```

The code configures a cluster node with ID=0, max ID=2, and the following default UDP multicast communication settings:

- address: 224.0.0.0
- port: 9997

You can configure the cluster node, using a more verbose code:

```
clusterSettings {
    id '0'
    maxId '2'
    udpMulticast {
        standardSettings()
    }
}
```

You can also create the cluster node by specifying each setting individually:

```
clusterSettings {
    id '0'
    maxId '2'
    udpMulticast {
        address '224.0.0.1'
        port 9999
        networkInterface 'en0'
        debug true
    }
}
```

UDP Unicast Configuration

To configure UDP unicast-enabled node, use the following code:

```
clusterSettings {
    id '0'
    maxId '2'
    udpUnicast {
        address '10.0.0.55'
        port 9999
        clusterNodes '10.0.0.56:9999', '10.0.0.57:9999'
```

```
        syncNodesInterval 1
        debug true
    }
}
```

Configuring Database Connections

To provide database connection parameters, use the `dbSetup` method. For more information, see the *dbSetup Method* section in HybrisPlatform API.

Configuring the Media Storage Setup

To configure media storage settings, use the `mediaStorageSettings` method. For more information, see the *mediaStorageSettings Method* section in HybrisPlatform API.

You can also set up local file media storage for all cluster nodes using the `sysTempMediaStorage()` method. It stores all media in the `medias` subfolder of the system `temp` directory.

Managing Tomcat

The Platform Plugin provides a simple API to manage Tomcat.

The following code excerpt shows how to deploy the DataHub Tomcat instance:

```
def CATALINA_OPTS = "-Xms4096m -Xmx4096m ........ "
tomcat.instance('dataHub').setup {

    ports {
        http 9033
        ssl  9034
    }

    webApps {
        webApp 'datahub-webapp.war', file(suiteHome + '/hybris/bin/ext-integration/datahub/web-app/
    }

    libraries {
        lib file("/Volumes/Data/projects/hybris_git/y/bin/platform/lib/dbdriver/mysql-connector-jav

        propertyFile "local.properties", {
            property 'dataSource.driverClass', 'com.mysql.jdbc.Driver'
            property 'dataSource.jdbcUrl', "jdbc:mysql://localhost/integration?useConfigs=maxPerfor
            property 'dataSource.username', 'root'
            property 'dataSource.password', ''
        }
    }
}.start(CATALINA_OPTS)
```

The code example does the following:

- Creates a directory named `dataHub` in the `INSTALLER_HOME/work/` directory.
- Creates the required Tomcat configuration files, such as configuring the ports, in the `dataHub/conf` directory.

- Copies the specified `war` file to `dataHub/webapps/datahub-webapp.war` file.
- Copies the MySQL driver from the specified absolute path to the `dataHub/libs` directory.
- Creates the `local.properties` file, which contains the default properties, in the `dataHub/lib` directory.
- Starts the Tomcat with the provided *<CATALINA_OPTS>*. All log files are stored in the `dataHub/logs` directory.

Description of the Deploy DataHub Code

Below you can find detailed information about the Deploy DataHub Code.

Instance Name

Many Tomcat instances can be created and run using this API. To distinguish between the Tomcat instances, use the name of the instance as shown in the following example:

```
tomcat.instance('dataHub')
```

Each Tomcat instance must use a different port. If you have more than one Tomcat instance, make sure that the instances don't collide.

Tomcat Ports

The **ports** elements is used to specify which port a Tomcat instance is to use. The ports element has two parameters:

- *<http>*
- *<ssl>*

Certificates

If you need to provide a custom ssl certificate to handle https traffic, you can use the `certificates` element.

```
certificates {
    keyStore Paths.get("/my/custom/keystore"), "123456"
}
```

The `certificates` element also allows you to configure the trust store used by the jvm:

```
certificates {
    trustStore Paths.get("/my/custom/truststore"), "123456"
}
```

You can change the trust store and the key store in a single `certificates` element:

```
certificates {
    keyStore Paths.get("/my/custom/keystore"), "123456"
    trustStore Paths.get("/my/custom/truststore"), "123456"
}
```

Web Applications

You can deploy multiple web applications on one Tomcat instance. Use the**webApps** element to list the web applications. Define each web application using the following syntax:

Specifying web applications

```
webApp targetWarName, warFile
```

where:

- *<targetWarName>* is a string that specifies the target file name of the web application in the `tomcat/webapps` directory.
- *<warFile>* points to the `war` file. Note that you can compose the path to the file using variables exposed by the Platform Plugin, such as *<suiteHome>*.

Libraries

You use the **libraries** element to specify the content of the `tomcat/lib` folder. You can add files to the `tomcat/lib` folder using the following syntax:

tomcat/lib folder

```
lib file("/some/absolute/path/to/mysql-connector-java-5.1.26.jar")
```

You can also create property files in `tomcat/lib` using the following syntax:

```
propertyFile "local.properties", {
    property 'key', 'value'
}
```

Starting and Stopping Tomcat

The `Tomcat` object has the following methods:

- `start(String tomcatOpts)`: Starts the Tomcat instance with the options provided.
- `stop`: Stops the Tomcat instance.

Managing the HSQLDB

The `HybrisHsqldb` object provides methods to configure the HSQLDB and run it as a standalone process. Unlike the SAP Commerce default database, the embedded HSQLDB, which accepts only one connection, the `HybrisHsqldb` object accepts many connections. It can therefore be used in a cluster environment.

HSQLDB Port

By default, the HSQLDB runs on port 9009. You can use the optional `port` method to change default values, as follows:

```
hsqldb.port 49302
...
hsqldb.startInBackground()
```

ℹ Note

This is an optional step. If you want to change the port on which the HSQLDB runs, you must do it before starting or stopping the database.

Configuring the Database

HSQLDB supports a maximum of nine (9) separate databases running on a single process. Use the *<dbNames>* property in the `HybrisHsqldb` object to configure the names of the databases as shown in the following example:

Configuring database names

```
hsqldb.dbNames = ['foo', 'bar']
...
hsqldb.startInBackground()
```

The following code shows how to configure the HSQLDB to host two databases: foo and bar. The detailed configuration is displayed during startup.

```
Hsqldb successfully started:
        foo             jdbc:hsqldb:hsql://localhost:9009/foo
        bar     jdbc:hsqldb:hsql://localhost:9009/bar
```

ⓘ Note

Make sure to configure the database **before** calling the `startInBackground()` method.

Configuring the Database Using the HybrisPlatform Object

You can configure the HSQLDB using the `HybrisPlatform` API. You can use the `createDb` method to automatically configure the database as a Platform object. The following code shows how to do it:

Configuring Database using the HybrisPlatform Object

```
def hsqldbPlatform = platform {
    dbSetup hsqldb.createDb("hybris")
    // or simply
    dbSetup hsqldb.createDb()
}
```

In the provided example:

- `createDb()`: Adds new database named *<db_N>* (where *<N>* is a number of the currently created database) and returns the configuration for *<dbSetup>* parameter.

  Note that this throws a runtime exception when the database limit of nine (9) is exceeded.

- `createDb(String databaseName)`: Adds a new database with the specified name and returns the configuration for the *<dbSetup>* parameter.

  ### ⓘ Note

  Note that this throws a runtime exception when the database limit of nine (9) is exceeded or when a database with the specified name already exists.

Starting and Stopping The HSQLDB

The `HSQLDB` object has the following methods:

- `startInBackground`: Starts an HSQLDB server instance on a configured port.

- `stopInBackground`: Stops a currently running HSQLDB server instance.
- `kill`: Kills a currently running HSQLDB process and clears the database work directory.

> **Note**
>
> Use the `kill` method cautiously because it kills processes abruptly and deletes all persisted and temporary data.

Detailed Information about Starting and Stopping the HSQLDB

During startup, the HSQLDB looks for a configuration file in its `work` directory. If a `pid.txt` file is present, the HSQLDB tries to connect to the URLs specified in the file and has the possible results:

- If successful, the HSQLDB assumes that the database was created previously; it stores some persistent data and becomes operational again.
- If not successful: if the HSQLDB cannot connect, it assumes that the database was stopped in an unsafe way and may be corrupted. The `HybrisHsqldb` object will try to kill the HSQLDBprocess that is currently running, clear the database `work` directory, and start the HSQLDB in a clean state.

Related Information

Installing SAP Commerce Manually
Environment Variables

Internationalization and Localization

Internationalization and localization are intended to adapt SAP Commerce to multiple languages and different requirements dependent on a region.

The ServiceLayer contains the **i18N** package with services that provide internationalization and localization support for SAP Commerce. In particular it:

- Allows developers to use Java I18N framework by providing SAP Commerce locale and timezone values, not necessarily the same as for Java Virtual Machine (JVM)
- Provides access to **CustomI18NService** supporting Platform domain objects
- Contains **L10NService** that returns localized resource bundles and values of localized attributes displayed in front-end applications.

Introduction to Internationalization and Localization

Internationalization in SAP Commerce is language- and country-specific logic that is independent of the operating system. It is related to:

- Java Platform objects like **Locale, Currency,** or **TimeZone**
- SAP Commerce domain objects like **LanguageModel, CurrencyModel** or **CountryModel**

Localization in SAP Commerce is the logic used to fetch a locale specific resource, that is **String** or **ResourceBundle**, which are used to present region specific content to the outside world.

Localization and Internationalization in SAP Commerce are not only limited to switching between different languages, but there are more aspects:

- Language-specific values: For every individual language, it is possible to specify a value for:
  - Type-system-related attributes
  - Classification attributes
- Localization settings: For users in web front ends Language enables switching between different locale settings. The screen texts are displayed in different localizations, depending on the language.
- Limited range of available languages for user groups

Locale is a set of settings related to the language and region, in which a computer program executes.

Internationalization Features

Java provides a **Locale** object as a representation of a specific geographical, political, or cultural region. For example, displaying a number is a locale-sensitive operation, because the number should be formatted according to the conventions of the user's native country or region. When you create a custom **Locale**, no validity check is performed. If you would like to check whether particular resources are available for the **Locale** you construct, you should query those resources. For example, ask the **NumberFormat** for the locales it supports using its **getAvailableLocales()** method.

For more information, see http://download.oracle.com/javase/6/docs/api/index.html : Java API documentation.

Java Objects vs. SAP Commerce Models

Java objects like **Locale** and **Currency** depend on Java Virtual Machine (JVM). The Platform does not contain one direct type corresponding to Java **Locale**. However, it provides SAP Commerce specific objects like **LanguageModel**, **CountryModel**, and **RegionModel** that may be used to create a specific locale.

Currency Java object represents an existing currency defined by ISO 4217 currency codes, whereas **CurrencyModel** can contain any currency configuration.

Java **Locale** can be transformed into **LanguageModel**. No validation is performed in this direction, so any existing **Locale** in the system can be stored as **LanguageModel**. Getting a language for a default **Locale** can be done in the following way:

```
LanguageModel lanmodel = commonI18NService.getLanguage(Locale.getDefault().getLanguage());
```

If no **LanguageModel** exists for the given locale, the **UnknownIdentifierException** is thrown.

A similar situation exists for Java **Currency** representing an existing currency in a system, which can be transformed into **CurrencyModel**:

```
CurrencyModel cmodel = commonI18NService.getCurrency(javaCurrency.getCurrencyCode() );
cmodel.setSymbol(javaCurrency.getSymbol());
```

The transformation in the opposite direction is also possible. **LanguageModel** can be transformed into **Locale**:

```
LanguageModel lanModel ....

Locale someLocale = commonI18NService.getLocaleForLanguage(lanModel);
```

During a transformation of **CurrencyModel** into Java **Currency** a validation is performed on Java Virtual Machine level. It is checked if **currencyCode** is supported by ISO 4217. If not, **Null** is returned. You can get **Currency** instance for the **CurrencyModel**:

```
Currency someCurrency = i18nService.getBestMatchingCurrency(currencyModel.getIsocode());
```

To distinguish between different object types, the internationalization is split into two services: **I18NService** and **CommonI18NService**.

The **I18NService** operates only on Java classes. It is related to Platform specific context, like tenant or session context, and their localization configuration, like language, country, currency. What is more, it is able to influence the Platform with that, for example by setting language for the current session. It also provides lists of plain Java types, related to i18n, supported by the Platform, like all supported **Currency** or **Locale** objects. Moreover, it enables to verify if a certain type instance, related to i18n, is supported by the Platform.

Sample code presenting how to get all supported **Locales**:

```
for(Locale locale : i18nService.getSupportedLocales())
{
    LOG.info( "hybris Suit supports country "+ locale.getCountry());
}
```

The **CommonI18NService** operates on the Platform domain objects and is used only for fetching SAP Commerce Models for a current context of the usage. For example, you may add new **LanguageModel**.

Localized Attributes

Localized attributes in SAP Commerce are attributes that can have different values for each language in the system. For example, the name of a unit might be different for English and German. Attributes are defined to be localized by a specific keyword in the attribute definition in the ***extension_name*** **`-items.xml`** file.

Attribute name	Localized
Identifier	No
Name	Yes
Conversion	No
Type	No

The idea of localized attributes is to store a map in Models containing a value for a specific attribute for every locale supported by the Platform. As a result, you can reach a proper value for a current localization settings. If there is no value for the current localization, the Language Fallback mechanism described below can be used.

If a type of Model that you are working on, you may directly set localized values for its attributes, for example for **ProductModel**:

```
//search for a product where the english name is "uniqueName_english" and the german name "uniqueNa
ProductModel exampleProduct = modelService.create(ProductModel.class);
//do some localized intialization
exampleProduct.setName("some_deutsch_name", Locale.GERMAN);
exampleProduct.setName("some_english_name", Locale.ENGLISH);
....
//save the model
modelService.save(exampleProduct);
```

Language Fallback

This language fallback mechanism is responsible for providing a content for a localized attributes if no value is provided for the current localization settings. It can be activated by **setLocalizationFallbackEnabled** method from **I18NService**. The example below shows how to create a fallback configuration and use it:

```
final LanguageModel firstLanguage = modelService.create(LanguageModel.class);
                firstLanguage.setIsocode("first");

                final LanguageModel secondLanguage = modelService.create(LanguageModel.class);
                secondLanguage.setIsocode("second");
                //set a fallback language
                secondLanguage.setFallbackLanguages(Arrays.asList(firstLanguage));
                modelService.saveAll();

                final ProductModel product = modelService.create(ProductModel.class);
                product.setCode("sampleProduct");
                product.setCatalogVersion(version);
                //set product description value in the first language
                product.setDescription("some value in first ", new Locale(firstLanguage.getIsocode
                //no value for the second language

                modelService.saveAll();
                //display product description in the first language
                i18nService.setCurrentLocale(new Locale(firstLanguage.getIsocode()));
                modelService.refresh(product);
                System.out.println(product.getDescription());
                //display product description in the second language
                i18nService.setCurrentLocale(new Locale(secondLanguage.getIsocode()));
                modelService.refresh(product);
                System.out.println(product.getDescription());
```

The output result in this case is as follows:

```
some value in first
some value in first
```

If the fallback mechanism is disabled, the result is:

```
some value in first
null
```

Context Settings

Depending on settings related to the current session, like user currency, time zone or locale, logic applied for the internationalization is customized.

ℹ Note

Find Out More

In order to pass a language or currency different from the current session settings, the logic should be called in a local session context, bound to the current thread. For this you should pass an own **SessionExecutionBody** to **sessionService.executeInLocalView**, which provides a thread bound copy of current session context for the body execution.

- Users can have various restrictions, what results in different supported locals. It means that for different users, different sets of available languages can be available.

  An example returning supported locales for **admin** user:

```
sessionService.executeInLocalView(new SessionExecutionBody()
{
    public Object execute() {
        sessionService.setAttribute("user",adminUserModel);
        return i18nService.getSupportedLocales();
    }
});
```

- Current **Locale** settings can affect values returned for a localized attribute.

  An example returning Chinese name for an **Online** catalog:

```
sessionService.executeInLocalView(new SessionExecutionBody()
{
    public Object execute() {
        i18nService.setCurrentLocale(chineseLocale);
        CatalogModel cat =  defaultCatalogService.getCatalog("Online");
        return cat.getName();
    }
});
```

- **Locale** or **Currency** for the current session can also affect returning format objects like **NumberFormat** and **DateFormat** from **FormatFactory**.

  An example presenting how to get a specific currency format with a custom locale and currency:

```
sessionService.executeInLocalView(new SessionExecutionBody()
{
    public Object execute() {
        i18nService.setCurrentLocale(customLocale);
        i18nService.setCurrentCurrency(customCurrency);
        return formatFactory.createCurrencyFormat();
    }
});
```

- **TimeZone** settings can affect returning format objects like **DateFormat** and **DateTimeFormat** from **FormatFactory**.

  An example presenting how to get a specific **DateFormat** with custom locale and time zone:

```
sessionService.executeInLocalView(new SessionExecutionBody()
{
    public Object execute() {
        i18nService.setCurrentLocale(customLocale);
        i18nService.setCurrentTimeZone(customTimeZone);
        return formatFactory.createDateTimeFormat(DateFormat.FULL,-1);
    }
});
```

Tenant context settings like locale and time zone, are used when the current session settings are not given. To provide a tenant-specific locale and time zone, use the SAP Commerce Administration Console:

If the session settings and tenant settings are given, the settings from the JVM are used. For more information about tenants see Multi-Tenant Systems, **Display List of Tenants** section.

Internationalization and Localization provides also **FormatFactory** for getting formats with context dependent settings. Having a session specific locale, currency and timezone, SAP Commerce provides a range of standard methods to fetch number, currency and date formatters. When retrieving or creating item data, these methods should be always preferred to standard Java formatter methods. Java methods use standard locale and time zone settings of the Virtual Machine if not explicitly specified. Each session may use different locale or time zone. It may lead to incorrect presentation or even wrong data stored in a database, for example when parsing date fields using the wrong time zone.

The **FormatFactory** provides the most common Format objects like:

- **NumberFormat** for formatting currency, number, percent
- **DateFormat** for date and time

> **Note**
>
> **Find Out More**

createCurrencyFormat method gets the information from the current session. In order to create a format for other than current session settings, the logic should be called in the local session context:

```
sessionService.executeInLocalView(new SessionExecutionBody()
{
    public Object execute() {
        i18nService.setCurrentLocale(customLocale);
        i18nService.setCurrentCurrency(customCurrency);
        return formatFactory.createCurrencyFormat();
    }
});
```

The example below presents how to work with **SimpleDateFormat** and **DecimalFormat**:

```
SimpleDateFormat format = (SimpleDateFormat)factory.createDateTimeFormat(DateFormat.DEFAULT,
format.applyPattern("yyyy-mm-dd");

DecimalFormat decimalFormat =(DecimalFormat) factory.createNumberFormat();
decimalFormat.applyPattern("some pattern");
```

Localization Features

To access the localized resource bundles, use `L10NService`. It covers the following functionality:

- Accessing the resource bundles and related resources
- Supporting SAP Commerce locale fallback mechanism
- Provides localized values of types and their attributes in front-end applications

It does not provide a front-end specific localization.

Type System Localization

Property files containing type system localization reside in **`/resources/localization`** directory in an extension. Every language in SAP Commerce can have individual localizations and, by a consequence, individual property files.

Name of the localization properties file follows the pattern: ***extensionname*** **`-locales_`** ***isocode*** **`.properties`**, where ***isocode*** is the identifier of a language in SAP Commerce. For example: **`core-locales_en.properties`** or **`category-locales_de_at.properties`**.

Be aware that the delimiter between filename parts is the underscore (_). Depending on the application server, other delimiters like - or . may be ignored.

ℹ Note

Renaming System Languages at Run Time Breaks Localizations

To allocate localization strings, SAP Commerce uses identifiers of languages, that is **isocode** attribute. You should make sure that the identifier of a language remains constant. If the value of the **isocode** attribute is modified at run time and the system is updated, the allocation between localization file name and language **isocode** will be broken. By consequence, localization strings will not be correctly assigned during system initialization or update. Names of localization property files should be adapted accordingly.

The basic structure of a localization property file looks like this:

```
type.{typecode}.name=value
type.{typecode}.description=value
```

```
type.{typecode}.{attributedescriptor}.name=value
type.{typecode}.{attributedescriptor}.description=value

type.customerorderoverview.name=Order statistics
type.customerorderoverview.description=Statistic of order volume of each single customer
```

For example, the line **type.tutorial.partofproduct.name=partOf Product** navigates downwards through the type system hierarchy like this: **ComposedType type ->subtype tutorial ->attributedescriptor partofproduct ->localizedStringAttribute name** and sets **partOf Product** value to the **localizedString**. It means that localizations for items and their attributes are related to actual objects in the database. The following code snippet shows a sample of a **`locales_en`** file.

```
type.tutorial.longtext.name=Long text
type.tutorial.longtext.description=This element may contain text that is quite long.
type.tutorial.name=Name
type.tutorial.partofproduct.name=partOf Product
type.tutorial.picture.name=Picture
type.tutorial.picture.description=Via this picture, you may better visualize the product.
type.tutorial.short.name=Short
```

ℹ Note

Update or Initialize after Changing Localization Files

After changes made in localization files, call **`ant`** in the extension directory. Afterwards, update or initialize SAP Commerce. Make sure that at least the **Localize types** check box is selected.

For more information, see the *Localize Types* section of Updating from the SAP Commerce Administration Console.

Influence of the Language Fallback

Language Fallback mechanism described above, affects the localization in that way that :

- Requesting the **ResourceBundle** returns some merged resource bundle for the current language and its fallback languages.
- When requesting some specific label value, the API looks for this property in the default **ResourceBundle** for the language as for all fallback languages, and returns the first not empty value for any of those bundles.

SAP Commerce provides a convenient way of defining a list of fallback languages for each system language. If enabled, this feature replaces missing values of localized attributes by possibly existing value for one of the specified fallback languages. For example, if **de_DE** has got **de, en** as fallback languages, a missing value for **de_DE** is substituted by the one for **de** or, if also empty, by the one for **en**.

Unfortunately the standard way of localization using **`java.util.ResourceBundle`** cannot obey these fallback rules. Since it is completely based upon **`java.util.Locale`**, the fallback mechanism for **de_DE** is always **de, <vm startup language>** regardless what has been specified within the SAP Commerce system.

This way you could end up with a confusing language fallback behavior. For example, storefront is currently showing data for the session language **es**, the fallback language has been set to **de**, the Java VM has been started with **en** as system locale. Missing localized item attributes are substituted with existing **de** values as intended, but missing resource bundle values are substituted by existing **en** values, because **`java.util.ResourceBundle`** will always fall back to the system locale as a last resort.

Accessing Localized Strings and Resource Bundle Using L10NService

Code snippet for fetching the bundle for a specific locale:

```
i18nService.setCurrentLocale(Locale.ENGLISH);
          ResourceBundle bundle = l10nService.getResourceBundle("servicelayer.test.testBundle");
```

```
String value = L10NService.getLocalizedString("some.key");
```

> **Note**
> Fallback for names of attributes is disabled by default.

Related Information

Localization
Jalo-Based Locale and Timezone Handling
Languages and Localization
Internationalization (I18N) Support in the ServiceLayer

Internationalization (I18N) Support in the ServiceLayer

The ServiceLayer contains a service called I18NService that provides internationalization support for SAP Commerce.

> **Note**
> As this document refers to Jalo, find relevant information in Internationalization and Localization Overview.

In particular it:

- Allows developers to use Java's rich I18N framework by providing SAP Commerce locale and timezone values (not necessarily the JVM locale and timezone)
- Provides access to SAP Commerce own custom I18N support
- Allows backwards compatibility to SAP Commerce earlier I18N framework.

Using Java's I18N Framework

To use Java's I18N framework, you need access to the application's current Locale. By default the Locale returned to the developer via `java.util.Local.defaultLocale` is the JVM's own Locale which could well be different to the Locale being used by SAP Commerce.

Therefore, call the `I18NService` 's `getCurrentLocale()` method (API Doc) to retrieve the Locale currently being used by SAP Commerce. Similarly, you should call `I18NService` 's `getCurrentTimeZone()` method to retrieve the SAP Commerce current timezone.

Subset of methods in `I18NService.java`:

```
//Get hybris' locale (and not JVM's locale), so you can use Java's standard I18N framework
Locale getCurrentLocale();
void setCurrentLocale(Locale loc);

// Get hybris' timezone (and not JVM's timezone), so you can use Java's standard I18N framework
TimeZone getCurrentTimeZone();
void setCurrentTimeZone(TimeZone loc);
```

With these in hand, you can make full use of Java's I18N API such as using `MessageFormat`, `NumberFormat`, `DateFormat` and `Calendar`. These methods allow us to internationalise strings, dates and numbers, but they do not address localising SAP Commerce business objects themselves (such as `Product`, `Catalog`, `CatalogVersion`). For that you need to look at SAP Commerce own I18N framework.

SAP Commerce I18N Framework

There are two areas of SAP Commerce I18N framework discussed below:

- hybris' resourceBundle Framework
- hybris' ServiceLayer Language Fallback

SAP Commerce resourceBundle Framework

SAP Commerce own I18N framework is based around a class named `CompositeResourceBundle` which is an extension of Java's `java.util.ResourceBundle` class. Before describing SAP Commerce variant, let us consider the behavior of Java's own `ResourceBundle` below (see http://java.sun.com/docs/books/tutorial/i18n/resbundle/propfile.html for further discussion).In the SAP Commerce test suite, there are test property files for testing and contrasting the ResourceBundle and CompositeResourceBundle logic:

- `testBundle_en_US.properties` containing the text

```
action.testFallback1=US
```

- `testBundle_en.properties` containing the text

```
action.testFallback1=En
action.testFallback2=En
```

- `testBundle_de.properties` containing the text

```
action.testFallback1=De
action.testFallback2=De
action.testFallback3=De
```

Let us instantiate a `ResourceBundle` for these files, passing in the locale `Locale.US`:

```
final ResourceBundle javaBundle =
   ResourceBundle.getBundle("testBundle", Locale.US, getClass().getClassLoader());
```

ResourceBundle Behavior

Here, you can expect the usual `ResourceBundle` behavior when using the javaBundle instance as detailed in the unit test code below:

```
//key available in testBundle_en_US.properties
assertEquals("US", javaBundle.getObject("action.testFallback1").toString());
//key not available in testBundle_en_US.properties but is available in testBundle_en.properties
assertEquals("En", javaBundle.getObject("action.testFallback2").toString());
try{
   //key not available in testBundle_en_US.properties or testBundle_en.properties
   javaBundle.getObject("action.testFallback3").toString();
   fail("Should throw MissingResourceException");
}
catch (final MissingResourceException e){
}
```

This follows the usual resourceBundle logic when requesting a localized version of a string:

- Look for the request key in the properties file `testBundle_en_US.properties`
- If the key is not found, try the properties file `testBundle_en.properties`

- If the key is not found, throw a `MissingResourceException`

CompositeResourceBundle Behavior

Let us now contrast that with how `CompositeResourceBundle` behaves. You can instantiate a `CompositeResourceBundle` as follows:

```
final ResourceBundle hybrisBundle = i18nService.getBundle("testBundle",
    new Locale[] { Locale.US, Locale.GERMAN }, getClass().getClassLoader()
);
```

The second parameter consists of a list of Locales, which describe the fallback logic particular to `CompositeResourceBundle`. What this means is that the logic now followed when accessing a string in the properties files becomes:

1. Look for the request key in file `testBundle_en_US.properties`
2. If the key is not found, try the file `testBundle_en.properties`.
3. If the key is not found, use the next Locale in your fallback list (in this case `Locale.GERMAN`) and therefore look in `testBundle_de.properties`
4. If the key is not found, throw a `MissingResourceException`.

This is illustrated in the following unit test code:

```
//key available in testBundle_en_US.properties
assertEquals("US", hybrisBundle.getObject("action.testFallback1").toString());
//key not available in testBundle_en_US.properties but is available in testBundle_en.properties
assertEquals("En", hybrisBundle.getObject("action.testFallback2").toString());
//key not available in testBundle_en_US.properties or testBundle_en.properties,
// but is available in testBundle_de.properties
assertEquals("De", hybrisBundle.getObject("action.testFallback3").toString());
try{
 //key not available anywhere
 hybrisBundle.getObject("action.testFallback4").toString();
 fail("Should throw MissingResourceException");
}
catch (final MissingResourceException e){
}
```

You can see the complete unit test code in the method `testCompositeResourceBundle2()` in file `I18NServiceTest.java`.

SAP Commerce ServiceLayer Language Fallback

SAP Commerce provides a language fallback mechanism when using the ServiceLayer to query models for their localized names. The I18NService interface includes the following method where you can enable or disable this feature:

```
void setLocalizationFallbackEnabled(boolean enabled);
```

To understand how this works, let us consider what happens if the LocalizationFallback feature is disabled.

Create a new language `aNewLanguage` and let us set its fallback language to the SAP Commerce default language. Your models will have no names that are localized to `aNewLanguage` as you have just created it, but they will have names localized to SAP Commerce default language.

```
final Language defaultLanguage = jaloSession.getSessionContext().getLanguage(); // hybris default l
final Language aNewLanguage = C2LManager.getInstance().createLanguage( "aNewLanguage" ); // a new l
```

```
aNewLanguage.setFallbackLanguages( Collections.singletonList( defaultLanguage ));
// sets the new language's fallback to hybris default language
```

With the localization fallback feature disabled, if you now ask the model for its name localized to `aNewLanguage` you should see the value null returned because the model `aProductModel` has no name that is localized to `aNewLanguage`.

```
i18nService.setLocalizationFallbackEnabled(false); // disable the fall back feature
assertNull(aProductModel.getName( aNewLanguage.getLocale() ) ); // query aProductModel for the loca
```

Now let us enable the localization fallback feature:

```
i18nService.setLocalizationFallbackEnabled(true);
```

When you again query `aProductModel` for its name localized to `aNewLanguage`, you will receive the name that is localized to SAP Commerce default language, as the framework has "fallen back" to `aNewLanguage`'s fallback language, which in this case is SAP Commerce default language.

```
assertNotNull(aProductModel.getName(aNewLanguage.getLocale()));
assertEquals(aProductModel.getName(), product.getName(aNewLanguage.getLocale()));
```

You can see the complete unit test code in the method `testLanguageFallback2()` in file `ItemModelTest.java`.

Backward Compatibility to SAP Commerce Earlier Jalo-Based I18N Logic

Before the I18nService was introduced into the ServiceLayer, developers had been using models such as LanguageModel, CurrencyModel and CountryModel for their I18N requirements. These are still supported, and can be accessed with the following methods in i18nservice.

Subset of methods in I18nService.java:

```
// For backwards compbatibility, provide access to the LanguageModel
LanguageModel getLanguage(String isocode);
Set getAllLanguages();
Set getAllActiveLanguages();

// For backwards compatibility, provide access to the CountryModel
CountryModel getCountry(String isocode);
Set getAllCountries();

// For backwards compbatibility, provide access to the CurrencyModel
CurrencyModel getCurrency(String isoCode);
Set getAllCurrencies();
CurrencyModel getBaseCurrency();
CurrencyModel getCurrentCurrency();
void setCurrentCurrency(CurrencyModel curr);
```

Jalo-Based Locale and Timezone Handling

The SAP Commerce Jalo layer is deprecated. Please refer to other localization topics to learn how to handle time zones.

⚠ Caution

This page refers to deprecated software. For further details, see Deprecation Status.

One of the most common tasks in SAP Commerce application development is rendering numbers, monetary amounts, and timestamps. Platform can run in multi-tenant mode, where tenant-specific locale and timezone handling is useful. Without this support, each web application would have to determine which locale and timezone would be appropriate for the current tenant, and create its own number, currency, and date formatters.

Session Provides Locale and Timezone

Each `JaloSession` carries a `Locale` and `TimeZone` directly inside the `SessionContext`. This way you can change it manually if necessary without affecting the session language item. Determining the locale upon session creation works as follows:

1. Determine the session language.
    - If the user provides a language, use it.
    - If not, try to find language matching the (new) tenant default locale.
    - Otherwise, fall back to the first active language.
2. Try to get the locale from the language.
3. If that fails, then use the tenant-specific locale.
4. If no tenant-specific locale is set, use the Java VM default locale.

The current tenant may also specify a default timezone. This default is used as a session timezone upon session creation. If no tenant-specific timezone has been set, then the Platform default is used. By holding a locale and timezone for each session, you can avoid having to implement your own locale and timezone guessing logic in the business logic. The session is an always present session container for you to fetch the locale and timezone from.

However, the most important reason for holding locale and timezone in the session is that Platform contains a number of dedicated methods in the `Utilities` class that provide number, currency and date formatter instances according to the current session settings.

Tenant Specific Locale and Timezone

Each tenant that is not a master tenant can specify a custom locale and timezone when it is created. If nothing is specified, then the tenant uses the Java VM default settings for locale and timezone. This way multiple tenants may be hosted together using their locale and timezone settings even if they're sharing web applications such as Backoffice.

Affected Modules

This feature affects many SAP Commerce modules.

ImpEx Import and Export

The default locale is always the locale of the current tenant.

Please keep in mind that ImpEx files working well for one tenant may break for another unless the locale is set explicitely inside the file as shown in the following example.

```
#% impex.setLocale( java.util.Locale.GERMANY );

                    INSERT Product; ... ; europe1Prices[translator=...]
                    ; ... ; 12,50 EUR

                    #% impex.setLocale( java.util.Locale.EN );

                    INSERT Product; ... ; offlineDate ; europe1Prices[translator=...]
                    ; ... ; 12.50 EUR
```

This works best when the CSV file is not used as an ImpEx file, but included within a separate ImpEx script.

Standard Methods for Getting Number, Currency and Date Formatters

SAP Commerce provides a range of standard methods to fetch **number**, **currency** and **date** formatters in the class `de.hybris.platform.util.Utilities`.

When reading or writing item data these methods should be always be preferred to standard Java formatter methods because Java methods will use the VM standard locale and timezone settings wherever not specified explicitly. As each session may use a different locale or timezone, this may lead to the wrong presentation or even the wrong data written to the database. when parsing date fields using the wrong timezone, for example.

- Shortcut for getting the locale from current session

method	returns	prefer to
`getDefaultLocale()`	Locale	`Locale.getDefault()`

- Creates a several number format instances using the session locale

method	returns	prefer to
`getNumberInstance()`	NumberFormat	`NumberFormat.getNumberInstance()`
`getIntegerInstance()`	NumberFormat	`NumberFormat.getIntegerInstance()`
`getPercentInstance()`	NumberFormat	`NumberFormat.getPercentInstance()`

- Creates a decimal format from pattern using the session locale

method	returns	prefer to
`getDecimalFormat(String)`	DecimalFormat	`new DecimalFormat(String)`

- Creates currency number format instances using the session locale **and** (session) currency;

  The returned number format will be initialized with the currency sign and digits from the currency item - the java vm defaults are overwritten here.

method	returns	prefer to
`getCurrencyInstance()`	NumberFormat	`NumberFormat.getCurrencyInstance()`
`getCurrencyInstance(Currency)`	NumberFormat	

- Creates calendar instances using the session timezone and (session) locale

method	returns	prefer to
`getDefaultCalendar()`	Calendar	`Calendar.getInstance()`
`getDefaultCalendar(Locale)`	Calendar	`Calendar.getInstance(Locale)`

- Creates SimpleDateFormat instances from pattern using the (session) locale and calendar (see above)

method	returns	prefer to
`getSimpleDateFormat(String)`	SimpleDateFormat	`new SimpleDateFormat(String)`
`getSimpleDateFormat(String,Locale)`	SimpleDateFormat	`new SimpleDateFormat(String)`

- Creates date (-time) format instances using the session timezone, (session) locale and optionally specified style flags

method	returns	prefer to
`getDateTimeInstance()`	DateFormat	`DateFormat.getDateTimeInstance()`
`getDateTimeInstance(int,int)`	DateFormat	`DateFormat.getDateTimeInstance(int,int)`
`getDateTimeInstance(int,int,Locale)`	DateFormat	`DateFormat.getDateTimeInstance(int,int,Locale)`
`getDateInstance()`	DateFormat	`DateFormat.getDateInstance()`
`getDateInstance(int,Locale)`	DateFormat	`DateFormat.getDateInstance(int,Locale)`

Language Fallback Rules versus Resource Bundles

SAP Commerce provides a convenient way of defining a list of fallback languages for each system language. If enabled this feature replaces missing localized attribute values with an existing value for one of the specified fallback languages. For example, if `de_DE` has got [`de`, `en`] as fallback languages, a missing value for `de_DE` would be substituted by the one for `de` or, if also empty, by the one for `en`. Unfortunately the standard way of localization via `java.util.ResourceBundle` cannot obey these fallback rules. Since it is completely based upon `java.uil.Locale` the fallback mechanism for `de_DE` is always [`de`, <vm startup language>] no matter what has been specified within the SAP Commerce system.

This way you could end up with confusing language fallback behavior like this: a storefront is currently showing data for the session language `es`; the fallback language has been set to `de`; the java vm has been started with `en` as system locale; now missing localized item attributes are substituted with existing `de` values (as intented) but missing resource bundle values are substitued by existing `en` values because `java.util.ResourceBundle` will always fall back to the system locale as a last resort.

Standard Methods for Obtaining Resource Bundles

SAP Commerce offers `de.hybris.platform.util.Utilities` methods for obtaining a resource bundle which does obey the SAP Commerce language fallback rules **and** the tenant specific default locale.

method	prefer to	remarks
`getResourceBundle(String)`	`ResourceBundle.getBundle(String)`	resource bundle for current session language/locale; to be

method	prefer to	remarks
		used inside a SAP Commerce extension class loader
`getResourceBundle(String,ClassLoader)`	`ResourceBundle.getBundle( String )`	resource bundle for current session language/locale; to be used if resource data is provided by a different class loader e.g. webapp class loader
`getResourceBundle(SessionContext,String)`	`ResourceBundle.getBundle( String,Locale )`	resource bundle for specified session language/locale; to be used inside a SAP Commerce extension class loader
`getResourceBundle( SessionContext, String, ClassLoader )`	`ResourceBundle.getBundle( String, Locale, ClassLoader )`	resource bundle for specified session language/locale; to be used if resource data is provided by a different class loader e.g. webapp class loader
`getResourceBundle( Language, String )`	`ResourceBundle.getBundle( String, Locale )`	resource bundle for specified session language/locale; to be used inside a SAP Commerce extension class loader
`getResourceBundle( Language, String, ClassLoader )`	`ResourceBundle.getBundle( String, Locale, ClassLoader )`	resource bundle for specified session language/locale; to be used if resource data is provided by a different class loader e.g. webapp class loader
`getResourceBundle( Locale[], String, ClassLoader )`	`ResourceBundle.getBundle( String, Locale, ClassLoader )`	resource bundle for a custom fallback list of locales

Languages and Localization

SAP Commerce supports multiple languages. This makes it easy to use SAP Commerce to run applications for different countries.

Languages in SAP Commerce

In addition to switching between display languages, language support in SAP Commerce also includes the following:

- **Language-specific values**: For every individual language, it is possible to specify a value for:
  - Type-system-related attributes
  - Classification attributes
- **Localization Setting:** A language is a means of switching between different locale settings. Screen texts are displayed in different localizations, depending on the language chosen.
- **Synchronization of catalog versions**
- **Allowed values for classification systems**

- **A limited range of available languages for user groups**

You can create any number of languages in SAP Commerce.

When specifying localization strings and localized attributes for web front ends, be aware that texts conveying the same message can be different in length across different languages. For example, texts in German tend to be longer than texts in English. By consequence, a fixed-size screen layout may fit for English, but end up broken for German. The effect of different text length applies to both the actual localization strings and the localized attribute values.

For a list of supported languages, see Supported Languages.

Localization

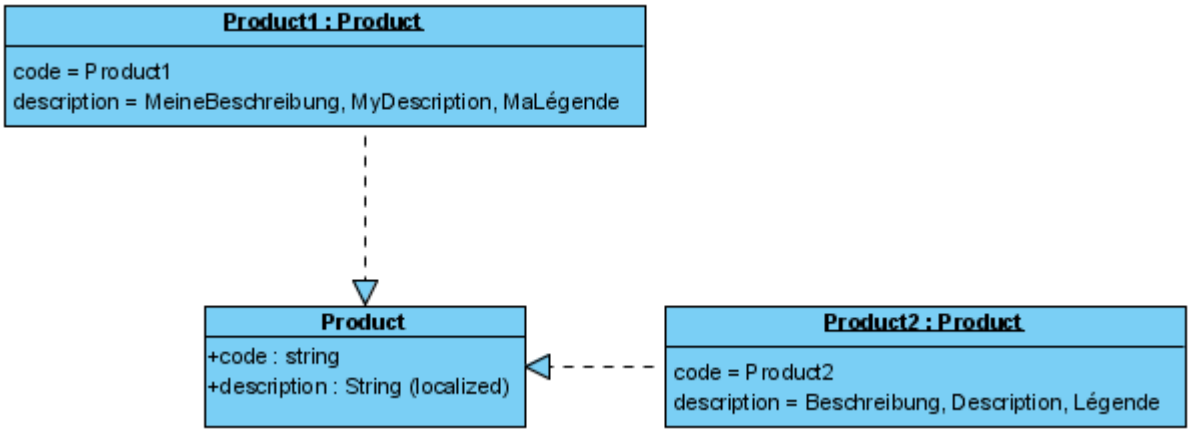
Basically, we need to make a difference between the actual localization mechanism on the one hand (which allows displaying language-specific attribute names, screen texts, and so on) and localized attributes on the other hand.

Localizations are displayed texts that are different for every individual language. This does not have anything to do with language-specific values.

This is caused by the technical way of how SAP Commerce handles attributes. An attribute is technically represented by an AttributeDescriptor, which can have an individual name per system language, and holds values.

- AttributeDescriptors for non-localized attributes can have a localized name and one individual value over all system languages. The name is set by the localization mechanism during type system generation, whereas the values are set by runtime.
- AttributeDescriptors for localized attributes can have a localized name and one individual value per system language. The name is set by the localization mechanism during type system generation, whereas the values are set by runtime.

In the following diagram, the `Product` type has two attributes: the non-localized attribute `code` and the localized attribute `description`.



Attribute	Attribute value for DE	Attribute value for EN	Attribute value for FR
Product1:code	(Product1)	(Product1)	(Product1)
Product2:code	(Product2)	(Product2)	(Product2)
Product1:description	Beschreibung	Description	Légende
Product2:description	MeineBeschreibung	MyDescription	MaLégende

For an in-depth discussion of the localization mechanism and how to set localization strings, see Internationalization and Localization.

Localized Attributes

Localized attributes in SAP Commerce are attributes that can have different values for each language in the system. For example, the name of a unit might be different for English, German, and French, as shown in the following screenshot. Attributes are defined to be localized via a specific key word in the attribute definition in the `items.xml` file. For more information, see items.xml.

Attribute name	Localized?
Identifier	no
Name	yes
Conversion	no

Where Do Languages Have Effect?

Languages in SAP Commerce take effect mostly in relation to user management, session management, and catalog access rights.

Even though related logically, from a technical point of view most of the currency and ordering-related concepts in SAP Commerce do not have a connection to languages by factory default.

Regions, countries, currencies, and addresses do not have a direct connection to languages. In other words: even if you only have English as a system language, your customers will still be able to order in EUR, USD, JPY, and set up addresses in Germany, France, and China. If you want to ensure that only inhabitants of certain countries will be able to use certain currencies or languages, you will need to implement the restricition explicitly.

Default Language for a User

A user (whether Customer or Employee) may select a default language optionally. If the user logs into SAP Commerce, the selected default language will be used until the user manually sets another language.

If a user does not select a default language explicitly, SAP Commerce uses the system default language. For more information, see Jalo-Based Locale and Timezone Handling.

On a more technical level, the language is set to the SessionContext of the JaloSession used. The default language of a user account is set to the SessionContext of the JaloSession when the JaloSession is created, and will remain the active language until changed explicitly.

- The basic mechanism of setting the language to the SessionContext:

```
Language de = C2LManager.getInstance().getLanguageByIsoCode( "de" );
JaloSession.getCurrentSession().getSessionContext().setLanguage( de );
```

- The basic mechanism of retrieving the language from the SessionContext:

```
Language currLang = JaloSession.getCurrentSession().getSessionContext().getLanguage();
```

Allowed Languages for a Catalog Version

For each catalog version, SAP Commerce allows specifying an individual range of languages. Languages set for a catalog version affect:

- the localization range of attributes displayed

On a catalog version and its related objects (categories, products, etc), SAP Commerce displays localized values only in those languages set for the catalog version. If no languages are specified explicitly, then localized values in all languages will be displayed.

  - For example, if the system languages are `en`, `de`, and `fr`, and only `de` is set for a catalog version, then only the attribute values in German will be visible on categories and products which are part of the catalog version.
  - For example, if the system languages are `en`, `de`, and `fr`, and no language is set for a catalog version, then the attribute values in English, German and French will be visible on categories and products which are part of the catalog version.

- the localization range of attributes used when synchronizing between catalog versions

  Synchronization only affects

  - non-localized attributes and
  - the localized values of the languages set for the catalog version.

Classification

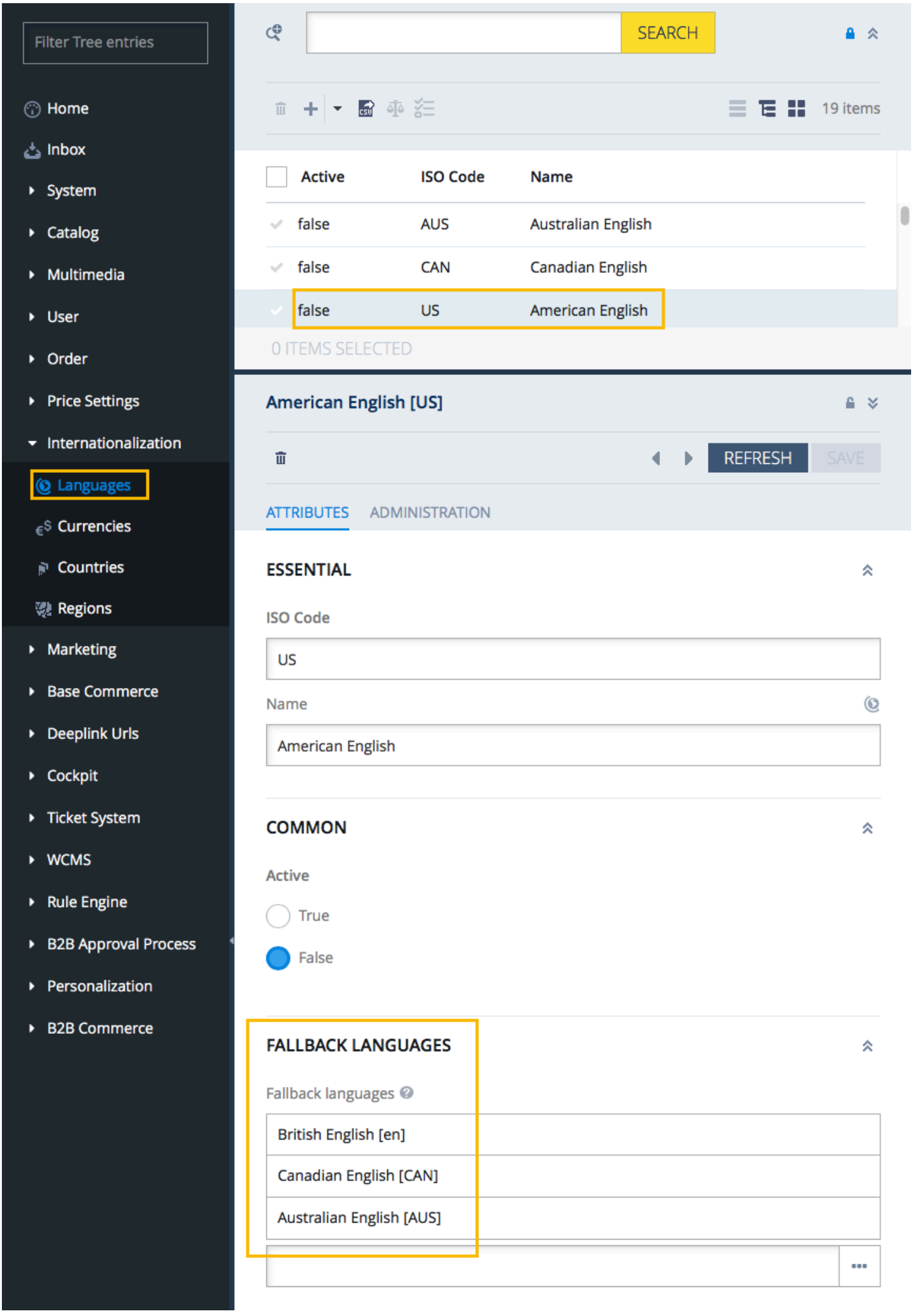
Like the type system -based attributes, the classification-based attributes can also be localized as to the attribute name and value.

Fallback Languages

A fallback is something you use if the original idea does not work out. In the case of fallback languages in SAP Commerce, a fallback language is a language that is used if the "original" language has no value set for a certain localization.

Basically, a fallback languages relies by and large on the localizations of at least one other language and overrides the other language's localizations in selected cases. Fallback languages are useful within a family of languages where many localizations are the same for several languages, but some individual localizations differ. A common application for fallback languages is the German family of languages, where German, Swiss German and Austrian use by and large the same vocabulary and grammar - except for individual differences.

The following screenshot gives an example on this: American English has British English, Canadian English, and Australian English set as fallback languages. By consequence, the localizations of first British English are used wherever American English has no localization.


Filter Tree entries
Home
Inbox
System
Catalog
Multimedia
User
Order
Price Settings
Internationalization
Languages
Currencies
Countries
Regions
Marketing
Base Commerce
Deeplink Urls
Cockpit
Ticket System
WCMS
Rule Engine
B2B Approval Process
Personalization
B2B Commerce
SEARCH
19 items
Active
ISO Code
Name
false
AUS
Australian English
false
CAN
Canadian English
false
US
American English
0 ITEMS SELECTED
American English [US]
REFRESH
SAVE
ATTRIBUTES
ADMINISTRATION
ESSENTIAL
ISO Code
US
Name
American English
COMMON
Active
True
False
FALLBACK LANGUAGES
Fallback languages
British English [en]
Canadian English [CAN]
Australian English [AUS]

When SAP Commerce looks up a localization, the mechanism iterates over fallback languages in the specified order until a localization is found. The first localization that is found in the process is returned. After a localization has been found, the iteration ends.

In addition, fallback hierarchies are not resolved. This is because for every single localization, the entire hierarchy of fallback languages would have to be searched, which would result in bad performance.

For example, assume again that American English has the fallback languages British English, Canadian English, and Australian English. This means that during localization lookup, a localization is first searched in British English, after that, in Canadian English, and, if still no localization has been found, in Australian English. If Australian English contains no localization, then a given attribute is considered unlocalized. If Australian English contains no localization but has a fallback language, that fallback language would not be checked for localization and the attribute would still be considered unlocalized.

Unlocalized attributes are displayed with square brackets (`[` and `]`, respectively) around their name, for example [code], [name], [catalogVersion].

Fallback languages also apply to localized error messages.

Countries and Regions

A Country in SAP Commerce forms an optional part of an Address and can hold any number of Regions.

A Country is an optional part of an Address. Addresses can be used without specifying a country.

A Country can optionally be part of a delivery zone. For more information, see Payment Transaction and Delivery Mode Handling.

Languages are not logically related to Countries, and neither are Countries logically related to Languages. By consequence, the range of languages a user can choose from is not affected by the country they could be living in. In other words, a user does not need to have an address located in an English-speaking country to be able to use the English language. See Ordering Process.

A Region is an optional part of an Address. An Address can optionally hold one single Region, but a Region can be used by any number of Addresses.

Regions may be necessary for cases when you have to sub-divide a country into different individual entities (such as federal states, for example). For example, the individual federal states of the United States of America have individual tax jurisdictions. When calculating tax values for an order, you could use the region of the delivery address to determine the actual tax jurisdiction.

Code Snippets for Using Languages

A localized attribute has two kinds of getter and setter methods. For both getter and setter methods, there is one method that gets or sets all values for all languages via a Java Map, and one method that gets or sets the value for the currently active language only.

For example, for the localized String attribute `MyAttribute`, there will be these generated getter and setter methods:

- Getter methods:
    - Language-specific

      Gets the value for the attribute for the language which is currently set for the active JaloSession. For example, if the JaloSession's language is set to **en**, calling the getter method will result in getting the attribute value for English, and for English only.

      ```
      Product prod = ...
      String name = prod.getName();
      ```

        - `public String getMyAttribute(final SessionContext ctx)`

- `public String getMyAttribute()`

- Generic

  The basic mechanism can be seen from the method parameters: you receive a Map whose key is the language for which to get the value, and whose value is the actual localized value.

  ```
  Map<String, Object> params = new HashMap<String, Object>();
  params.put( WorkflowAction.NAME, getAllName() );
  ```

  - `public Map<Language,String> getAllMyAttribute(final SessionContext ctx)`
  - `public Map<Language,String> getAllMyAttribute()`

- Setter methods:

  - Language-specific

    Sets the value for the attribute for the language which is currently set for the active JaloSession. For example, if the JaloSession's language is set to **en**, calling the setter method will result in the attribute value for English to be set, and for English only.

    ```
    Product p1 = ProductManager.getInstance().createProduct("p1");
    p1.setDescription("description without html");
    ```

    - `public void setMyAttribute(final SessionContext ctx, final String value)`
    - `public void setMyAttribute(final String value)`

  - Generic

    The basic mechanism can be seen from the method parameters: you pass a Map whose key is the language for which to set the value, and whose value is the actual localized value.

    ```
    Map names = new HashMap();
    names.put( de, "beschreibung_de" );
    names.put( en, "description_en" );
    names.put( fr, "légende_fr" );

    Product p = ...
    p.setAllDescriptions( names );
    ```

    - `public void setAllMyAttribute(final SessionContext ctx, final Map<Language,String> value)`
    - `public void setAllMyAttribute(final Map<Language,String> value)`

ImpEx

When setting localized attribute values, you may

- specify the language explicitly for which the value is to be set (recommended)

  Sets the value for the attribute for the specified language.

  ```
  INSERT_UPDATE Category;code[unique=true];name[lang=de];name[lang=en]
  ;CL1100; Jeans; Jeanshosen; jeans
  ```

- omit a language specification (discouraged)

  In this case, the value for the attribute is set for the language which is selected for the current JaloSession. If you can be sure which language will be active for the current JaloSession, then this approach is possible. However, SAP recommends specifying explicitly a language so that you can be sure which language's value will be set.

```
INSERT_UPDATE Category;code[unique=true];name
;CL1100; Jeans; Jeanshosen
```

C2LManager

The Manager class for creation and removal of localization-related SAP Commerce items is the `C2LManager` (`de.hybris.platform.jalo.c2l` package).

For example:

```
Set<Language> allLangs = C2LManager.getInstance().getAllLanguages();
```

Related Information

Internationalization and Localization

Localization

The type system localization uses property files. Every language in SAP Commerce can have individual localizations and, by consequence, have individual property files

ℹ Note

As this document refers to Jalo, find relevant information in Internationalization and Localization Overview.

Type System Localization

The type system localization files are property files and reside in an extension's `/resources/localization` subdirectory.

For more information about property files, see http://en.wikipedia.org/wiki/.properties . For more information about languages, see Languages and Localization.

The localization properties file's name follows the pattern `${extensionname}-locales_${isocode}.properties` (such as `core-locales_en.properties` or `category-locales_de_at.properties`), where:

- `${extensionname}` is the name of the extension
- `${isocode}` is the identifier of a language in SAP Commerce.

Be aware that the delimiter between filename parts is the underscore (`_`). Depending on the application server, other delimiters (for example `-` , or `.`) may be ignored.

ℹ Note

Renaming system languages by runtime will break localizations

SAP Commerce uses the identifier of languages (`isocode` attribute) to allocate localization strings, you will need to make sure that the identifier of a language remains constant. If you modify the value of the `isocode` attribute of a language by runtime and re-update the system, you will break the allocation between localization file name and language isocode. By consequence, localization strings will not be correctly assigned during system initialization or update.

In essence, if you rename languages without adapting the filenames of localization property files, you will end up with incorrectly assigned localization strings - or with no localization strings at all.

The Basic Structure of a Localization Property File

The basic structure of a localization property file looks like this:

```
type.{typecode}.name=value
type.{typecode}.description=value
type.{typecode}.{attributedescriptor}.name=value
type.{typecode}.{attributedescriptor}.description=value

type.customerorderoverview.name=Order statistics
type.customerorderoverview.description=Statistic of order volume of each single customer
```

The line `type.tutorial.partofproduct.name=partOf Product` will navigate downwards through the type system hierarchy like this: `ComposedType type -> subtype tutorial -> attributedescriptor partofproduct -> localizedStringAttribute name` and set this `localizedString` to the value `partOf Product`. This means that localizations for Platform items and those items' attributes relate to actual objects in the database. The following code snippet shows you a sample of an `locales_en` file.

```
type.tutorial.longtext.name=Long text
type.tutorial.longtext.description=This element may contain text that is quite long.
type.tutorial.name=Name
type.tutorial.partofproduct.name=partOf Product
type.tutorial.picture.name=Picture
type.tutorial.picture.description=Via this picture, you may better visualize the product.
type.tutorial.short.name=Short
```

After you made changes to the localization files, call `ant` in the extension directory. Also, you need to update (or initialize) SAP Commerce for the changes to take effect.

ℹ Note

Renaming system languages will break localizations

SAP Commerce uses the identifier of languages (`isocode` attribute) to allocate localization strings, you will need to make sure that the identifier of a language remains constant. If you modify the value of the **isocode** attribute of a language, you will break the allocation between localization string and language. By consequence, localization strings will not be correctly assigned during system initialization or update.

In essence, if you rename languages without adapting the filenames of localization property files, you will end up with incorrectly assigned localization strings - or with no localization strings at all.

Using Other Localized Strings

SAP Commerce also allows using the localization mechanism for localization strings that are not for the Type System, such as for the web shop or for a custom Cockpit implementation.

To use such localization strings:

1. Add the localization values to the type system localization files of your extension, located in the `/resources/localization` subdirectory, such as:

   ```
   my.localized.key=my localized value
   ```

2. Retrieve the localized value in your code by calling

```
String localizedString = de.hybris.platform.util.localization.Localization.getLocalizedString
```

Related Information

Internationalization and Localization Overview

Character Encoding Requirements for Localization Files

Localization files support Unicode in order to display characters of almost all living languages.

This leads to requirements concerning character encoding and file handling.

These requirements are relevant for everyone involved in preparing localization files before they are sent for translation and later reintegrated into SAP Commerce.

Localization files hold source language texts or their localizations, which in turn are used in user interfaces and other system outputs.

What Is a Byte Order Mark?

A byte order mark(BOM) is a sequence of bytes at the beginning of a file. For UTF-8 representation of the BOM, this sequence looks like **EFBBBF** in hexadecimal notation. Find it by opening the file with an hexadecimal editor like:

- 0xEd on the Mac OS X operating system
- HexEdit on the Windows operating system
- VIM with :%!xxd on the Linux operating system

This mark usually defines which character encoding is being used and is quite common with Unicode files.

Character Encoding

Localization files use the following character sets:

Language	Character Encoding	Character Set	Special Character Handling
English	UTF-8	Unicode	Normal
All others	UTF-8	Unicode	Normal

ℹ Note

Byte Order Mark

For proper encoding processing by translators and translation service providers, a BOM needs to be added to all text files before being sent out for translation.

File Handling Before and After Translation

Localization files that are sent out for translation to translators or translation service providers, need to be modified in order to allow proper encoding. After getting the localization files back, they need to be reintegrated.

Requirements for Localization Files to Be Sent Out

The following requirements must be fulfilled before sending out localization files:

1. Add the UTF-8 BOM at the beginning of each file.
2. Convert Unicode to normal UTF-8 characters, for example **U+006Aj**.
3. Correct encoding in UTF-8 representation.

If this is not done, the translation service provider might not be able to translate the files correctly or might not even be able to open them.

Requirements for Localization Files Before Reintegration

Before reintegrating, the following requirements must be fulfilled for localization files:

1. There should be no BOMs at the beginning of the file. If there are any, you need to remove all UTF-8 BOMs.
2. Convert Unicode to normal UTF-8 characters. In this case, it is easier to read these files.
3. Correct encoding in UTF-8.

If this is not done correctly, errors might occur while importing/building the Platform and it could cause character corruptions in the frontend/backend.

As you can see, reintegration and sending out processes are quite similar to each other.

Related Information

Localization Files Reference

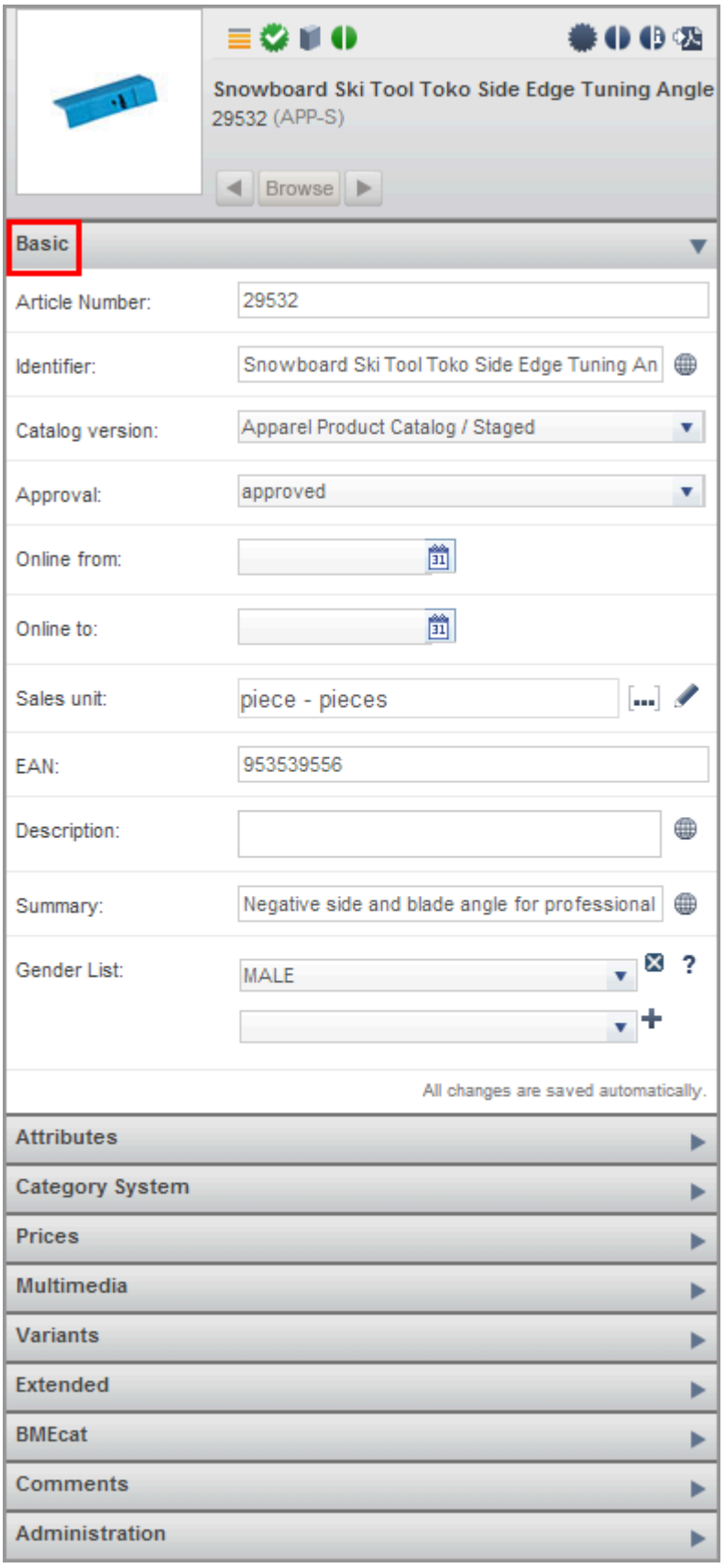
Cockpit UI Element Localization

One of the steps in the process of SAP Commerce localization is providing the localization of cockpit UI elements.

To do that you need to add an element to UI XML file, and next put the localized string with the key into the properties file. It is an easy way to manage all the different languages.

What Are the Cockpit UI Elements?

Cockpit UI Elements are elements like labels of the different sections. In the cockpits UI, you can find the Editor Area in the Organizer. For example, in the Product cockpit in the Editor Area the main top first label is **Basic**.



The most common UI elements of the cockpit are Advanced Search, List View, Editor, and Wizard.

Where to Find the UI XML Files

The XML files can be found in one of the following folders:

- *<HYBRIS_BIN_DIR/ext.@{extname}.path>* `/resources/@{extname}/import/config/`
- *<HYBRIS_BIN_DIR>* `/platform/ext/@{extname}/resources/@{extname}`

In these folders, you can find files which names are associated with the areas existing in the UI cockpits. For example, the expressions included in the XML filenames and associated with the names of areas are `AdvancedSearch`, `ListView`, `Editor`, and `Wizard`. The rest part of the XML filenames specifies also which part of the UI cockpit is concerned.

For instance, you can search for a product file associated with the **Editor Area** in the Product cockpit for the cockpit user. You can find it in *<HYBRIS_BIN_DIR>*`platform/ext/cockpit/resources/cockpit/import/config/` folder as an `Editor_Product_CockpitUser.xml` file.

Where to Find the Properties Files

In SAP Commerce, there are properties files corresponding to the cockpit UI XML files.

The properties files can be found in one of the following folders:

- *<HYBRIS_BIN_DIR/ext.@{extname}.path>*`/resources/localization/`
- *<HYBRIS_BIN_DIR>*`/platform/ext/@{extname}/resources/localization`

The naming convention is `i3-label_` **languagetag**`.properties`.

ℹ Note

The `i3-label.properties` does not have `_en` language tag in the filename.

Localization of the Cockpit UI Elements

The localization of the cockpit UI elements requires two substeps. Firstly, you need to provide correct **label key** in the UI XML file. In the second substep, translate the string in the properties file. This string is assigned to the **label key** placed in the UI XML file.

While localizing the cockpit UI elements, add:

1. A `<label key="` **`key.name`** `"/>` element to your UI XML file where you want your string to be placed later on.
2. The localized string with the **key.name** into the appropriate localization properties file accordingly: **key.name** = **Key Value**.

UI XML File

In the UI XML file, insert `<label key="` **`key.name`** `" />` to add a key for your label that you can localize through a language specific properties file later on.

In the following example, the added **label key** is `<label key="cockpit.config.label.General" />`.

`Editor_Product_CockpitUser.xml`

```
<editor>

        <group qualifier="General" visible="true" initially-opened="true">
                <label key="cockpit.config.label.General" />
                <property qualifier="product.code" />
                <property qualifier="product.name" />
                <property qualifier="product.catalogversion"  editor="shortListEditor"/>
                <property qualifier="product.approvalStatus" />
                <property qualifier="product.onlineDate" />
                <property qualifier="product.offlineDate" />
                <property qualifier="product.unit" />
                <property qualifier="product.ean" />
                <property qualifier="product.description" editor="wysiwyg"/>
```

```
        </group>

        <custom-group
                class="de.hybris.platform.cockpit.services.config.impl.ClassAttrEditorSectionConfig
                qualifier="classification"
                initially-opened="false"
                show-if-empty="false">
                <label key="config.general.attributes" />
        </custom-group>
```

Properties File

To properly localize the properties file, open it concerning the correct extension and language tag. Next, add the key and localized string to it.

You can find the example of part of the localized properties file below. The first line of the code shows how to assign the localized string in German with the correct **label key** found in the UI XML file: `cockpit.config.label.General=Stammdaten`. The UI XML file is the `Editor_Product_CockpitUser.xml` file.

`i3-label_de.properties`

```
cockpit.config.label.General=Stammdaten
cockpit.config.label.Additional=Erweitert
cockpit.config.label.Other=Andere
cockpit.config.label.All=Alle

config.general.actions=Aktionen
config.general.administration=Verwaltung
config.general.assigneduser=Zugewiesenen Benutzer
config.general.attributes=Attribute
config.general.categorystructure=Kategoriestruktur
config.general.categorysystem=Kategoriesystem
config.general.comments=Kommentare
```

Related Information

Character Encoding Requirements for Localization Files
Localization Files Reference

Localization Files Reference

SAP Commerce comes with a variety of localization files. Localization files hold source language texts or their localizations, for example used for display in the user interface or within other system output.

Language Code in the Filename

The language code in file names is indicated by a language tag: ***_languagetag***. ***extension***.

The language tag follows language code pattern ISO 639-1 for all files to be translated, for example:

- `_en.properties`
- `_en.zul`
- `_de.properties`
- `_fr.properties`
- `_ja.properties`

- `_zh.properties`
- `_it.properties`
- `_pt.properties`

ℹ Note

There are English source files which do not have `_en` language tag such as:

- `BasecommerceMessages.properties`
- `i3-label.properties`

If multiple variants of a language are to be offered, for example German for Germany and German for Switzerland, the following standard is applied ***_languagetag _ countrytag.extension***.

For examples:

- `_de_DE.properties`
- `_de_CH.properties`
- `_pt_BR.properties`

Localization Files

Localization files sent out for translation or translations received from the translation service providers need to be modified in order to have proper encoding. Before sending localization files out or reintegrating them, you need to prepare them. You can read about the requirements for localization files in Character Encoding Requirements for Localization Files.

In the next sections there are tables which indicate how to find the expressions which are to be searched for and then translated.

In the **Filename pattern** row there are filenames which you need to search for. These files include the expressions needed to be translated.

In the **Content pattern** you can find structure indicating the strings, called also values, to be translated.

Properties Files

Properties files have the extension `.properties`. They are distributed over multiple directories of the different extensions.

Filename Pattern	`.properties`
Content Pattern	***key=value***
To Be Translated	***value***

There are different kinds of properties files each focusing on a different kind of object:

- **Types**.
- **UI Components**.
- **Messages**.
- **Cockpits**.

Type Driven Files

Filename pattern	***extension*** `-locales_` ***languagetag*** `.properties`
Content pattern	**type.personobjectclassenum.name=PersonObjectClassEnum**

Example	**ldap-locales_en.properties**
	
	`type.personobjectclassenum.name=PersonObjectClassEnum`
	`type.personobjectclassenum.description=`
	`type.personobjectclassenum.person.name=Person`
	`type.personobjectclassenum.person.description=`
	`type.personobjectclassenum.posixaccount.name=posixAccount`
	`type.personobjectclassenum.posixaccount.description=`

UI Component Driven Files

Filename pattern	`locales_` ***languagetag*** `.properties`
Content pattern	**tab.ldapconfig.enhanced=Enhanced Settings**

Example	**locales_en.properties**
	
	`tab.ldapconfig.basics=Basic Settings`
	`tab.ldapconfig.enhanced=Enhanced Settings`
	`section.ldap=LDAP`
	`section.ldap.test.connection=Connection Test`
	`tab.ldapconfig.settings=LDAP`
	`section.ldap.login=Login`
	`section.ldap.jndi=JNDI`
	`section.ldap.pool=Pool`
	`section.ldap.internal=Internals`
	`action.testldapconfiguration=Test`
	`action.ldapconnection.notyetcreated=Item not created\!`
	`action.ldapconnection.succesfull=Connection test successfull\!`
	`action.ldapconnection.failed=Connection test failed\!`
	`b2bcommercegroup.units.description=B2B Units are nodes of a B2B Organization such as a company o`
	`b2bcommercegroup.customers.description=Customer who are members of a B2B Organization who make p`
	`type_tree_ldifimportwizard=Basic Import`

Messages Files

Filename pattern	***extension*** `-messages_` ***languagetag*** `.properties` • Search for: **messages.properties** or **messages_en.properties**
Content pattern	**paymentOption.billingInfo.validation.noLastName=Missing last name**

Cockpit i3 Files

These files are like the UI component type files which might include acronyms such as ba = browser area or na = navigation area.

Filename pattern	`i3-label.properties`
Content pattern	**attribute_pricecalculationdate = Price calculation date**

Example	**i3-label.properties**
	
	`ba.constraint_browsermodel_label=Validierungsregeln`
	`ba.constraint_pojo_browsermodel_label=POJO Validierungsregeln`
	
	`importcockpit.perspective.job=Jobs`
	`importcockpit.perspective.mapping=Mapping`
	`importcockpit.navigationarea.jobhistory=Execution history`
	`attribute_pricecalculationdate = Price calculation date`
	`attribute_root_chapter = Root Chapter`
	`attribute_status = Status`
	`attribute_subtitle = Sub Title`
	`attribute_subtitle2 = Subtitle 2`
	`attribute_title = Title`

ℹ Note

The `i3-label.properties` do not have `_en` language tag in the filename.

Cockpit UI Element Configuration XML Files

Cockpit UI element configuration files are XML files with names such as `Editor_Product_ProductManager.xml`. They are included in the cockpit extensions.

These files contain source text and translations for all languages.

Filename pattern	***text*** `_` ***text*** `.xml` or ***text*** `_` ***text*** `_` ***text*** `.xml` • Search for: **.xml**. The file contains **<label lang="en"**. All related files are XML files with one of these root tags: ◦ **<editor** ◦ **<advanced-search** ◦ **<list-view** ◦ **<wizard-config** They are located in the ***${HYBRIS_BIN_DIR}*** / ***extension*** / `resources` directory of a cockpit-like extension such as `cockpit` extension, `productcockpit` extension, `cmscockpit` extension. But they also can be found in other extensions, for example `sampledata` extension or `hyend2` extension.
To Be Translated	The following tags indicate strings that have to be translated: **<label lang="en">** ***Source language text*** **</label>**.

	The corresponding translations are stored below in the same file: **<label lang="** ***languagetag*** **">** ***Language translation*** **</label>**. For example: **<label lang="fr">** ***French translation*** **</label> <label lang="de">** ***German translation*** **</label> <label lang="ja">** ***Japanese translation*** **</label>**

Related Information

Character Encoding Requirements for Localization Files

Supported Languages

See the language options available in Backoffice and included in the shipped SAP Commerce software.

The language options include:

- en: English
- cs: Czech
- de: German
- es: Spanish (Spain)
- es_CO: Latin-American Spanish
- fr: French
- hi: Hindi
- hu: Hungarian
- id: Indonesian
- it: Italian
- ja: Japanese
- ko: Korean
- pl: Polish
- pt: Brazilian Portuguese
- ru: Russian
- zh: Simplified Chinese
- zh_TW: Traditional Chinese

Jalo Layer

The Jalo Layer is the traditional SAP Commerce functional layer. It combines both data model and business logic aspects. Jalo is now deprecated.

ℹ Note

To read about the ServiceLayer, which has replaced Jalo, see Transitioning to the ServiceLayer

Schematic Overview

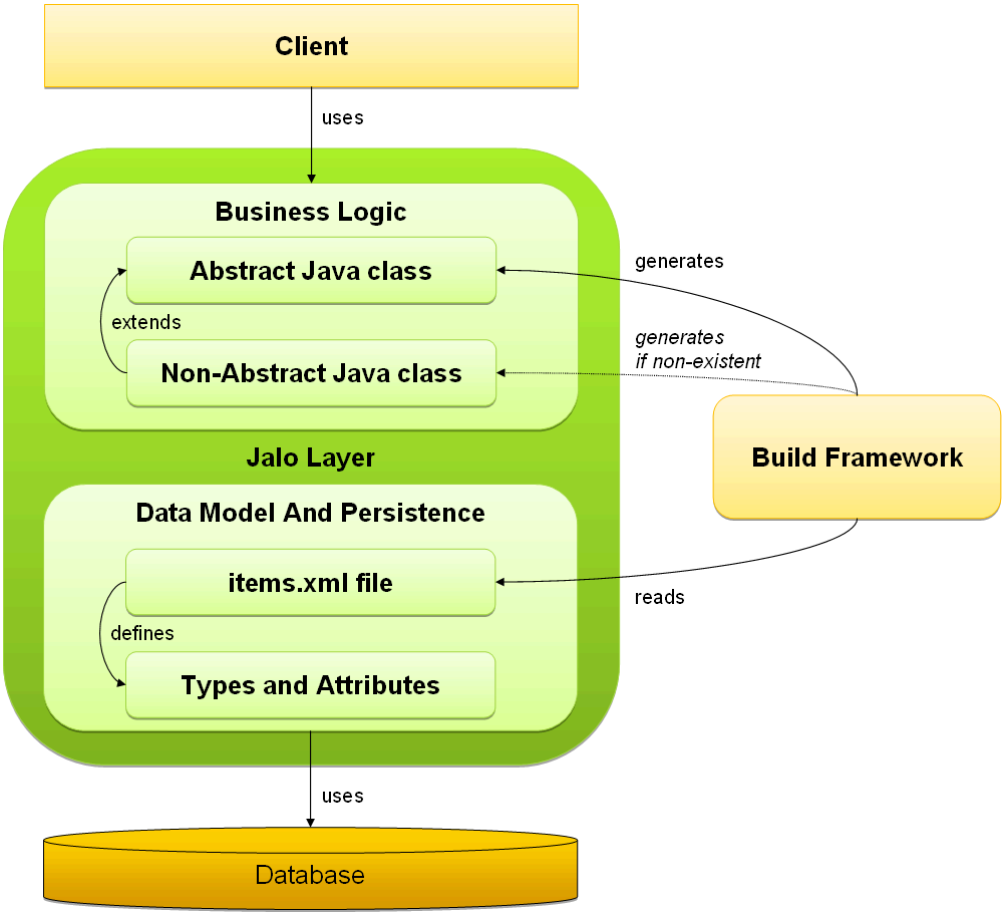
The Jalo Layer offers an API to clients using SAP Commerce and abstracts from the database. It also covers two major aspects of SAP Commerce:

- **Data Model**, which you can define in the `items.xml` files in the form of types and attributes.
- **Business Logic**, which you implement in Java classes.

For each type defined in the `items.xml` file, the SAP Commerce Build Framework generates two Java files:

- **Abstract Java class**: Automatically generated by the SAP Commerce Build Framework and generated again on every build. Abstract Java classes contain automatically-generated getter and setter methods for the defined type attributes in the `items.xml` file.
- **Non-Abstract Java class**: Extends from the abstract Java class. Non-Abstract Java classes are only generated by the SAP Commerce Build Framework if they are non-existent. Non-Abstract Java classes are not overwritten during builds.

Overview of the Jalo Layer:



The Jalo Layer is a tight coupling between data model and business logic, as the implemented business logic in Java classes that are generated are based on the data model. If the data model changes, your business logic might need adaption as well. For example, if you rename a type, the Java classes have to be renamed too. For a more loose coupling between data model and business logic, see ServiceLayer.

You can implement Java classes that are not backed by the data model, but the instances are runtime objects only and are not persistent. An example for such non-persistent Java classes is the Jalo Session.

Using the Jalo Layer

The basic workflow of using the Jalo layer is as follows:

- Define your data model in terms of types and attributes using the `items.xml` file.
- Build SAP Commerce to have the Java classes generated.

- Implement business logic in the non-abstract Java class.

Below is a concrete example of using the Jalo Layer.

The following `items.xml` file sample defines the type `MyType` with a single String attribute, `MyAttribute`:

```
<itemtype code="MyType" autocreate="true" generate="true"
        jaloclass="de.hybris.jalolayer.sample.MyType">
        <attributes>
                <attribute type="java.lang.String" qualifier="MyAttribute">
                        <persistence type="property" />
                </attribute>
        </attributes>
</itemtype>
```

Running a build of SAP Commerce will result in two Java classes being generated:

Fully Qualified Java Class Name	Path and File Name
`de.hybris.jalolayer.sample.GeneratedMyType`	`gensrc/de/hybris/jalolayer/sample/GeneratedMyType.java`
`de.hybris.jalolayer.sample.MyType`	`src/de/hybris/jalolayer/sample/MyType.java`

Related Information

items.xml
items.xml Element Reference
The Type System
Building SAP Commerce
Jalo-only Attributes
ServiceLayer
About Extensions
FlexibleSearch
Using ImpEx with Backoffice or SAP Commerce Administration Console

Jalo Session

The **JaloSession** is a core concept of SAP Commerce. A JaloSession wraps up data about the current user and their settings. It forms a context in which SAP Commerce processes user-specific requests.

Overview of Jalo Sessions

Every request from a web browser to SAP Commerce is associated with a JaloSession. But also SAP Commerce CronJobs run in a JaloSession, see The Cronjob Service.

ℹ Note

Jalo is now deprecated. For more information, refer to Transitioning to the ServiceLayer

A JaloSession is always associated with a SessionContext that holds references to the following:

- JaloSession's User
- JaloSession's Language
- JaloSession's Currency
- JaloSession's Pricefactory
- JaloSession's Locale
- JaloSession's Timezone

During the JaloSession's instantiation, the SessionContext is preset with system defaults, which depend on the current tenant. The factory default user account is `anonymous`.

To access the JaloSession assigned to a thread, call the static `JaloSession.getCurrentSession()` method. This method does the following:

- Returns the currently active JaloSession.
- Creates a new JaloSession if none is currently active.

Characteristics

- There's always 1 JaloSession active for a given thread.
- JaloSession objects are held in memory only, they aren't made persistent (written into the database). The JaloSession class implements the `Serializable` interface so that JaloSession objects could be transferred across application server nodes within a SAP Commerce cluster.

  Being nonpersistent, creating and removing JaloSessions is a quick process. However, as JaloSession objects aren't made persistent, a JaloSession cannot be recovered once it has been deactivated. A JaloSession that has timed out or that has been deactivated cannot be restored.
- Although a JaloSession holds a reference to a Cart, the Cart is not actually instantiated along with the JaloSession. Unlike JaloSessions, Carts are made persistent (written into the database). Because the database access, creating a Cart is

slower than creating a JaloSession. Therefore, SAP Commerce only instantiates the JaloSession's when the Cart is addressed via the JaloSession class' `getCart()` method.

Calling the `getCart()` method therefore instantiates the Cart associated with the JaloSession, and - by consequence - is a rather slow process. To find out whether or not a JaloSession has a Cart, call the JaloSession class' `hasCart()` method.

- like HttpSession or EJB Session beans
    - State objects that hold state between calls
- For web application use, a JaloSession can be assigned to an HTTP session using the `httpSession` attribute. See also Transparent HTTP Session Failover.

JaloSession Lifecycle

Getting a Jalo Session

There are 4 approaches to getting a JaloSession:

- Explicitly instantiating the JaloSession via the SAP Commerce API, such as:
    - `JaloSession jaloSession = JaloConnection.createAnonymousCustomerSession();`
    - `JaloSession jaloSession = JaloConnection.createSession( Map );`
    - `JaloSession jaloSession = JaloSession.createInstance( Map );`
- "Externally" using Java Servlet filters, such as the `HybrisInitFilter`. SAP Commerce automatically creates a JaloSession when a web browser connects to a Java Servlet filter in SAP Commerce.
- "Implicitly" by using a JaloSession without explicit instantiation, such as
    - `JaloSession js = getSession();`
    - `final JaloSession session = JaloSession.getCurrentSession();`

Instantiation of a JaloSession

By calling `JaloSession.getCurrentSession()`, a reference to the currently active JaloSession is returned. If no JaloSession is active, SAP Commerce creates a new JaloSession. The running order given here is rather random from a technical point of view.

A new instance of `JaloSession` is created. SAP Commerce calls `newInstance()` method on the JaloSession class as JaloSessions aren'tnot persistent. SAP Commerce makes sure that the new JaloSession is marked as used by calling the `touch()` method. By assigning the tenant ID to the JaloSession, the JaloSession is bound to the currently active tenant. By creating and assigning a SessionID (basically a PK) to the JaloSession, the JaloSession is made identifiable. A set of basic login properties is set to the JaloSession and the user is logged in. In addition, SAP Commerce sets the language and the currency to the JaloSession.

- Creation of a new SessionContext
    - Set the tenant's time zone for the SessionContext
    - Assign the SessionContext to the JaloSession

Modifying a JaloSession and the SessionContext

A JaloSession holds a certain constellation of user, currency, tenant, and language using itself and its `SessionContext` object. By default, the user set to the JaloSession's SessionContext is `anonymous`. To change any of these settings (because a customer has logged in, for example), you need to modify the JaloSession or the SessionContext.

Some common modifications of a JaloSession:

- Setting the JaloSession to the User whose login (code) is `myUserName`:

```
jalosession.getSessionContext().setUser( jalosession.getUserManager().getUserByLogin( "myUser
```

- Setting the JaloSession to an admin user:

```
jalosession.getSessionContext().setUser( jalosession.getUserManager().getAdminEmployee() );
```

- Setting the JaloSession's language:

```
jalosession.getSessionContext().setLanguage( jalosession.getC2LManager().getLanguageByIsoCode
```

For other means of modifying JaloSessions, also refer to the JaloSession API Doc and the SessionContext API Doc.

Closing a JaloSession

Closing a JaloSession results in the JaloSession being made invalid. SAP Commerce will not allow users or CronJobs to use closed JaloSessions. If a user tries to log into SAP Commerce, such as shop application, SAP Commerce creates a new JaloSession for the login process. There are 2 aspects that militate in favor of closing JaloSessions:

- Security

  Closing JaloSessions prevents "taking over" existing JaloSessions

- Performance

  Keeping the number of active JaloSessions down to a minimum reduces load on the SAP Commerce server.

There are 2 approaches to closing a JaloSession:

- Automatically via a session timeout

  The JaloSession's `timeout` attribute specifies the expiration of the JaloSession in seconds. For many practical purposes, letting the JaloSessions time out automatically will do.

  #### ℹ Note

  SAP Commerce removes expired JaloSessions automatically via the JaloSession class.

  The default timeout value is specified by the `default.session.timeout` property in the SAP Commerce `project.property` file (please refer to Configuring the Behavior of SAP Commerce for additional details). The `default.session.timeout` property defaults to 3600 (specified in seconds). Specifying a non-positive value for the `timeout` attribute of a JaloSession disables the timeout, as in the following code snippets. CronJobs are a common field of application for using a non-timeout JaloSession, for example.

  - Timeout of ten seconds:

```
JaloSession.getCurrentSession().setTimeout( 10 );
```

- No timeout:

```
JaloSession.getCurrentSession().setTimeout( 0 );
```

- No timeout:

```
JaloSession.getCurrentSession().setTimeout( -1 );
```

Setting Session Timeout for Specific Web Applications

It is also possible to configure session timeout per web application. Just set the following property to a non-negative number in the `local.properties` file:

```
[extension].session.timeout=1500
```

Values lower than 0 will be ignored and *<default.session>* timeout will be used instead. 0 means that session will never time out. For example, in order to set Administration Console session timeout to 1000 seconds, add the following property to `local.properties`.

```
hac.session.timeout=1000
```

Restart the server for your changes to take effect.

By calling the JaloSession class's `close()` method, the JaloSession is invalidated outright, as in

```
JaloSession.getCurrentSession().close();
```

Calling this method is useful if the JaloSession needs to be turned invalid right away, such as during a log-out. Do not call this method if you need to use the JaloSession later on.

JaloSessions and Tenants

A JaloSession is always bound to a tenant. The binding of a JaloSession to a tenant occurs during the JaloSession's instantiation and cannot be changed. Setting the tenant to which a JaloSession belongs is possible only during the JaloSession's instantiation but no more once the JaloSession has been instantiated (that is, after the `JaloSession` class' `createInstance( ...)` method has completed). No setter method for the tenant is available for the JaloSession.

As JaloSessions are bound to the tenant at instantiation, you cannot use one single JaloSession across a multi-tenant system (that is, using one single JaloSession to manage multiple tenants is not possible). Instead, you have to use at least 1 JaloSession per tenant.

When developing, you will be able to switch between JaloSessions that are bound to individual tenants (using the `JaloConnection.getSession( String id)` method, for example). However, each JaloSession remains tenant-specific, and you'll need to use one individual JaloSession per tenant.

JaloSession Inside HTTPSession

How JaloSession is Contained in HTTPSession

The Hybris specific session object (the `JaloSession` class) is usually bound to `HttpSession` and has the same life time. This means that if an http session is closed or encounters a timeout, both the http session and the SAP Commerce session are

destroyed.

Technically, this behavior requires a proper setup of the SAP Commerce filter chain with `SessionFilter` being responsible for the process. For more information, see Platform Filters.

Defining the Global Session Timeout

By default, the Hybris Platform overrides the http session timeout for all web applications using the `default.session.timeout` property that you can override in `local.properties`. Note that this means that you **cannot** set the timeout in `web.xml`!

Executing Custom Code on Session Invalidation

In addition to managing the http session timeout, the `SessionCloseStrategy` also allows you to hook into the session invalidation. For this purpose, use the `de.hybris.platform.servicelayer.web.DefaultSessionCloseStrategy#closeSessionInHttpSession` method. See the example:

```
public class HacSessionCloseStrategy extends DefaultSessionCloseStrategy
{
    private static final Logger LOG = Logger.getLogger(HacSessionCloseStrategy.class.getName());

    @Override
    public void closeSessionInHttpSession(final HttpSession httpSession)
    {
        if (LOG.isDebugEnabled())
        {
            LOG.debug("HACSessionCloseStrategy works!");
        }
        super.closeSessionInHttpSession(httpSession);
    }
}
```

In the `closeSessionInHttpSession(..)` method, JALOSession is available via the `httpSession` attribute:

```
final JaloSession js = (JaloSession) httpSession.getAttribute(Constants.WEB.JALOSESSION)
```

When entering `closeSessionInHttpSession(..)`, JALOSession is already active and gets closed immediately after leaving the method, so that any custom logic related to the session can be executed here.

> ⚠ **Caution**
>
> When extending `DefaultSessionCloseStrategy` like you can see above, it is crucial to call `super.closeSessionInHttpSession(httpSession)` because this deals with the correct closing and disposal of the embedded `JaloSession`!

For more information, see Creating Web Applications.

Using a Custom JaloSession Implementation

SAP Commerce allows using a custom JaloSession implementation for each tenant using Spring. Refer to Spring Framework in SAP Commerce for details.

Related Information

Users in Platform

Logging

Logging is an important part of software development that helps you to analyze the behavior of your applications.

Logging with Log4j 2 Through SLF4J

SAP Commerce comes prebundled with Apache Log4j 2.

JDBC Logs from Database Statements

SAP Commerce sends database statements to query and modify data in the database. You can create a record of these statements. The process of recording the statements is called JDBC logging.

Item Attribute Modification History

SAP Commerce has a built-in mechanism to keep track of modifications of attribute values. These SavedValues store the original value of the attribute (before modification) and the new value (after modification). This provides a useful history upon which new features may be built.

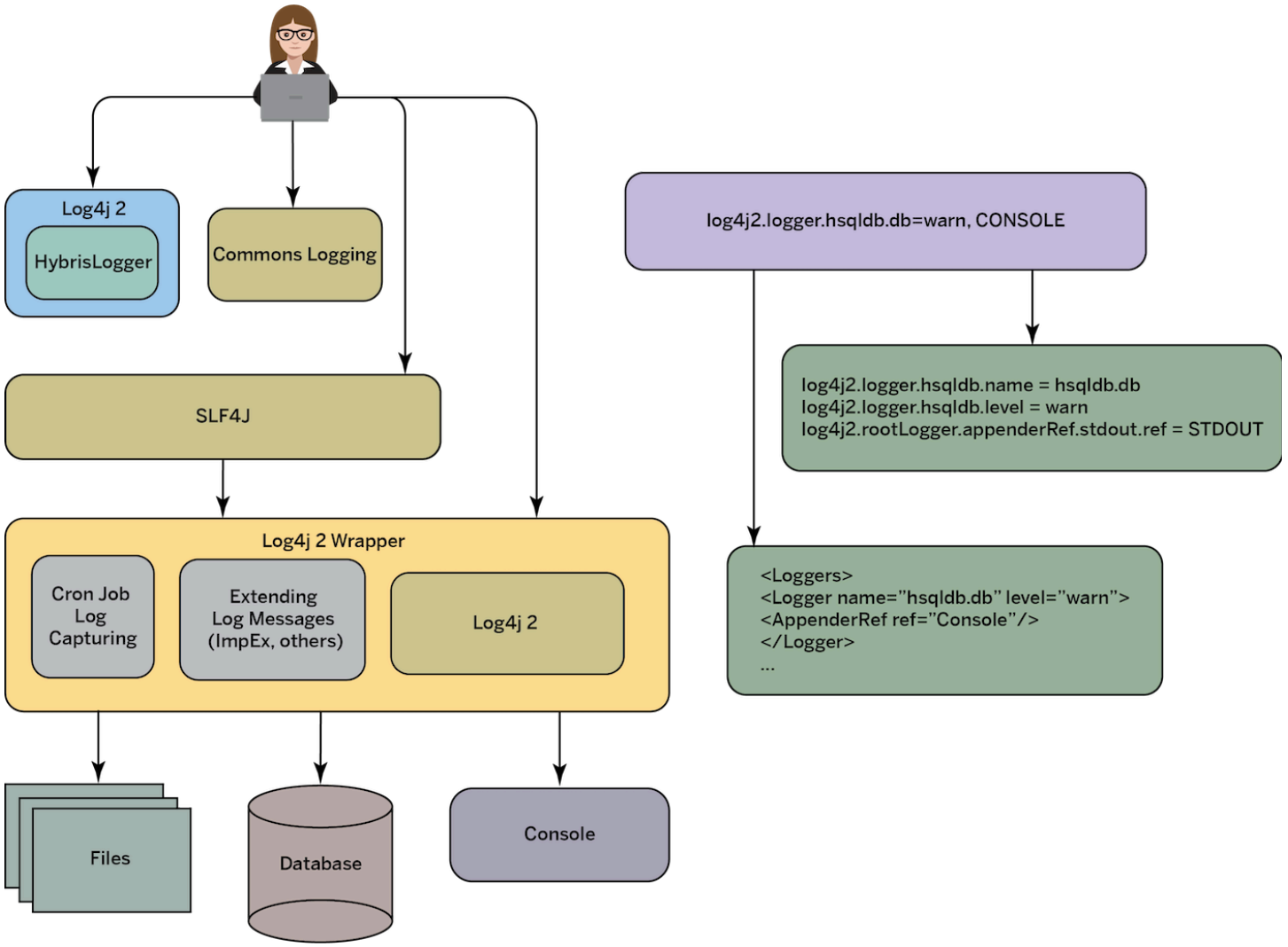
Logging with Log4j 2 Through SLF4J

SAP Commerce comes prebundled with Apache Log4j 2.

Log4j 2 is the standard logging framework in SAP Commerce.

It's recommended that you use Simple Logging Facade for Java (SLF4J) as the logging API for Log4j 2:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
final Logger slf4jLogger = LoggerFactory.getLogger(LoggingFrameworksBridgeTest.class);
slf4jLogger.error("slf4j error");
```



Related Information

Apache Log4j

Log4j 2 Configuration Elements

Configure Log4j 2 using loggers, appenders, and layouts.

There are three main configuration elements in Log4j 2:

- loggers
- appenders
- layouts

Loggers

Loggers are objects that are responsible for capturing logging information from events. Configure loggers to ensure that the system logs all the relevant information.

Whether a certain event triggers a certain logger depends on the event's importance called priority or log level. If the event's log level in Log4j 2 is equal to or higher than the logger's log level threshold, the logger prints out a message. If the event's log level is lower than the logger's log level, the logger isn't triggered.

Log4j 2 uses the following log levels:

Log Level	Description	Triggered Log Levels
OFF	The highest log level that turns logging off	OFF
FATAL	Severe errors that might stop the system	OFF, FATAL
ERROR	Errors that aren't likely to stop the system	OFF, FATAL, ERROR
WARN	Potentially damaging situations	OFF, FATAL, ERROR, WARN
INFO	General information about the system	OFF, FATAL, ERROR, WARN, INFO
DEBUG	Detailed information about the system that can be used in debugging	OFF, FATAL, ERROR, WARN, INFO, DEBUG
TRACE	More detailed information about the system	OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE
ALL	All levels	OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE, ALL

As shown in the table, a logger set to the ALL level reports all events that are passed to it. A logger that is set to OFF reports events only of the OFF level. Loggers that are created without a log level inherit log levels from their closest parent logger.

ⓘ Note

Logging at the DEBUG, TRACE, and ALL levels may leak confidential library data into the logs. To prevent libraries from leaking such data, do not enable these levels in production environments and other environments where loss of confidentiality could have harmful effects. Enable the DEBUG, TRACE, and ALL levels selectively and to the extent absolutely necessary, for example only for specific loggers such as package names.

You can configure and create loggers in any order. It's also possible to create loggers whose parents haven't been created yet. A logger automatically puts itself in its correct place in the inheritance hierarchy once its ancestors are all there.

Additivity of Loggers

Child loggers that inherit logging settings from their parent loggers pass output to a selected logging destination twice or more due to the additivity of loggers in Log4j2. To avoid duplicated logging, disable additivity for selected loggers.

See the following example of two loggers for *<myBasePackagename>* and *<myBasePackage>.<extension>* packages:

```
log4j2.logger.<myBasePackage>=info, appconsole
log4j2.logger.<myBasePackage>.<extension>=debug, appconsole
```

For `log4j2.logger.`*`<myBasePackage>`*, the system runs every logging statement once. For `log4j2.logger.`*`<myBasePackage>.<extension>`*, the system runs every logging statement twice - once for the global `log4j2.logger.`*`<myBasePackage>`* logger and once for the extension-specific `log4j2.logger.`*`<myBasePackage>.<extension>`* logger that inherits its settings from the `log4j2.logger.`*`<myBasePackage>`* parent logger. To disable logging data for `log4j2.logger.`*`<myBasePackage>.<extension>`*, use the following property:

```
log4j2.additivity.<myBasePackage>.<extension>=false
```

Creating Loggers

Learn how to create loggers to log information about events.

Procedure

1. Import the `org.apache.log4j.Logger` class:

   ```
   import org.apache.log4j.Logger;
   ```

2. Call the `getLogger(String name)` method of the `Logger` static class. For more information about this class, see Log4J API .

   ```
   static final Logger log = Logger.getLogger( "my.log4j.topic" );
   ```

3. Call your logger:

   ```
   ${instancename}.${logLevel}("${message}");
   ```

Results

You have created a listing similar to the following:

```
import org.apache.log4j.Logger;
static final Logger log = Logger.getLogger( "my.log4j.topic" );
log.fatal("This is wrong!");
log.warn("Something hasn't worked out");
log.info("Just for you to know.");
```

This listing creates three log requests. A log request is enabled if its log level is higher or equal to the trigger's log level and disabled if it's lower.

See an example of an extended listing:

```
import the package
import org.apache.log4j.Logger;

static final Logger log = Logger.getLogger("my.log4j.topic");
private int test = 27;

try {
    someMethod(test);
} catch (NullPointerException NPEx) {
    log.fatal("test has value null!");
    test = 0;
    break;
} catch (Exception Ex) {
    log.warn("Something went wrong, but at least test is not null!");
}
if (test == 27)
    log.debug("test is 27");
else
    log.info("test iUsing s different from 27!");
```

Appenders

Appenders designate the output destination of your logged data.

Appenders ensure that logged data is sent to a selected destination, such as a system console, log files, UNIX daemons, or GUI elements. You can assign one logger to multiple appenders to log data in multiple destinations at the same time.

Loggers inherit appenders from their closest parent logger. For example, when you create a logger called “log1” and assign it to the “app1” appender, any other loggers that are derived from the “log1” logger, for example “log2” and “log3”, receive the “app1” appender. If you add a new “app2” appender to “log2” and create a “log4” logger from it, the new logger gets both the “app1” and “app2” appenders thanks to the additivity of loggers:

Layouts

Layouts allow you to decide how your logged data is formatted and what information is included in your log messages.

Layouts are templates that you can use to format messages passed on by your loggers. They allow you to include various information in your log messages, such as names of loggers that have been used or the system's running time. For more information on logging patterns, see Layouts .

Configuring Your Log Environment

SAP Commerce is shipped with a default logger configuration that you can find in your `project.properties` file. You can modify it to fulfill the needs of your project.

You can configure Log4 2 in SAP Commerce using the following files:

- XML files
- `local.properties` file

XML Configuration Files

To configure Log4j 2 using XML, navigate to the `platform/bin/platform/ext/core/resources/hybris-log4j2.xml` file and add your configuration file there. When your configuration is ready, navigate to the `local.properties` file and add the name of the configuration file as the value of the `log4j2.config.xml` property:

```
log4j2.config.xml=hybris-log4j2.xml
```

Note

Don't create multiple XML configuration files as SAP Commerce is unable to merge them.

`local.properties` Configuration

The `project.properties` file contains a default logger configuration that you can overwrite by adding properties to your `local.properties` file.

Using the `local.properties` file, you can configure selected loggers separately for each of your extensions, for example:

```
log4j2.logger.<extensionName>.name = de.hybris.platform.<extensionName>
log4j2.logger.<extensionName>.level = debug
log4j2.logger.<extensionName>.additivity = false
log4j2.logger.<extensionName>.appenderRef.stdout.ref = STDOUT
```

Configuration Example

See an example of how to configure Log4j 2 appenders and loggers using properties:

```
# Console logging (default setting, located in <hybrishome>/bin/platform/project.properties)
log4j2.appender.console.type = Console
log4j2.appender.console.name = STDOUT
log4j2.appender.console.layout.type = PatternLayout
log4j2.appender.console.layout.pattern = %highlight{%-5p [%t] %X{RemoteAddr}%X{Tenant}%X{CronJob}[%

log4j2.rootLogger.level = info
log4j2.rootLogger.appenderRefs = stdout
log4j2.rootLogger.appenderRef.stdout.ref = STDOUT

# Rolling file appender
log4j2.appender.rolling.type = RollingFile
log4j2.appender.rolling.fileName = ${HYBRIS_LOG_DIR}/hybrislog.txt
log4j2.appender.rolling.filePattern = ${HYBRIS_LOG_DIR}/hybrislog-%i.txt
log4j2.appender.rolling.name = Rolling
log4j2.appender.rolling.policies.type = Policies
log4j2.appender.rolling.policies.size.type = SizeBasedTriggeringPolicy
log4j2.appender.rolling.policies.size.size=10MB
log4j2.appender.rolling.strategy.type = DefaultRolloverStrategy
log4j2.appender.rolling.strategy.max = 10
log4j2.appender.rolling.layout.type = PatternLayout
log4j2.appender.rolling.layout.pattern = %highlight{%-5p [%t] %X{RemoteAddr}%X{Tenant}%X{CronJob}[%

log4j2.rootLogger.appenderRefs = stdout, files
log4j2.rootLogger.appenderRef.files.ref = Rolling

# Logging your package
log4j2.logger.your.name = yourpackage
log4j2.logger.your.level = debug

# Logging your package only to a dedicated file
log4j2.appender.yourfile.type = File
log4j2.appender.yourfile.fileName = ${HYBRIS_LOG_DIR}/your.log
log4j2.appender.yourfile.name = YourAppender
log4j2.appender.yourfile.layout.type = PatternLayout
log4j2.appender.yourfile.layout.pattern = %{%-5p [%t] %X{RemoteAddr}%X{Tenant}%X{CronJob}[%c{1}] %m

log4j2.logger.your.appenderRefs = yourRef
log4j2.logger.your.appenderRef.yourRef.ref = YourAppender

# additivity = false will stop logging your.package to other more generic appenders
log4j2.logger.your.additivity = false

# some useful logging statements and settings

# --- for logging all flexiblesearch statements with DEBUG into "Rolling" appender, but with info o
# --- the additivity setting will only give the logger to the own configured logger and not
# --- (in 'addition' to that) to the root logger
log4j2.logger.flexiblesearch.name = de.hybris.platform.jalo.flexiblesearch
log4j2.logger.flexiblesearch.level = debug
log4j2.logger.flexiblesearch.additivity = false
log4j2.logger.flexiblesearch.appenderRefs = rolling
log4j2.logger.flexiblesearch.appenderRef.rolling.ref = Rolling

# --- for debugging cronjobs and triggers:
log4j2.logger.timertask.name  = de.hybris.platform.cronjob.jalo.TimerTaskUtils
log4j2.logger.timertask.level = debug

log4j2.logger.cronjobtimer.name  = de.hybris.platform.cronjob.jalo.CronJobTimerTask
log4j2.logger.cronjobtimer.level = debug
```

```
# --- for debugging switching of tenants:
log4j2.logger.registry.name  = de.hybris.platform.core.Registry
log4j2.logger.registry.level = debug

log4j2.logger.abstracttenant.name  = de.hybris.platform.core.AbstractTenant
log4j2.logger.abstracttenant.level = debug
log4j2.logger.mastertenant.name  = de.hybris.platform.core.MasterTenant
log4j2.logger.mastertenant.level = debug
log4j2.logger.slavetenant.name  = de.hybris.platform.core.SlaveTenant
log4j2.logger.slavetenant.level = debug
```

Additional Appender Attributes

SAP Commerce allows you to use the following `layout.pattern` appender attributes:

- `RemoteAddr` - returns the address of the remote host request that is being processed
- `Tenant` - returns the ID of the tenant where selected events are being performed
- `CronJob` - returns the code of a cron job that is performing selected events

Colored Logging in Console

Color logging improves the readability of logged information.

Enabling Colored Logging

Colored logging in consoles isn't enabled by default. To enable it, add the following property to your `local.properties` file:

```
ansi.colors=true
```

Colored logging in consoles is supported on all platforms except for Microsoft Windows, which doesn't support it natively.

Default Colors

See the default colors of Log4j 2 log levels:

Parameter	**fatalColor**	**errorColor**	**warnColor**	**infoColor**	**debugColor**	**traceColor**
Default color	Bright red	Bright red	Yellow	Green	Cyan	Black

Modifying Colors

You can modify the default colors of your log levels in your `local.properties` file. See an example of a layout that sets new colors for log levels:

```
log4j2.appender.console.layout.pattern = %highlight{%-5p [%t] %X{RemoteAddr}%X{Tenant}%X{CronJob}[%
```

This layout sets the following color pattern:

- `FATAL` - white
- `ERROR` - red
- `WARN` - blue

- `INFO` - black
- `DEBUG` - green
- `TRACE` - blue

Upgrading to Log4j 2

SAP Commerce allows you to check whether your system still uses any Log4j 1.x configuration properties.

Use the following properties to check whether your system still uses any Log4j 1.x configuration properties:

- `log4j.deprecated.properties.warn=true` - allows you to check whether your instance of SAP Commerce still uses any Log4j 1.x properties by printing a warning in your logs. The property is set to `true` by default.
- `log4j.deprecated.properties.print=false` - allows you to print a list of Log4j 1.x properties in your logs that are still being used by the system. The property is set to `false` by default.

Restart your system to apply these properties. Replace any old Log4j 1.x properties with their Log4j 2.x equivalents to ensure that SAP Commerce logs all information correctly.

See an example of a Log4j 1 configuration property:

```
log4j.logger.hsqldb.db=warn, CONSOLE
```

This setting can be represented by the following Log4j 2 properties:

```
log4j2.logger.hsqldb.name = hsqldb.db
log4j2.logger.hsqldb.level = warn
log4j2.logger.hsqldb.appenderRef.stdout.ref = STDOUT
```

For more information on migrating to Log4j 2 configuration, see Log4j 2 Configuration Format .

Ignoring Log4j 2 Properties

Use the `remove` attribute to make SAP Commerce ignore selected Log4j 2 properties.

To ignore selected Log4J properties, use the `remove` attribute, for example:

```
remove.log4j2.appender.jdbcConsoleLogger.layout.pattern=true
```

Using the `remove.log4j2.appender.jdbcConsoleLogger.layout.pattern=true` property allows you to disable the `log4j2.appender.jdbcConsoleLogger.layout.pattern` property.

JDBC Logs from Database Statements

SAP Commerce sends database statements to query and modify data in the database. You can create a record of these statements. The process of recording the statements is called JDBC logging.

SAP Commerce uses JDBC to connect to databases. As a JDBC log contains information on statements, execution times, and other information, you can use it to identify and remedy performance bottlenecks on the database.

You can decide how you want to store JDBC logs by choosing one of the following implementations of the logger class:

- `FileLogger`
- `JDBCSLF4JAwareLogger`

Understanding a JDBC Log File (`FileLogger`)

In a JDBC log file, each line is an individual logged statement. Each line consists of a number of bits of information, delimited by the pipe character (**|**). This example full statement is broken down into multiple lines so that it's easier to read:

```
34|master|091106-15:01:04:136|1 ms|statement|SELECT  item_t0.PK  FROM tasks item_t0 WHERE item_t0.p
SELECT  item_t0.PK  FROM tasks item_t0 WHERE item_t0.p_failed  =0 |
PreparedStatementImpl:57:DataSourceImplFactory:108:ConnectionImpl:338:ConnectionImpl:627:FlexibleSe
FlexibleSearch:1854:FlexibleSearch:1515:FlexibleSearch:1366:FlexibleSearch:1357:DefaultFlexibleSear
DefaultSessionService:89:DefaultFlexibleSearchService:446:DefaultFlexibleSearchService:179:QueryBas
QueryBasedTasksProvider:102:ShufflingTasksProvider:31:BufferedTasksProvider:51:DefaultTasksProvider
DefaultTaskService:1142:DefaultTaskService$Poll:1018:DefaultTaskService$Poll:969:AbstractTenant$5:2
```

Note

By default, you don't get full logs like the one in the example. To get full logs, that is logs consisting of both your prepared statement and your actual statement with values, set `db.log.sql.parameters` to `true`. To get stack trace info, set `db.log.appendStackTrace` to `true`.

Each line of the JDBC log file consists of the following fields (from left to right):

Field Name	Field Description	Value in Sample Line
Thread ID	The ID of the thread that executed the statement.	34
Datasource ID	The ID of the used datasource.	master
Start Time	Date and time when the statement was started.	091106-15:01:04:136
Elapsed Time	Amount of time that the execution of the statement took.	1 ms
Category	The category of the statement. Possible values: `statement`, `commit`, `rollback`.	statement

Prepared statement	Each statement sent to the database has to be compiled. When executing numerous similar statements (for example, when the only difference is in parameters) prepared statements are used for better performance. The prepared statement is precompiled without parameters and then can be reused many times.	`SELECT  item_t0.PK`
		`FROM tasks item_t0`
		`WHERE item_t0.p_failed  = ?`

Field Name	Field Description	Value in Sample Line
SQL statement	The actual SQL statement that was executed on the database. Disabled by default. Set the `db.log.sql.parameters=true` property to enable it.	`SELECT item_t0.PK FROM tasks item_t0 WHERE item_t0.p_failed = 0`
Stack trace	The actual stack trace. Disabled by default. Set the `db.log.appendStackTrace=true` property to enable it.	`PreparedStatementImpl:57:DataSourceImplFactory:108:ConnectionI FlexibleSearch:1854:FlexibleSearch:1515:FlexibleSearch:1366:Fl DefaultSessionService:89:DefaultFlexibleSearchService:446:Defa QueryBasedTasksProvider:102:ShufflingTasksProvider:31:Buffered DefaultTaskService:1142:DefaultTaskService$Poll:1018:DefaultTa`

JDBC SLF4J Logger (`JDBCSLF4JAwareLogger`)

`JDBCSLF4JAwareLogger` acts as a wrapper and delegates all logging to SLF4J, which in turn uses Log4j 2 as an implementation. That allows you to customize log layout and define where you want to store logs. There are two appenders configured for this logger: file, and console. In contrast to the JDBC Log File, some of the parameters are passed in context (by MDC - Mapped Diagnostic Context).

Here are the names of the parameters passed in context:

Field name	Parameter name
Datasource ID	`jdbc-dataSourceId`
Elapsed Time	`jdbc-elapsedTime`
Category	`jdbc-category`
Start Time	`jdbc-startTime`

The information stored in the MDC can be exposed in the set layout, for example, in PatternLayout using the `%X{parameterName}` placeholder (see https://logging.apache.org/log4j/2.x/manual/layouts.html):

```
log4j2.appender.console.layout.pattern = %-5p [%t] %X{jdbc-dataSourceId} %c{1}] %m%n
```

To enable or disable JDBC through SLF4J logging, use the `db.log.active` property. All messages are logged with the debug level. The debug level is enabled for `JDBCSLF4JAwareLogger`.

Here is the current log4j2 configuration for `JDBCSLF4JAwareLogger`:

```
log4j2.appender.jdbcConsoleLogger.type = Console
log4j2.appender.jdbcConsoleLogger.name=JdbcConsoleLogger
log4j2.appender.jdbcConsoleLogger.layout.type=PatternLayout
log4j2.appender.jdbcConsoleLogger.layout.pattern=%-5p [%t] %d{HH:mm:ss.SSS} [%X{jdbc-dataSourceId}

log4j2.appender.jdbcFileLogger.type = File
log4j2.appender.jdbcFileLogger.fileName=${HYBRIS_LOG_DIR}/jdbc.log
log4j2.appender.jdbcFileLogger.name=JdbcFileLogger
log4j2.appender.jdbcFileLogger.layout.type=PatternLayout
log4j2.appender.jdbcFileLogger.layout.pattern=%tid|%X{jdbc-dataSourceId}|%X{jdbc-startTime}|%X{jdbc

log4j2.logger.jdbc-sl4j.name = de.hybris.platform.jdbcwrapper.logger.JDBCSLF4JAwareLogger
log4j2.logger.jdbc-sl4j.level = debug
```

```
log4j2.logger.jdbc-sl4j.additivity = false
log4j2.logger.jdbc-sl4j.appenderRef.jdbcFileLogger.ref = JdbcFileLogger
log4j2.logger.jdbc-sl4j.appenderRef.jdbcConsoleLogger.ref = JdbcConsoleLogger
```

The configured `JdbcFileLogger` appender must be enabled and must have a defined layout pattern in order to use the JDBC Log Analysis function from SAP Commerce Administration Console:

```
%tid|%X{jdbc-dataSourceId}|%X{jdbc-startTime}|%X{jdbc-timeElapsed} ms|%X{jdbc-category}|%m%n
```

JDBC SLF4J Logger Tip

With `db.log.active` enabled with the default configured appenders, the log is stored on console and in the `jdbc.log` file. You can disable logging to these appenders and define your own appender with, for example, the JSON format:

```
log4j2.logger.jdbc-sl4j.appenderRef.jdbcFileLogger.level=OFF
log4j2.logger.jdbc-sl4j.appenderRef.jdbcConsoleLogger.level=OFF


log4j2.appender.jdbcFileJsonLogger.type = File
log4j2.appender.jdbcFileJsonLogger.fileName=${HYBRIS_LOG_DIR}/jdbc.log
log4j2.appender.jdbcFileJsonLogger.name=JdbcFileJsonLogger
log4j2.appender.jdbcFileJsonLogger.layout.type=JsonLayout
log4j2.appender.jdbcFileJsonLogger.layout.properties=true

log4j2.logger.jdbc-sl4j.appenderRef.jdbcFileJsonLogger.ref = JdbcFileJsonLogger
```

Enabling and Disabling JDBC Logging

You can enable or disable JDBC logging by using SAP Commerce Administration Console, Java code, or through the configuration properties file.

Enabling or disabling JDBC logging in SAP Commerce Administration Console has a run-time effect only - logging is set back to default after an SAP Commerce restart.

Enabling or disabling JDBC logging by using Java code has a runtime effect only - logging is set back to default after an SAP Commerce restart.

Enabling or disabling JDBC logging through the configuration properties file has a persistent effect - logging is enabled or disabled after an SAP Commerce restart.

ℹ Note

JDBC Logging Reduces Performance

Because every single database statement is logged, the database performance is reduced with JDBC logging enabled. Enable JDBC logging only when needed.

Enabling JDBC Logging in SAP Commerce Administration Console

In SAP Commerce Administration Console, you can enable or disable JDBC logging on the **Monitoring** > **Database** > **JDBC logging** page.

Enabling JDBC Logging Using Java Code

To enable JDBC logging in your Java code, add the following line:

```
de.hybris.platform.util.Config.setParameter("db.log.active", "true");
```

To disable JDBC logging in your Java code, add the following line:

```
de.hybris.platform.util.Config.setParameter("db.log.active", "false");
```

See an example of how to enable and disable JDBC logging:

```
private void myMethod()
   {
         de.hybris.platform.util.Config.setParameter("db.log.active", "true");
         myProductModel.setCode("ModelCode");
         modelService.save(myProductModel);
         de.hybris.platform.util.Config.setParameter("db.log.active", "false");
   }
```

Enabling JDBC Logging Through a Properties File

To ensure that the logging setting remains enabled or disabled persistently, set an appropriate property value in the `local.properties` file.

To enable JDBC logging, use `db.log.active=true`.

To disable JDBC logging, use `db.log.active=false`.

Specify a fully qualified class name of the logger class to the `db.log.loggerclass` config property. You can choose one of the following implementations:

- `de.hybris.platform.jdbcwrapper.logger.FileLogger`
- `de.hybris.platform.jdbcwrapper.logger.JDBCSLF4JAwareLogger`

Here is an example:

```
db.log.loggerclass=de.hybris.platform.jdbcwrapper.logger.FileLogger
```

For details about the `local.properties` file, see Configuring the Behavior of SAP Commerce.

Configuration Properties

Use properties to customize JDBC logging:

Property	Possible Value	Description
`db.log.sql.parameters`	`true` / `false`	When enabled, it logs a full sql statement with actual parameters. It's disabled in the default configuration to prevent leaking of confidential data, such as passwords, into logs.

Related Information

Third-Party Databases
Performance Tuning

Tips on Performance Tuning
Available Interceptors

Item Attribute Modification History

SAP Commerce has a built-in mechanism to keep track of modifications of attribute values. These SavedValues store the original value of the attribute (before modification) and the new value (after modification). This provides a useful history upon which new features may be built.

SAP Commerce allows logging the modification of attribute values of SAP Commerce items.

i Note

As every single attribute value modification may be logged, the database tables in which the logs are stored might grow to a very large size over time. If you don't need the history of SavedValues, you can remove the SavedValue-related database table content from time to time. Also see Performance Tuning Overview.

Related Information

Creating Strategy to Process Instances

Viewing Attribute Modifications in Backoffice

Follow the steps to view `SavedValues` modifications using Backoffice.

Context

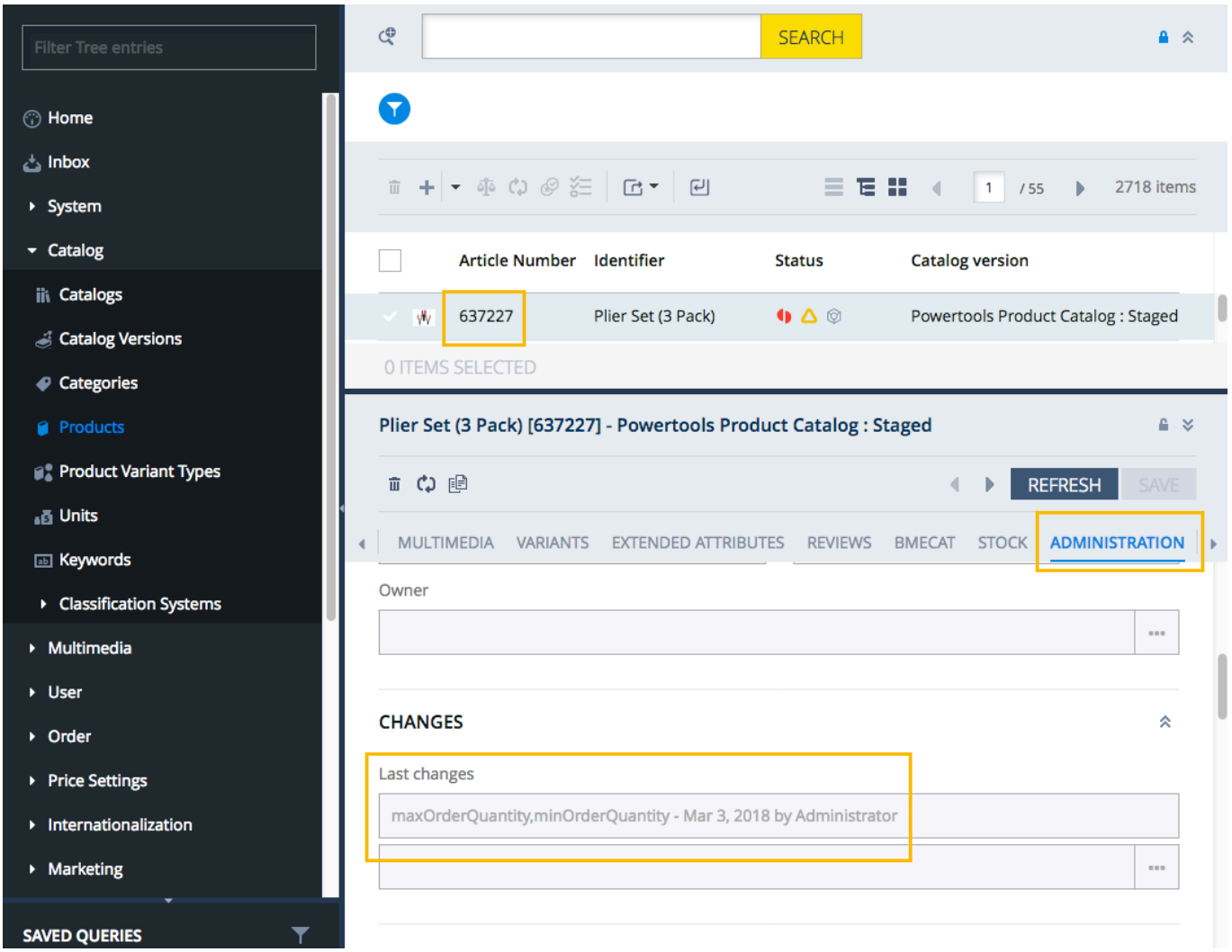
If you modify the value of a SAP Commerce item attribute in Backoffice, the original value is stored in the SAP Commerce database. This process is run automatically, you do not have to perform any manual steps.

To review the changes made on an attribute values:

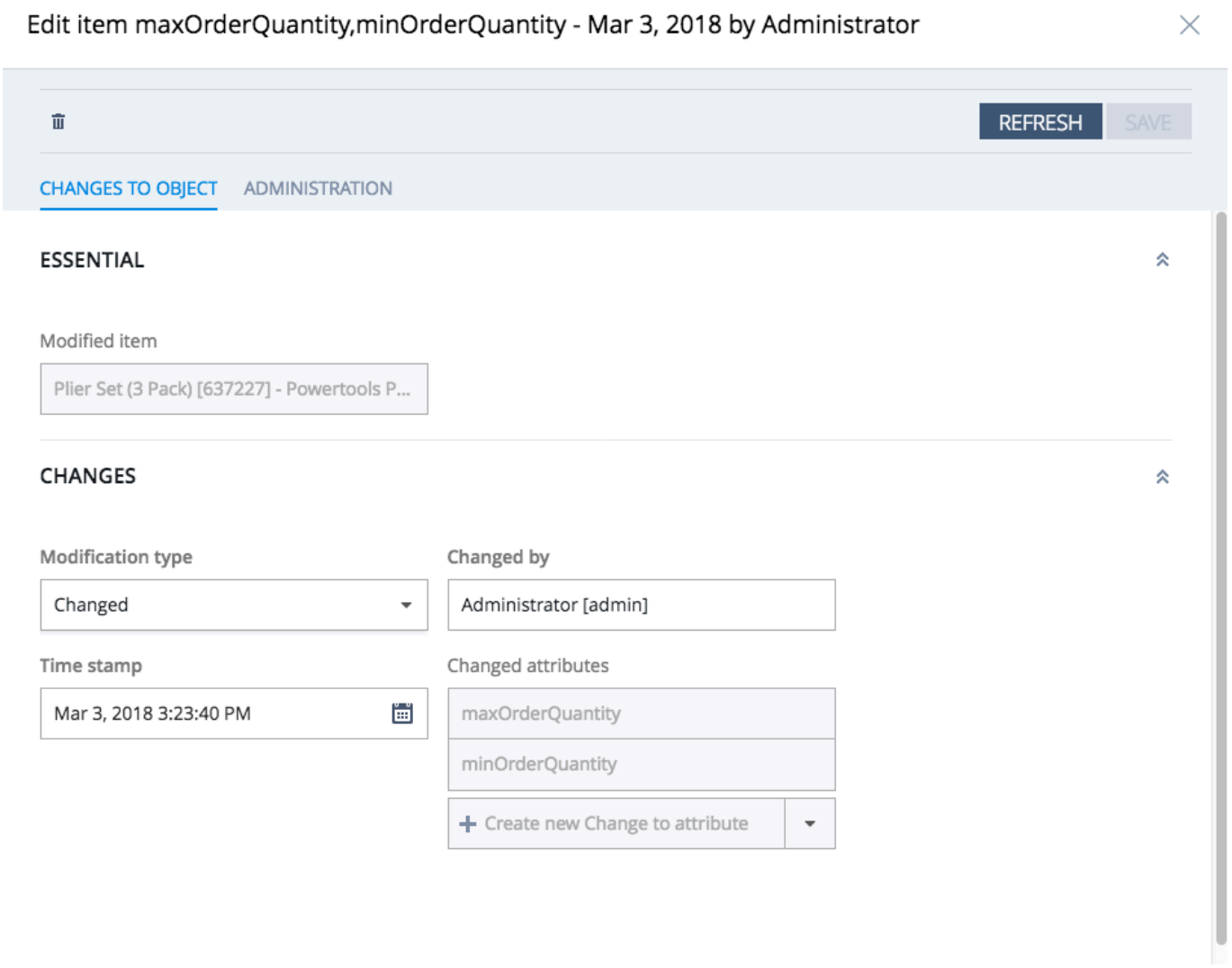
Procedure

1. Log into Backoffice.
2. Open the item that contains the modified attribute values that you want to see.
3. Navigate to the **Administration** tab.

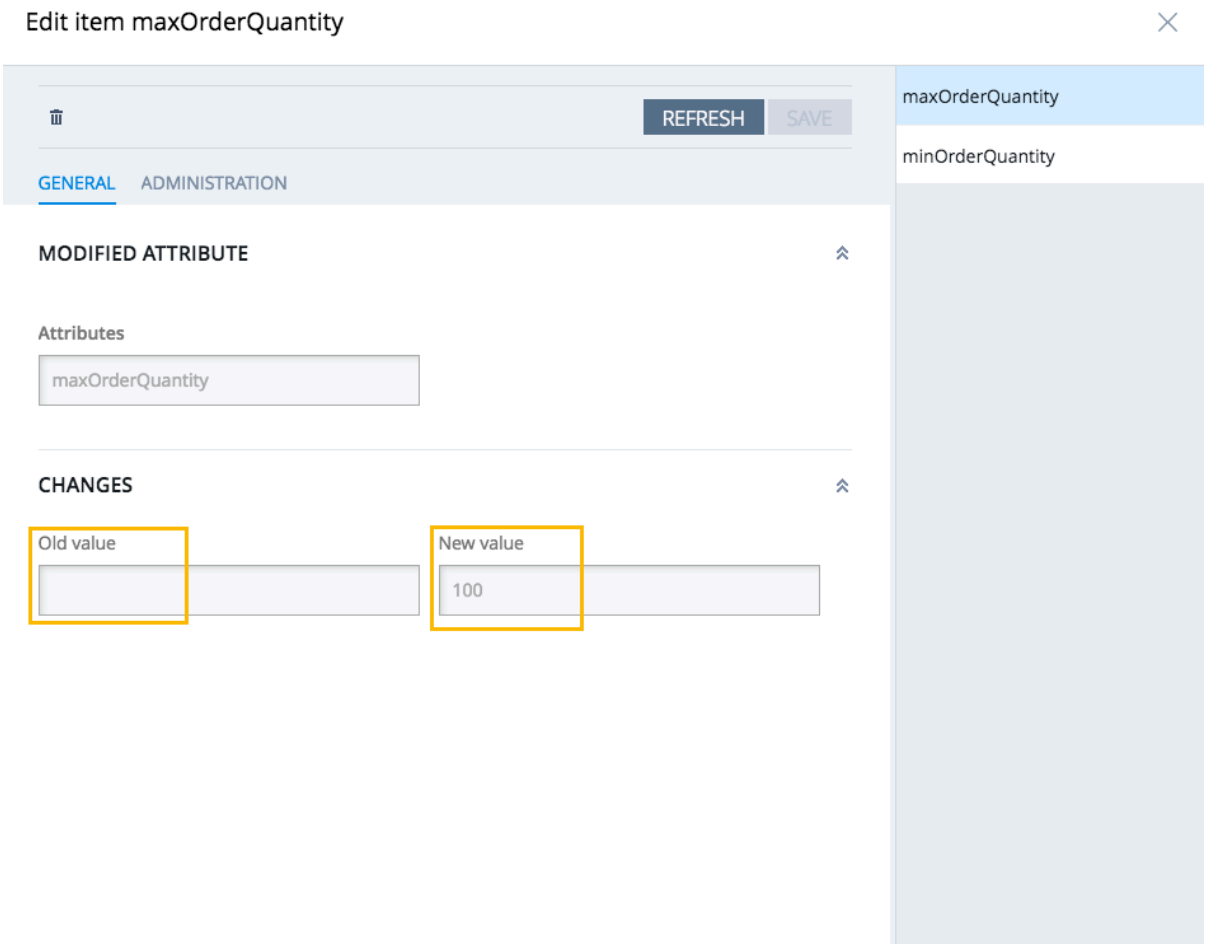
   Logs for attribute changes are shown in the **CHANGES** > **Last changes** section. You can see that the log includes changes in two attributes, `maxOrderQuantity` and `minOrderQuantity`.



4. Click a given log twice. A window displays with names of modified attributes:



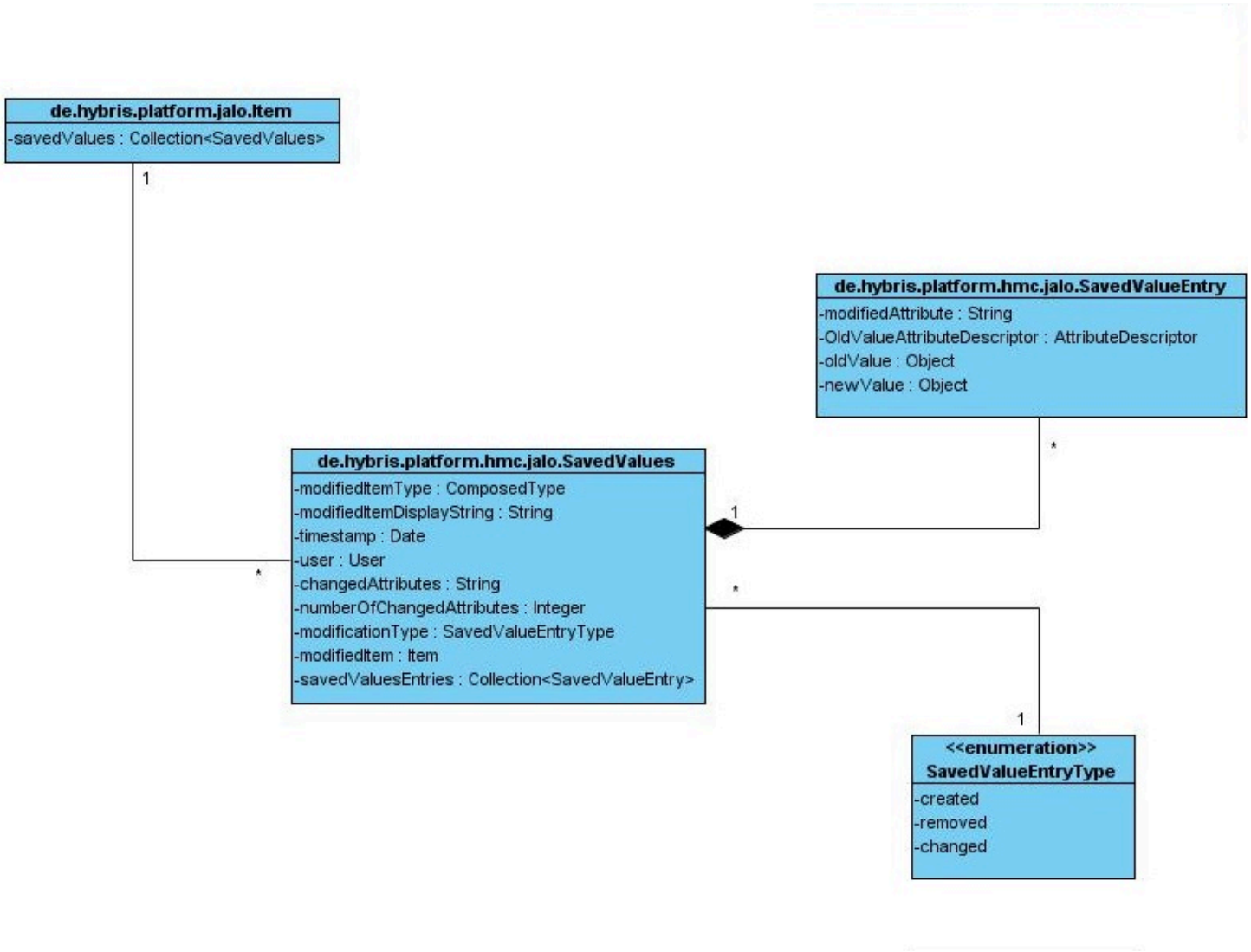
5. Click a given attribute name twice to display the actual changes made to the attribute values:



Technical Discussion

This section gives an overview of how to use the SavedValues functionality on a technical level.

Every SAP Commerce item can contain a collection of `SavedValues`. A single `SavedValues` consists of timestamp data and a collection of `SavedValueEntries`. Each `SavedValueEntry` stores a single individual attribute value modification. A `SavedValue` can hold several `SavedValueEntries` as several attributes can be modified at once (for example, using ImpEx).



SavedValues Cleanup

You can decide when you want Platform to clean up existing `SavedValues`.

The current default cleanup of `SavedValues` works in the following way:

- The number of maximum `SavedValues` that are persisted for an instance is determined by the Platform property: `hmc.storing.modifiedvalues.size`.
- Current threshold for `SavedValues` creation is 20. It means that if the instance is modified for the 21 time and new `SavedValues` is created, the oldest `SavedValues` is automatically removed.

It is possible to use the `CleanupSavedValuesStrategy` to remove all `SavedValues`, which are over the threshold defined in the Platform `hmc.storing.modifiedvalues.size` property. This strategy uses the same threeshold value as set in `hmc.storing.modifiedvalues.size` property. It means that if you did not set a value for this property lower then currently set, then executing strategy does not delete any `SavedValues`, because the existing number of `SavedValues` did not exceed the current threshold. In other words, the `CleanupSavedValuesStrategy` is useful when used in the following way:

- You modify the `hmc.storing.modifiedvalues.size` property to have lower threshold then currently set. It does not mean that after server startup `SavedValues` that were created before and already exceed the new threshold are automatically removed. Below is an example of new threshold value:

  `local.properties`

```
hmc.storing.modifiedvalues.size=15
```

- You should assume that there are already `SavedValues` that were created before modification of the default threshold. `SavedValues` that are over the new threshold are removed:
    - After next modification of the specific instance for which `SavedValues` were created.
    - When running a cron job with assigned `cleanupSavedValuesPerformable` job.

OAuth2

The `oauth2` core extension has replaced the `webservicescommons/oauthauthorizationserver` extension. It exposes the HTTP endpoint as an authorization server. It doesn't introduce any new significant functionalities.

Enabling and Configuring oauth2

To enable the authorization server, add the `oauth2` extension entry into the `localextensions.xml` file:

```
<extension name="oauth2" />
```

By default, the web root is `authorizationserver/` but you can change it using the `oauth2.webroot=...` property.

To configure the `oauth2` extension, in the `project.properties` file use the following properties:

Property Name	Description	Type	Default Value
`webservicescommons.required.channel`	Channel protocol for oauth/* endpoints	http/https	https
`oauthauthorizationserver.tokenServices.reuseRefreshToken`	Specifies if a new refresh token should be created during refreshing an access token	boolean	false
`oauth2.supportRefreshToken`	Enables the refresh token	boolean	true
`oauth2.refreshTokenValiditySeconds`	Refresh token time-to-live	seconds	2592000 (30 days)
`oauth2.cleanupAccessToken.maxRows`	Max rows fetched on cleaning up the obsolete access token table	int	100
`oauth2.accessTokenValiditySeconds`	Access token time-to-live	seconds	43200 (12 hours)
`oauth2.optimize.accesstoken.save.enabled`	Optimizes the performance of the `oauth2` extension by ensuring that the `OAuthAccessTokenModel` object isn't updated when the `saveAccessToken` method is called and there are no	boolean	true

Property Name	Description	Type	Default Value
	changes to the `OAuth2Authentication` object.		
`oauth2.accesstoken.save.retry`	Ensures that the system retries to save access tokens in situations when an attempt to save such tokens ends with the `ModelSavingException` exception being thrown. Such situations are a result of duplicate access token IDs being created by two or more threads trying to create the same access token in `HybrisOAuthTokenStore`.	boolean	true
`oauth2.authorizationcode.stored.as.sha.signature`	Enables storing authorization codes as an SHA signature.	boolean	true
`oauth2.authorizationcode.length`	Allows you to configure the length of generated authorization codes.	int	30
`oauthauthorizationserver.tokenServices.refreshWithLock`	Prevents the `DuplicateKeyException` exception caused by `authenticationIdIdx` index unique constraint violation. With selective thread locking implemented for HSQLDB and database row locking implemented for databases other than HSQLDB, refresh tokens cannot longer be consumed more than once by concurrent requests. For databases other than HSQLDB, the property also introduces database row locking for refresh tokens before refresh tokens are removed, which prevents potential deadlocks that could occur when access tokens are refreshed and refresh tokens are revoked at the same time.	boolean	true

In the **System**/**OAuth** tab in Backoffice, you can manage OAuth clients and access tokens.

Deserialised Access Tokens after Spring Security Upgrade

Since all access tokens are serialized in the database, they are prone to deserialization when Spring Security is upgraded. Remove all your access tokens whenever Spring Security is upgraded.

For more information on upgraded libraries, see Additional Information.

Authorization Server

The authorization server exposes two endpoints:

- /oauth/authorize
- /oauth/token

For details on usage and interfaces, see the RFC specification at https://tools.ietf.org/html/rfc6749#section-1.3.3 .

Response Encoding

Response content type might be in a JSON format, for example:

/oauth/token - proper credentials - json

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{"access_token":"a084da1c-72af-4cf7-a461-986de3f3070f","token_type":"bearer","expires_in":34191,"sc
```

or in XML, for example:

/oauth/token - proper credentials - xml

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<oauth><access_token>a084da1c-72af-4cf7-a461-986de3f3070f</access_token><expires_in>33966</expires_
```

The same applies to errors:

/oauth/token - bad credentials - json

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="hybris", error="invalid_client", error_description="Bad client crede
Content-Type: application/json;charset=UTF-8

{"error":"invalid_client","error_description":"Bad client credentials"}
```

/oauth/token - bad credentials - xml

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="hybris", error="invalid_client", error_description="Bad client crede
Content-Type: application/xml;charset=UTF-8

<oauth><error_description>Bad client credentials</error_description><error>invalid_client</error><,
```

Resource Server

There are two ways to validate a client token on custom web services. You can use the `ywebservices` template to generate a restful extension with a configuration that is already set up for integrating the resource server present in Platform.

You can also prepare your configuration manually:

Manual setup

```
<!-- load oauth2 definitions for resource server -->
<import resource="classpath*:oauth2-resource-spring.xml"/>
<!-- specify the filter change, the 'id' must be the same on web.xml filter -->
<bean id="oauth2SecurityFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterChair
  <constructor-arg>
    <list>
      <ref bean="springSecurityFilterChain"/>
    </list>
  </constructor-arg>
</bean>

<!-- spring security. the pattern will be the one specific of your application -->
<http pattern="/rest/**" create-session="never"
   entry-point-ref="oauthAuthenticationEntryPoint"
   access-decision-manager-ref="accessDecisionManager"
   xmlns="http://www.springframework.org/schema/security">
   <anonymous enabled="false"/>
   <intercept-url pattern="/rest/**" access="IS_AUTHENTICATED_FULLY" method="GET"/>
   <custom-filter ref="resourceServerFilter" before="PRE_AUTH_FILTER"/>
   <access-denied-handler ref="oauthAccessDeniedHandler"/>
</http>
```

Preventing Brute Force Attacks

Using the `oauth2.maxAuthenticationAttempts` property, you can set the maximum number of failed client authentication attempts. If the max number of failed authentication attempts is reached, the `oauth2` client is disabled. The property isn't enabled by default. You can set it in `local.properties`, as shown in the example:

```
oauth2.maxAuthenticationAttempts=10
```

The `oauth2.maxAuthenticationAttempts` property defines common configuration for all OAuth clients.

Once a client is disabled, it's not possible to access resources using an access token previously issued for this client. The client can also no longer be used in OAuth flows, which means that it's impossible to issue a new access token or refresh an existing one by using this client in the request.

For disabled clients, each subsequent authentication attempt results in a 401 HTTP error response similar to the following:

```
{"error":"unauthorized","error_description":"Authentication for clientId:'e2eTests' is disabled"}
```

You can re-enable disabled clients in the **Clients** tab under **OAuth** in Backoffice. Once your client is re-enabled, access tokens that are issued for this client can be used again if they haven't already expired.

Preventing Brute Force Attacks for Public Clients

The `oauth2.maxAuthenticationAttempts` configuration property is also applied for OAuth public clients. Public clients are not required to provide any password parameters in their authentication requests.

Once a public client is disabled, it's not possible to access resources using an access token previously issued for this client. The client can also no longer be used in OAuth flows, which means that it's impossible to issue a new access token or refresh an existing one by using this client in the request.

ℹ Note

Providing non-empty passwords in authentication requests for public clients causes authentication errors.

For information on prevention against user login brute force attacks, see User Account.

Disabling Client-side SSL Certificate Checks and Hostname Verification

If you want to switch to HTTPS for the token endpoint and other RESTful resources that our API offers, but still do not want to use an official server certificate, you need to disable the client-side checks. The following code shows you how this task can be accomplished.

ℹ Note

Keep in mind that this is to be used only during development and once a real certificate replaces the non-official certificate that ships with the SAP Commerce development package, you need to remove this code.

The `DummyHostnameVerifier` disables hostname checks and simply verifies all hostnames:

```
package de.hybris.platform.ycommercewebservices.test.groovy.webservicetests;
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;

public class DummyHostnameVerifier implements HostnameVerifier {
        @Override
        public boolean verify(String hostname, SSLSession session) {
                return true;
        }
}
```

The `DummyTrustManager` overrides all checks and trusts all clients, even if the certificate cannot be verified:

```
package de.hybris.platform.ycommercewebservices.test.groovy.webservicetests;

import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;

import javax.net.ssl.X509TrustManager;

public class DummyTrustManager implements X509TrustManager
{

        @Override
        public void checkClientTrusted(final X509Certificate[] arg0, final String arg1) throws Cert
        {
                //TODO unimplemented
        }

        @Override
        public void checkServerTrusted(final X509Certificate[] arg0, final String arg1) throws Cert
        {
                //TODO unimplemented
        }

        @Override
        public X509Certificate[] getAcceptedIssuers()
        {
                return null;
        }
}
```

Before you can create *<HTTPSUrlConnections>*, you need to modify your SSL context to use instances of the above classes.

<Applying DummyHostnameVerifier and DummyTrustManager>

```
def trustManager = new DummyTrustManager()
def hostnameVerifier = new DummyHostnameVerifier();
def sc = javax.net.ssl.SSLContext.getInstance("SSL");
sc.init(null, [trustManager]as X509TrustManager[], new java.security.SecureRandom());
javax.net.ssl.HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
javax.net.ssl.HttpsURLConnection.setDefaultHostnameVerifier(hostnameVerifier);
```

Roles

OAuth 2.0 defines the following roles:

- **Resource Owner**: An entity that can grant access to a protected resource. When the resource owner is a person, then it is called: end-user.
- **Resource Server**: The server that hosts the protected resources, capable of accepting and responding to protected resource requests using the access tokens.
- **Client**: An application making protected resource requests on behalf of the resource owner and with its authorization. The term client does not imply any particular implementation characteristics (for example, whether the application runs on a server, desktop, or any other particular device).
- **Authorization Server**: The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

  The authorization server is defined in the OAuth2 and the example resource server is configured in ycommercewebservices Extension and ywebservices Extension.

Flows

OAuth 2.0 comes with four flows. SAP Commerce supports all of them:

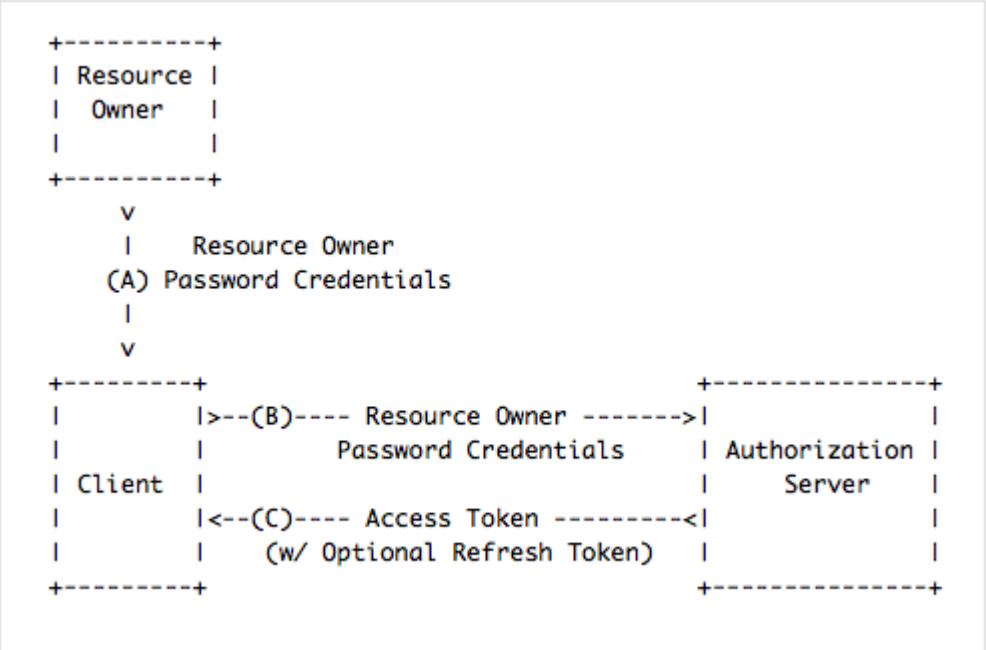
- As the **Resource Owner Password Flow** holds a user's *<username>* and *<password>*, it is less secure and not for third-party applications.
- The **Authorization Code Flow** is best to use if the web application can keep the*< client_secret>*.
- The **Implicit Flow** does not need any authorization tokens. As such, it is easier but less secure. JavaScript running within a browser is less trusted, and the refresh tokens are not issued. This is suited for the client-side web applications that need temporary access.
- The **Client Credentials Flow** gives the client access to the resources it owns.

Depending on your client application, you need to choose the appropriate flow. You also need to disable the other flows that you do not use.

Resource Owner Password Flow

This flow is somewhat similar to the basic authentication flow but it has a few benefits. It is typically used for trusted mobile applications, such as mobile Android or iOS applications. The flow includes sending the user's *<username>* and *<password>* to the token endpoint in exchange for an *<access_token>*. Replying with a refresh token is optional. The mobile *<client>* otherwise has to keep the username and password for long-lived access.

The flow needs a *<username>* and *<password>* for the *<access_token>*. However, keep in mind that the API provider provides access tokens combined with a*< refresh_token>*. The client therefore does not need to save the username and password, but it only has to pass this information on. The *<access_token>* and *<refresh_token>* need to be persisted locally, which is better than storing user credentials. The following diagram describes this flow.



Detailed description of the presented flow:

1. **Step (A)**: The *<client>* receives the *<username>* and *<password>*. In this step, the user enters this information directly into the client application. Note that users must have a way to identify the application as being the official application they can trust.

2. **Step (B)**: Next, the client application makes a request to the Authorization Server, for example the `/oauth/token` endpoint. There are two ways the *<client_id>* and *<client_secret>* can be sent along: either in a regular basic authentication request header, or as a part of the parameters passed in the request payload (that is, the request body). See the list of parameters to be passed:

    - *<client_id>* and *<client_secret>*: Either passed as parameters or as a basic authentication header. Basic authentication means that *<client_id>* and *<client_secret>* are treated as username and password, concatenated using a colon (:) and then *<Base64>* encoded. This value is then used as a part of the authorization request header, for example: `Authorization: Basic <Base64-encoded username:password>`

    - *<username>* and *<password>*: Credentials for the resource owner, the user's real credentials.

      > **Note**
      >
      > If One-Time Password for Customer Login is enabled and the token is issued for a Customer type user, then *<Token Id>* value is expected as *<username>* and *<Token Code>* value as *<password>*. For more information about One-Time Password for Customer Login, see One-Time Password.

    - *<grant_type>*: Needs to be set to *<password>* for this flow.

3. **Step (C)**: The authentication server returns the *<access_token>* with an optional *<refersh_token>*.

The following code sample describes this flow.

```
import com.google.appengine.api.urlfetch.*
import groovy.json.*
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'

if (!params.username)
{
        out << "No username given..."
        return
```

```
}

if (!params.password)
{
        out << "No password given..."
        return
}

def username = URLEncoder.encode(params.username, 'UTF-8')
def password = URLEncoder.encode(params.password, 'UTF-8')

HTTPResponse res
URL tokenURL = "http://localhost:9001/authorization/oauth/token".toURL()

if (params.basic)
{
//direct exchange of username and password for access token
def headers = [Authorization:"Basic ${"${client_id}:${client_secret}".toString().bytes.encodeBase64
res = tokenURL.post(deadline: 30, headers: headers, payload:"grant_type=password&username=${usernam
}
else
{
        //direct exchange of username and password for access token
        res = tokenURL.post(deadline: 30,  payload:"client_id=${client_id}&client_secret=${client_s
}

out << res.text
```

The above code is written in Groovy and uses the Gaelyk Web Framework. This is, however, only for demonstration purposes. Typically, the code would not be a part of the server-side web application, but it serves as an example of how the single request against the authorization server is performed. First, *<client_id>* and *<client_secret>* need to be known for the code issuing the request. In this case, *<username>* and *<password>* are passed as parameters. In a real scenario, the credentials would be entered into a form through the application user interface.

Depending on the basic parameter, we either pass the username and password in a HTTP Post request as the authorization header or as a part of the parameters in the body of the request. The above code simply prints the response, which shall be a JSON-formatted document like in the example below.

```
{"access_token":"f2288c2b-1d08-4aac-a210-d62c0901f915","token_type":"bearer","refresh_token":"3f945
```

Using CURL For Resource Owner Password Flow

During the development phase, you may want to quickly try out a RESTful request. If you run this request against an OAuth 2.0 protected resource, you need an *<access_token>*. Unfortunately, OAuth 2.0, just like basic authentication, is not supported by the browser. The easiest option is to use cURL, the command-line utility for HTTP requests in combination with the resource owner password flow.

The token endpoint needs to be HTTPS in the production environment, but during the development phase the following example should work fine.

```
curl -X POST -d "client_id=mobile_android&client_secret=secret&grant_type=password&username=demo&pa
```

This will result in a JSON response with the *<access_token>*.

```
{
                                        "access_token": "a503faf9-45b5-4fec-8334-337284a66ea4",
                                        "token_type": "bearer",
                                        "refresh_token": "486adfde-757b-4d37-81d7-446c2ec4bd91",
                                        "expires_in": 43199
                                        }
```

Next, if you want to access a protected resource, you have to pass the authorization header. The following code sample shows how to access the *<current user>* resource.

```
curl --header "Authorization: Bearer a503faf9-45b5-4fec-8334-337284a66ea4" http://localhost:9001/re
```

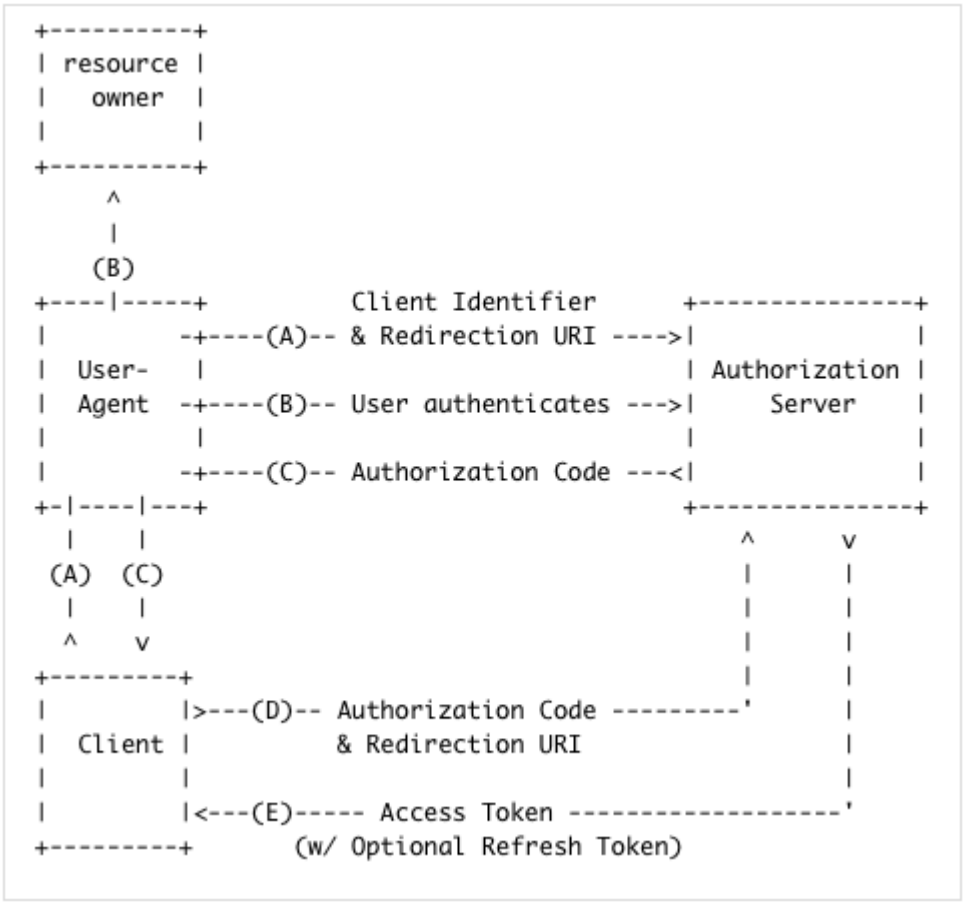
The response is presented in the next sample below.

```
{
                                        "uid": "demo",
                                        "name": "demo",
                                        "firstName": "Klaus",
                                        "lastName": "Demokunde"
                                        ...
                                        }
```

Authorization Code Flow Or Server-side Flow

The flow includes sending the client user by redirecting to the provider's login and authorization page. Next, it redirects back to the web application and passes an authorization code in the URL parameters. Then, it can be exchanged for *<access_token>* which needs to be passed in the HTTP request headers to obtain access to the user's data. With the *<access_token>* and information about its expiry, there is also a *<refresh_token>* issued. In the next step, a *<refresh_token>* can be exchanged against a new *<access_token>* for a long-lived access.

The following flow diagram describes this flow.



Detailed description of the presented flow:

1. **Step (A)**: Redirect the user to the authorization server (typical endpoint should be `/oauth/authorize`). All communication should occur through HTTPS so that *<client_id>* contained in the URL is safe. The following parameters have to be included:

- *<response_type>*: The value needs to be *<code>*, as we use the authorization code flow.
- *<client_id>*: For SAP Commerce web services, the OAuth 2.0 client has to be manually set up in the Spring Security OAuth2 XML configuration.
- *<client_secret>*: The *<secret>* for the *<client_id>*, and it needs to be on the server-side so that users cannot see it.
- *<redirect_uri>*: Optional, but recommended. When a user is redirected to the authorization endpoint, this value needs to be passed, and it has to match the server configuration settings.
- *<scopes>*: Used to allow different access levels.
- *<state>*: Optional, but recommended. Used to overcome CSRF (cross-site request forgery) attacks.

  The following code sample uses the described parameters.

```
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'http://localhost:8080/oauth2_callback'

def scopes = [
        'customer'
]

def state = new Random(System.currentTimeMillis()).nextInt().toString()
request.getSession(true).setAttribute('state', state)

redirect "http://localhost:9001/authorizationserver/oauth/authorize?client_id=${client_
```

2. **Step (B):** The user authenticates himself and is granted authorization. This can be a two-step process: if the user is not logged in, then he has to log in first and then provide access to the *<client>*.
3. **Step (C)**: The authorization server redirects back to the web application. The parameters passed in this redirect to the application are:
   - **code**: After the *<grant_type>* is set to *<code>*, the authorization code is accessible. Later, *<code>* is used to request the *<access_token>*.
   - **state**: The *<state>* specified in the first redirect to the authorization server. It should be checked with the session if the value is the same.
4. **Step (D)**: The web application verifies the redirect and exchanges the authorization code for an access token. The following code snippet is an example of how it works.

```
import com.google.appengine.api.urlfetch.*
import groovy.json.*
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'http://localhost:8080/oauth2_callback'

if (!params.code)
{
        out << "User denied access."
        return
}

if (!params.state)
{
        out << "State parameter missing, somethign is wrong..."
        return
}

if (!session)
{
```

```
                out << "No session, how weird!"
                return

        }

        if (params.state != session.state)
        {
                out << "State does not match! (${params.state} != ${session.state})"
                return
        }

        def code = params.code

        //exchange code for real oauth token
        URL tokenURL = "http://localhost:9001/authorizationserver/oauth/token".toURL()
        HTTPResponse res = tokenURL.post(deadline: 30, payload:"code=${code}&client_id=${client_id}&c

        out << res.text
```

The above code first verifies that the *<code>* and *<state>* parameters are there and then checks whether the current state is equal to the session's state. Then the final request is made to obtain the *<access_token>*. The parameters passed into the URL are:

- *<code>*: The obtained code.
- *<grant_type>*: Tells the token endpoint we want the access token. This is now set to *<authorization_code>*.
- *<redirect_uri>*: Used for consistency.
- *<client_id>* and *<client_secret>*: Used for identifying the client application.

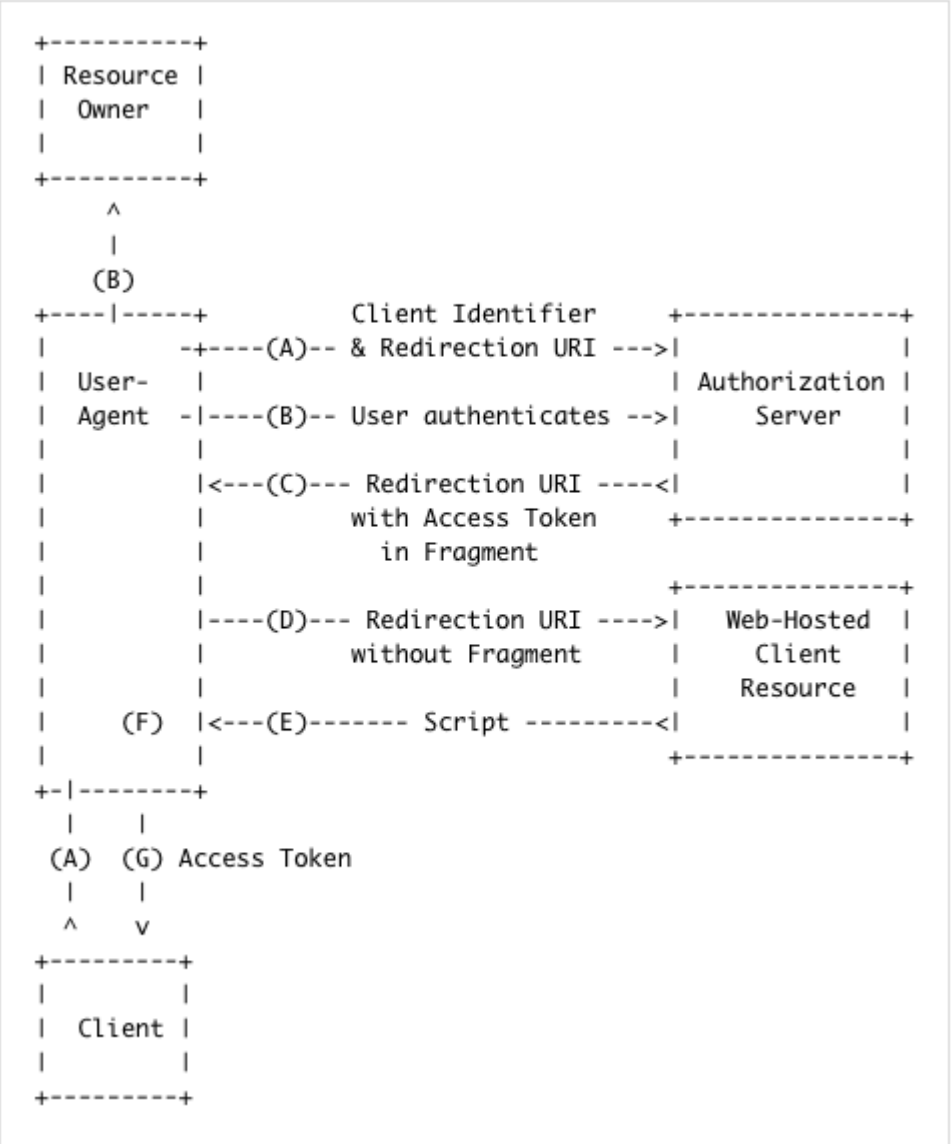
5. **Step (E)**: Eventually, access token is passed on to the client. The output looks like in the following sample.

```
{"access_token":"865b3ecf-54d3-413c-951e-fae0b00c906f","token_type":"bearer","refresh_token":
```

Implicit Flow Or Client-Side Flow

In the Implicit Flow (also called, Implicit Grant Flow), the required access token is sent back to the client application without the need of an authorization request token. This makes the flow easier, but also less secure. As the client application, which is typically JavaScript running within a browser, is less trusted, no refresh tokens for long-lived access are returned. You should use this flow for the client-side web applications (JavaScript clients) that need temporary access (for example, a few hours) to the user's data. Returning an access token to the JavaScript clients also means that your browser-based application needs to take special care. Take into account the XSS (Cross-Site Scripting) attacks that could leak the access token to other systems.

The following flow-diagram describes this flow.



Detailed description of the presented flow:

1. **Step (A)**: The user information is sent to the authorization server. Following parameters are sent by URL:
    - *<response_type>*: Tells the authorization server to respond directly with an access token. It needs to be set to *<token>*.
    - *<client_id>*: Identifies which client is requesting access for a user.
    - *<redirect_uri>*: Strongly recommended and needs to match the settings for the *<client_id>*.
    - *<scope>*: Optional, but currently not required for the SAP Commerce OCC Web Services, as there is a single *<customer>* scope.
    - *<state>*: Optional, but recommended to mitigate XSS (Cross Site Scripting) attacks.

      The following code sample uses the described parameters.

```
import java.net.URLEncoder

def client_id = 'client-side'
def redirect_uri = 'http://localhost:8080/oauth2_implicit_callback'

def scopes = [
        'customer'
]

def state = new Random(System.currentTimeMillis()).nextInt().toString()
```

```
        session.state = state

        redirect "http://localhost:9001/authorizationserver/oauth/authorize?client_id=${client_
```

2. **Step (B)**: Now, the user may log in and then be presented an authorization screen where they are asked to grant access to the specified scope.

3. **Step (C)**: The authorization server redirects back to the client web application. The *<access_token>* is part of this redirect URI, but in the *<#hash>* fragment of the URI. This makes it invisible for some server-side code, like the J2EE, which does not allow access to the hash fragment (as it is only intended for the client). At this point, nothing can be done for the redirection request, but as it hits the web application again at the *<redirect_uri>* you need to make sure the request will work.

   For the purpose of a demo client application, a controller endpoint is used that forwards directly to the HTML page. The remaining logic (which is, getting access to the *<access_token>* and verifying the state parameter) all happens in JavaScript. In the demo application we chose to extract the state from the user's session for which we need to route the redirect request through the controller.

   ```
   request.setAttribute('state', session.state)
   session.removeAttribute('state')
   forward '/WEB-INF/pages/oauth2_implicit_callback.gtpl'
   ```

4. **Step (D)**: Now, the user-agent (browser) follows the redirect and the client-side web application server responds with an HTML page that includes JavaScript to parse the hash fragment.

5. **Step (E)**: The following sample shows the code of the returned HMTL page.

   ```
   <!doctype html>
   <html>
     <head>
       <title>Oauth2 Implicit Callback Page</title>
     <head>
   <body>

   <div id="oauthParams" data-state=${request.getAttribute('state')}></div>

   <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
   <script src="/js/oauth.js"></script>

   </body>
   </html>
   ```

   Elements to note:

   - **state**: The state that was into the user's session in the controller is extracted and passed on by the J2EE HttpServletRequest as an attribute. This state is passed on to a *<data-*>* field of the *<oauthParams>* HMTL * * element. Later, it can be extracted and verified with the state arrived in the redirection URI.

   - **scripts**: In the sample, two script sources are included. One is the CDN-hosted jQuery, the second is our own oauth JavaScript.

6. *<Step (F)>*: Now, at the user-agent end, JavaScript will access the *<#hash>* fragment to extract the access token and compare the state value. The JavaScript resource looks like in the following sample.

   ```
   $(document).ready(function(){

   var oauthParams = {},
           queryString = location.hash.substring(1),
           regex = /([OAuth 2.0 (PSI 2 Update)^&=]+)=([OAuth 2.0 (PSI 2 Update)^&]*)/g,
           m,
           html='
   ';

   var state = $('#oauthParams').data('state');
   ```

```
        while (m = regex.exec(queryString))
        {
                oauthParams[decodeURIComponent(m[1])] = decodeURIComponent(m[2]);
        }

        console.log(oauthParams);

        for (pos in oauthParams)
                html += pos + '=' + oauthParams[pos] + '
        ';

        if (oauthParams.state == state)
                html += 'State matches, use the access_token!';
        else
                html += 'State does NOT match! Issue!';

        $('#oauthParams').html(html);

        });
```

The OAuth 2.0 parameters are part of the hash fragment, that is, the part of a URL after the *<#>* sign.

7. **Step (G)**: Finally, the *<access_token>* is passed on to the client.

Client Credentials Flow

The purpose of this flow is giving the client access to the resources it owns. It identifies the client and the OCC Web Services using the client credentials flow to restrict access to various calls.

The flow to obtain a client credentials token is very simple: you need to pass the *<client_id>* and *<client_secret>* parameters to the token endpoint. Keep in mind that in the production environment the token endpoint needs to use HTTPS. The following cURL sample provides an example.

```
curl -X POST -d "client_id=mobile_android&client_secret=secret&grant_type=client_credentials" http
```

The authorization server responds with the access token if the credentials are valid.

```
{
  "access_token": "b4dc1410-6b15-4a7c-bd22-1abea1f0fa3a",
  "token_type": "bearer",
  "expires_in": 42921
}
```

The token identifies, authenticates, and authorizes the client.

Refreshing an Expired Access Token

There is a need to refresh the issued *<access_token>*. Without refreshing these tokens, a *<client>* must remember user credentials, which is a less secure approach. Providing access tokens and refreshing the tokens means that the client only has to remember the tokens. The following code describes how to issue refresh tokens.

```
import com.google.appengine.api.urlfetch.*
import groovy.json.*
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'http://localhost:8080/oauth2_callback'

if (!params.token)
{
        out << "No refresh token given..."
        return
```

```
}

def token = params.token

//exchange code for real oauth token
URL tokenURL = "http://localhost:9001/authorizationserver/oauth/token".toURL()
HTTPResponse res = tokenURL.post(deadline: 30, payload:"refresh_token=${token}&client_id=${client_

out << res.text
```

It is a single request and response. In real life, the token endpoint would only be accessible by HTTPS. Parameters that must be sent to the `/oauth/token` endpoint to refresh a token are:

- *<client_id>* and *<client_secret>*: As the refresh request is sent from a server, keep the *<client_secret>* really secret. Both parameters need to be sent along.
- *<redirect_uri>*: Required, although not used here. The server responds with a JSON file with the token information directly. If you have a *<redirect_uri>* configured for the client, you need to pass this on for the refresh request.
- *<grant_type>*: It needs to be *<refresh_token>*, indicating the exchange of a refresh token for a new *<access_token>* and also a refresh token for the next time.
- *<refresh_token>*: For the client; the refresh token passed as the token request parameter and then into the refresh token request.

The server responds with a JSON response like in the following example.

```
{"access_token":"b70aeeb9-9095-4899-9480-2dbf432746ac","token_type":"bearer","refresh_token":"61f0
```

Configuring OAuth Clients

Follow the steps to configure OAuth clients using the Backoffice Administration Cockpit or ImpEx.

Context

OAuth client is an application making protected resource requests on behalf of the resource owner and with its authorization. Each client application, such as mobile android client, mobile iOS client, which should have access to resources need to be registered in OAuth authorization server.

Adding OAuth Clients

Context

Follow the steps listed below to add a new OAuth client.

Procedure

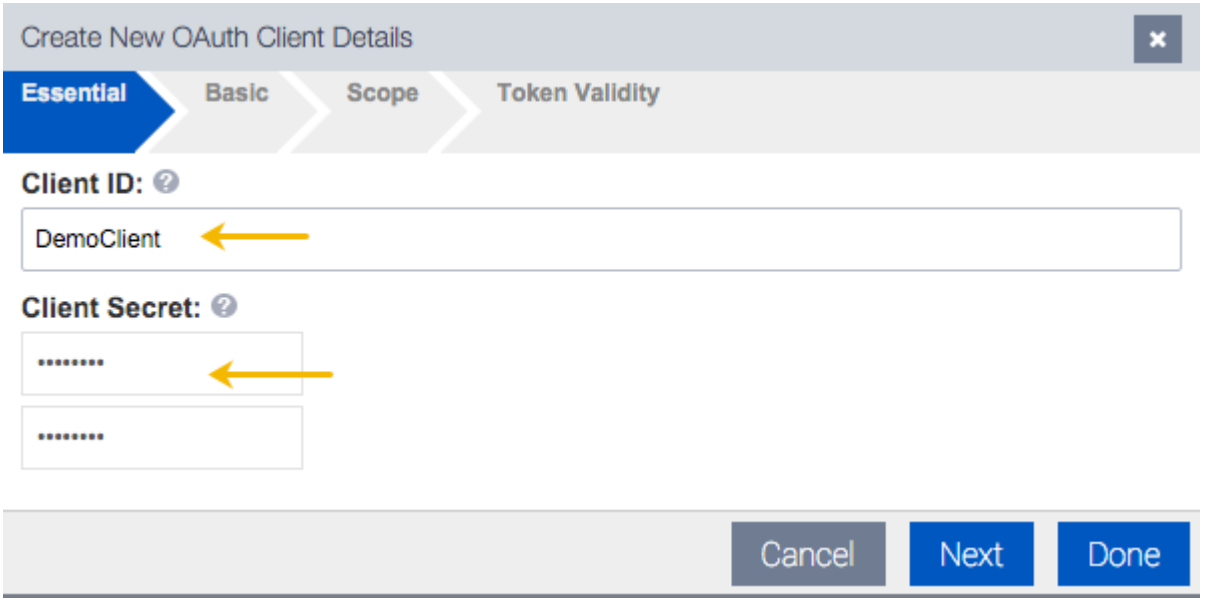
1. Navigate to the OAuth Clients node (**System** > **OAuth** > **OAuth Clients**).

   On the left you see a list of already created OAuth Clients. Here you can easily manage the clients.

2. In order to add a new OAuth client click the **OAuth Client Details** field.

   The editor appears.

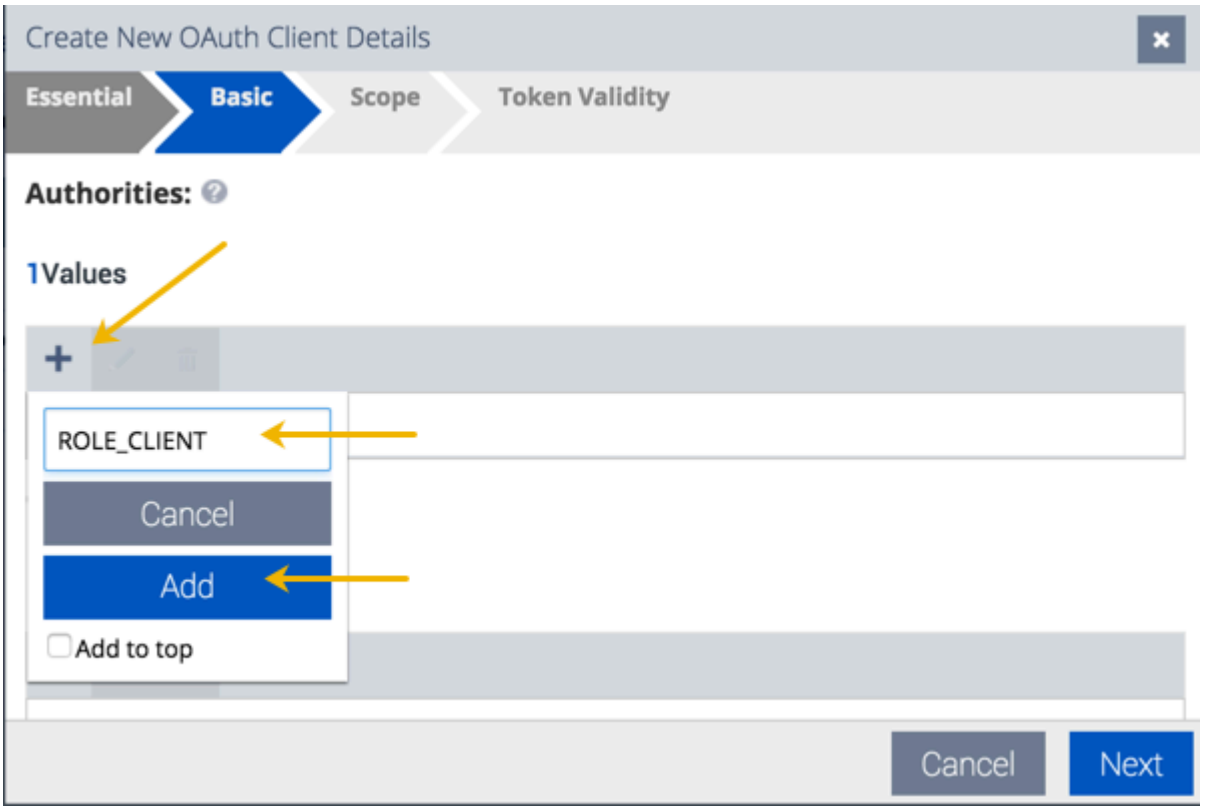
3. In the **Essential** section, provide*< Client ID>* and *<Client Secret >*(an encoded password).



4. In the Basic section, provide the following values.

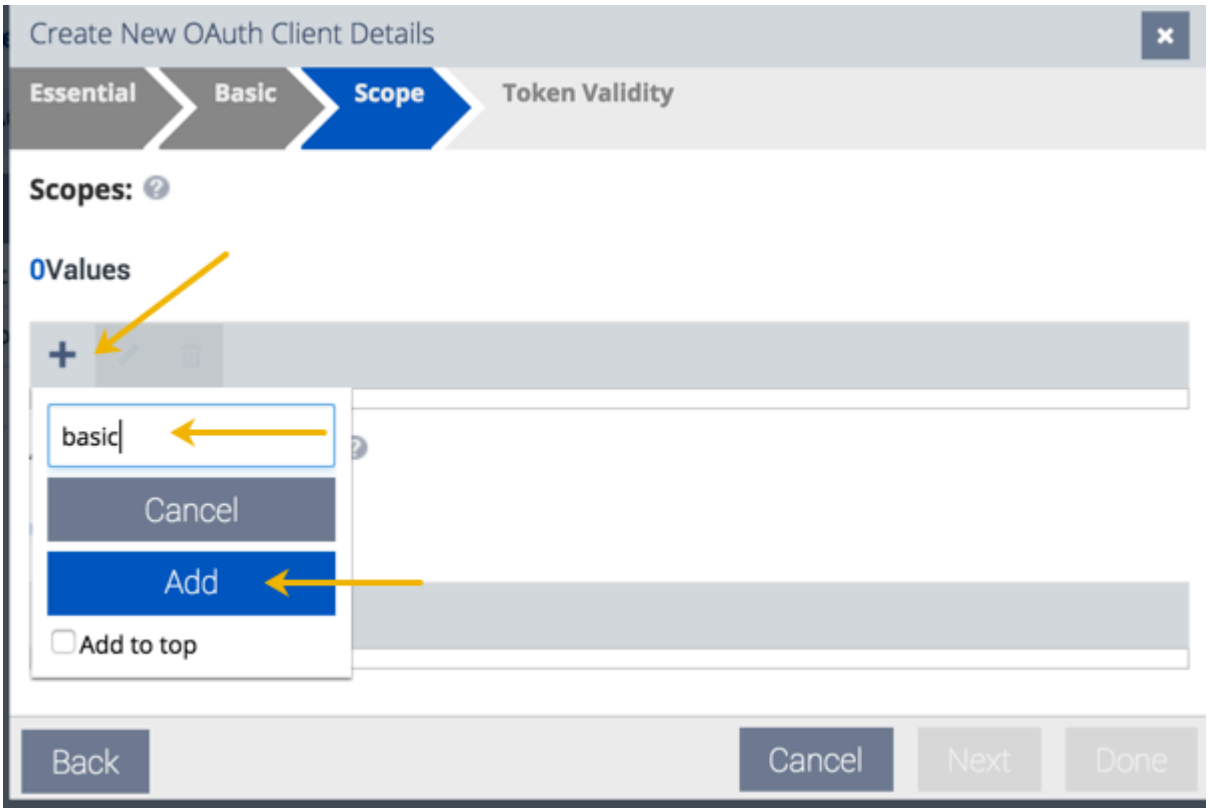
   - **Authorities**: Authorities - List of authorities (roles) that are granted to the OAuth client such as ROLE_CLIENT, ROLE_TRUSTED_CLIENT
   - **Authorized Grant Types**: Grant types available for client for example: refresh_token, password, authorization_code, client_credentials Grant types decide, which getting token flow can be used by this client. Grant types supported by the authorization server can be configured in <authorization-server> element.
   - **Resource Ids** : The resource identifiers to which this client can be granted access.
   - **Redirect Uri** : Redirect Uri allowed for the client in Authorization Code Flow and Implicit Flow

   To enter a particular value, use the **plus** sign and confirm with **Add**.

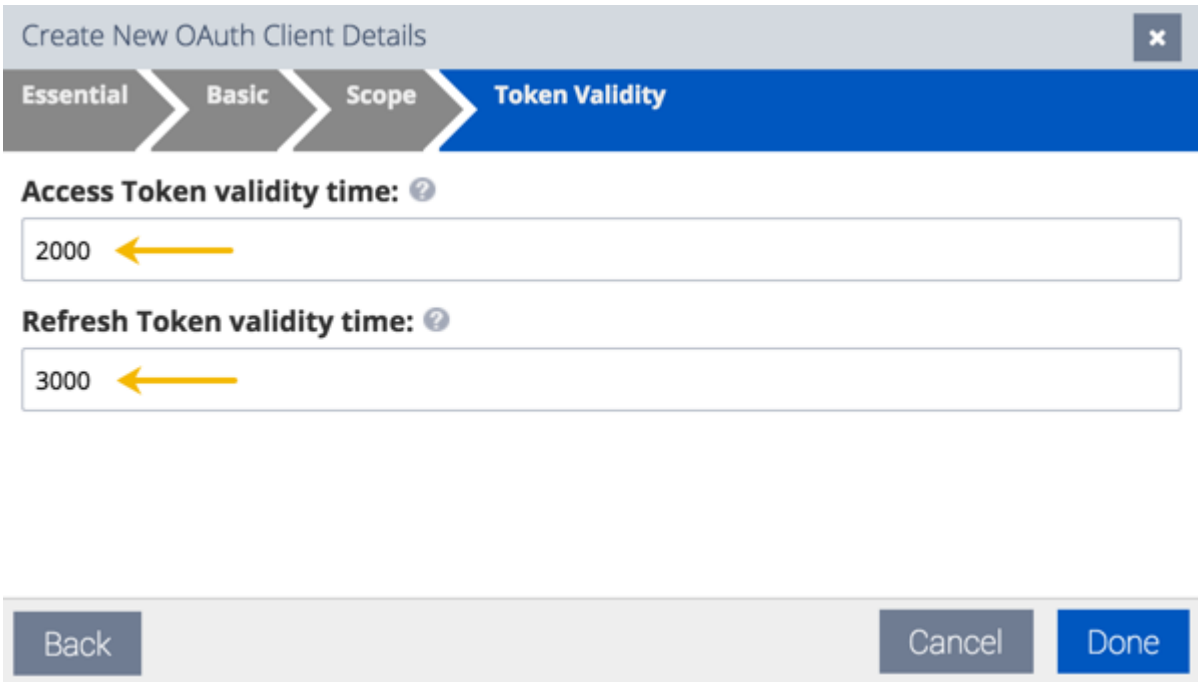


5. In the **Scope** section, provide the following values:

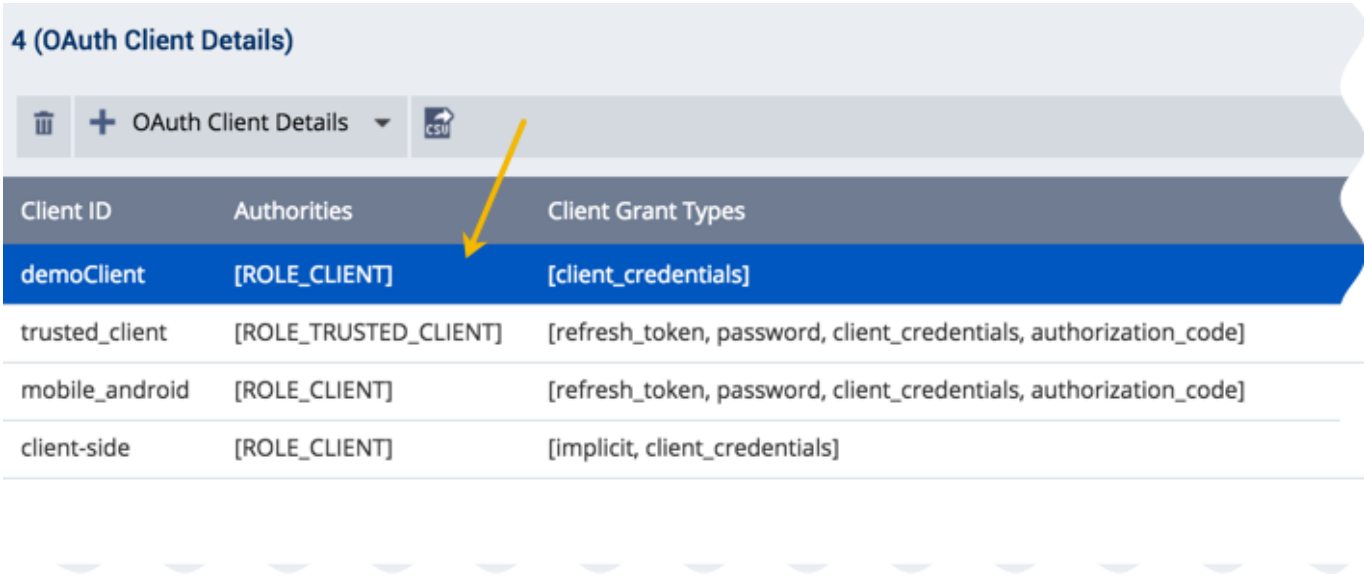
   - **Scopes**: List of scopes to which the client is limited.
   - **Auto Approve Scopes**: Scopes the Client does not need the User approval for.



6. In the **Token Validity** section, provide the following values:
    - **Access Token Validity Seconds**: The access token validity period in seconds.
    - **Refresh Token Validity Seconds**: The refresh token validity period in seconds.



7. A new client is added to the list.



Editing OAuth Clients

Context

Follow the steps listed below to edit an already existing OAuth client.

Procedure

1. Navigate to System > OAuth > OAuth Clients.

   On the left you see a list of already created OAuth Clients. Here you can easily manage the clients.
2. Select the client you want to edit.
3. Click Save after selected values.

Removing OAuth Clients

Context

Follow the steps listed below to remove an already existing OAuth client.

Procedure

1. Navigate to System > OAuth > OAuth Clients.

   On the left you see a list of already created OAuth Clients. Here you can easily manage the clients.
2. Select the client you want to remove.
3. Click the **bin** icon to remove the row. A dialog is displayed for you to confirm the action.

Defining OAuth Clients in an ImpEx File

OAuth clients data are stored in the database using the `OAuthClientDetailsModel`, so it can be also created using an ImpEx file.

Procedure

1. Define the OAuth client in the `projectdata.impex`file.

```
INSERT_UPDATE OAuthClientDetails;clientId[unique=true]    ;resourceIds       ;scope        ;a
                                 ;client-side             ;hybris            ;basic        ;i
                                 ;mobile_android          ;hybris            ;basic        ;a
```

2. **i Note**

   When defining the clients remember to assign either the ROLE_CLIENT or ROLE_TRUSTED_CLIENT to them, because these roles allow client access to `ycommercewebservices`. Be careful with the ROLE_TRUSTED_CLIENT because it has got specific, extended rights.

If you want to add the ROLE_TRUSTED_CLIENT, you will also need to define it in the `projectdata.impex` file.

```
INSERT_UPDATE OAuthClientDetails;clientId[unique=true]    ;resourceIds       ;scope        ;a
                                 ;trusted_client          ;hybris            ;extended     ;a
```

Platform as an OpenId Connect Identity Provider

With OpenId Connect, software developers don't have to bother with managing or storing passwords to authenticate end users. You can use this technology to authenticate client application users to access external systems by using Platform as the identity provider.

To use this function, you have to configure Platform as an ID provider, and register it in the external system.

The example below uses the OpenID Connect Implicit Grant flow. First, your application requests Platform for an ID token. Platform returns the requested ID token, and the application sends it to the external system. In return, the external system sends back the access token that allows you to access specific resources from the external system.

Configuring Platform as an ID Provider

OpenId Connect is an identity layer based on the OAuth 2.0 authorization framework. For that reason, you have to configure the OAuth Client role. For details, see OAuth 2.0.

Here is an example of how you can configure the client role. The client used in all examples has id `client-side`.

exampleOpenIDClientDetails configuration impex

```
INSERT_UPDATE OpenIDClientDetails;clientId[unique=true]    ;resourceIds       ;scope        ;author
                                 ;client-side              ;hybris            ;basic,email,profile,
```

Here is an example of how you can configure the external system scopes.

```
insert_update OpenIDExternalScopes;code[unique=true];clientDetailsId(clientId)[allownull=true,force
                                  ;editor             ;client-side
                                  ;reader             ;client-side
                                  ;admin              ;client-side
```

KeyStore

To use the OpenID feature, your SAP Commerce instance must have a Java KeyStore repository containing both public and private keys. Platform is shipped with a demo keystore but it is only for development use. You can generate a new keystore using the Java keytool. We recommend using the RSA algorithm for encryption.

For example, this command generates a 2048-bit RSA keystore.

```
>  keytool -genkey -keyalg RSA -alias alias -keystore keystore.jks -keysize 2048
```

The password is stored in the properties.

Properties

Property	Definition
`oauth2.idTokenValiditySeconds=43200`	Id_token validity in seconds.
`oauth2.client-side.kid=test1`	The id (alias) of the public/private key used by a particular client for signing id tokens.
`oauth2.client-side.keystore.location=/security/keystore.jks`	Location of the KeyStore for a particular client id.
`oauth2.client-side.keystore.password=nimda123`	Password for the given KeyStore.
`oauth2.algorithm=RS256`	The algorithm used in the KeyStore; the same algorithm should be used for signing id tokens; the default implementation is RS256 and shouldn't be changed without having an additional implementation of another algorithm.

Configuring the ID Provider in an External System

You have to configure Platform as an ID provider in your external system. For that reason, refer to the documentation of your external system.

Implicit Grant Flow Requests

First, request an `id_token` from the external provider (Platform).

```
GET https://EC_IDP_URL/authorize/?
    response_type=id_token token
    &client_id=EXTERNAL_CLIENT_ID
    &redirect_uri=REDIRECT_URI_TO_WEB_PAGE
    &scope=SCOPE
    &nonce=NONCE
    &state=WEB_STATE
```

It should redirect you to the external page and finally redirect to the original page. The `id_token` should be in the URL of the web page.

Example URL:

```
GET https://localhost:9002/authorizationserver/oauth/authorize?response_type=token id_token&client_
```

The redirected URL contains the `id_token` value.

```
http://MY_APPLICATION/ #access_token=cfc4af6d-b38b-4152-a923-9af8de6c7f33%26token_type=bearer%26sta
```

The encoded `id_token` is:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6InRlc3QxIn0.eyJzdWIiOiJlZGl0b3IiLCJzY29wZSI6WyJvcGVuaWQ
```

After decoding the `id_token`, you get:

```
HEADER:ALGORITHM & TOKEN TYPE
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "test1"
}
PAYLOAD:DATA
{
  "sub": "editor",
  "scope": [
    "openid",
    "hybris.product_read_unpublished"
  ],
  "iss": "ec",
  "state": "3",
  "exp": 1501849757,
  "nonce": "5",
  "iat": 1501846157
}
```

Now exchange the `id_token` for the `access_token`:

```
GET https://external_system.io/oauth2/v1/authorize/?
    &client_id=EXTERNAL_SYSTEM_CLIENT_ID
    &hybris_id_provider=ID_PROVIDER_REGISTERED_IN_EXTERNAL_SYSTEM
    &id_token_hint=ID_TOKEN_FROM_EXTERNAL_PROVIDER
    &nonce=NONCE
    &redirect_uri=REDIRECT_URI_TO_WEB_PAGE
    &response_type=token
    &state=WEB_STATE
```

It should first redirect you to the external system OAuth, then to the external provider (Platform), and finally to the original page. The `access_token` should be in the URL of the web page.

Exchange the `id_token` for the `access_token`.

```
GET https://external_system.io/oauth2/v1/authorize/?client_id=ZwXU5ktg3TEYdZnDjrf4SCZjycO9KYd4&id_t
```

The redirected URL contains this `access_token` value.

```
http://MY_APPLICATION/#token_type=Bearer&access_token=022-de34d0e4-0557-4c18-b8ef-7c8544e8d6b9&expi
```

OpenID Connect Discovery

OpenID Connect Discovery enables clients to verify the identity of the end user based on the authentication performed by SAP Commerce.

OpenID Connect Discovery is based on two endpoints provided by SAP Commerce.

The Discovery endpoint is available at `https://{server}:9002/authorizationserver/.well-known/openid-configuration`. The following is an example response from the Discovery endpoint:

```
{
                  "issuer" : "https://myserver.sap.corp:9002/authorizationserver",
                  "authorization_endpoint" : "https://myserver.sap.corp:9002/authorizationserver/oa
                  "token_endpoint" : "https://myserver.sap.corp:9002/authorizationserver/oauth/toke
                  "jwks_uri" : "https://myserver.sap.corp:9002/authorizationserver/.well-known/jwks
                  "token_endpoint_auth_methods_supported" : [ "client_secret_post", "client_secret_
                  "subject_types_supported" : [ "public" ],
                  "response_types_supported" : [ "code", "code id_token", "id_token", "token id_tok
                  "scopes_supported" : [ "openid", "email", "groups" ],
                  "id_token_signing_alg_values_supported" : [ "RS256" ]
                }
```

The JWKS endpoint is available at `https://{server}:9002/authorizationserver/.well-known/jwks.json`. The following is an example response from the JWKS endpoint:

```
{
  "keys" : [ {
    "kty" : "RSA",
    "use" : "sig",
    "kid" : "kyma",
    "alg" : "RS256",
    "n" : "ALyXGDrQ3QZcugzjcJDPsVsK7ORTNHxnJIEt0ANM_tPxzf5wd7lFaZ7xjbL2rZaNICxT7MqfY-euvX1PAsXkby8\
    "e" : "AQAB"
  } ]
}
```

The following is an example configuration:

Property	Example Value	Info
`oauth2.kyma.algorithm`	RS256	Defaults to RS256
`oauth2.kyma.responseTypes`	`code`, `code id_token`, `id_token`, `token id_token`	Defaults to `code`, `code id_token`, `id_token`, `token id_token`
`oauth2.kyma.kid`	`kyma`	**kid** stands for **key id** (alias) for the kyma keytore
`oauth2.kyma.keystore.location`	`/security/keystore.jks`	Kyma keystore location
`oauth2.kyma.keystore.password`	`nimda123`	
`oauth2.kyma.public.address`	`http://www.example.com`	Replaces `myserver.sap.corp` from the example Discovery endpoint response with chosen address

To enable configuration per client, you can access endpoints per configuration/key:

- `https://{server}:9002/authorizationserver/.well-known/openid-configuration?client_id={id}`
- `https://{server}:9002/authorizationserver/.well-known/ jwks.json ?client_id={id}`

For example:

- `https://{server}:9002/authorizationserver/.well-known/openid-configuration?client_id=kyma`
- `https://{server}:9002/authorizationserver/.well-known/ jwks.json ?client_id=kyma`

Token Revocation

You can use the `/revoke` endpoint in the `oauth2` extension to revoke issued tokens.

This endpoint conforms with the RFC-7009 requirements described at https://tools.ietf.org/html/rfc7009 .

You don't need any additional authorization to revoke tokens - you are authorized with a token you are about to revoke.

The example tokens used in this topic are:

- access token d7689e7c-957e-46ea-949b-c39afb1c9935
- refresh token 00d275c4-0daa-4876-8a3b-4aa731f3e13a

To revoke the access token, call the `/revoke` endpoint with a POST method, and pass the access token in the `token` parameter:

```
-X POST -H https://localhost:9002/authorizationserver/oauth/revoke -d “token=d7689e7c-957e-46ea-949
```

The response you get is:

```
HTTP/1.1 200
Content-Length: 0
```

You can revoke an access or a refresh token by passing it with a hint indicating the token type. To do it, add to the request the `token_type_hint` parameter set to `access_token` or `refresh_token` respectively:

```
curl -X POST -H https://localhost:9002/authorizationserver/oauth/revoke -d “token=00d275c4-0daa-487
```

The response you get is:

```
HTTP/1.1 200
Content-Length: 0
```

A request with `token_type_hint` revokes a token of the specified type. If you pass, for example, an access token and set the hint to `refresh_token`, then the token doesn't get revoked.

Setting `token_type_hint` is optional. If you don't specify it, then Platform first tries to revoke your token as an access token. If your token isn't an access token, Platform tries to revoke it as a refresh token.

If you specify an unsupported token type, then you get the following error message:

`“unsupported_token_type: The authorization server does not support the revocation of the presented token type. That is, the client tried to revoke an access token on a server not supporting this feature”`

If you specify an invalid token in the `token` parameter, you don't receive any error message. The request returns the 200 OK status response.

Configuring customTokenGranter for External IdPs

Enhance user experience by enabling additional login methods through external IdPs.

Context

provides an alias `customTokenGranter` bean as part of the Spring Security OAuth Authorization Server. The baseline implementation, `DefaultCustomTokenGranter`, accepts the "custom" grant type and, in the default configuration, throws an exception stating that there is no implementation. Replace it with your own implementation of the `TokenGranter` interface to integrate external IdPs with SAP Commerce Cloud, composable storefront or any of your custom storefronts. By using external IdPs, you can expand the list of available login methods, allowing your customers to log into your application through already authenticated accounts without the need of going through the process of creating new ones from scratch. For more information on `TokenGranter`, see TokenGranter in Spring documentation.

Note

You can only use one `customTokenGranter` per instance. To integrate multiple IdPs with your storefront, ensure that your custom implementation of `customTokenGranter` supports such configuration.

Procedure

1. Create your custom implementation of `customTokenGranter` to override the existing configuration of the `TokenGranter` interface:

```
<alias name="specificCustomTokenGranter" alias="customTokenGranter" />
<bean id="specificCustomTokenGranter" ... />
```

2. In your custom implementation of `customTokenGranter`, define what information the client needs to pass into the token request. This includes custom or specific fields needed by a specific IdP.
3. Ensure that the external IdP client passes the required information that is defined in `customTokenGranter` as part of the token request into the `oauth2` extension. The IdP client needs to use the "custom" grant type that routes its requests to the `customTokenGranter` implementation.
4. Perform any necessary post-validation processing, such as logging, auditing, or updating user sessions, before returning the OAuth token to the client.
5. Once the OAuth token is generated and returned to the client, it can be used for subsequent requests to access protected resources. Ensure that your OAuth server validates these tokens for each request to ensure secure access to resources.

Ordering, Payment and Pricing Standards

Ordering, payment, and pricing are processes that depend on many factors. The default implementations of strategies supporting those processes shipped with Platform provide for general business cases. However, they are extensible and granular in architecture, which allows you to replace them with your own implementations that fully support your specific business cases.

The topics covered include:

Order Framework Extensibility

The Order Framework is delivered with SAP Commerce. The framework focuses on handling subtypes of `AbstractOrder`, for example Orders and Carts.

Price, Tax, and Discount Calculation

The `europe1` extension handles all price, tax, and discount calculations in SAP Commerce.

Extensible Cart Calculation

The extensible cart calculation is a solution based on service layer that can replace the default implementation of PriceFactory (Europe1PriceFactory). It provides an API and default implementations of the services and strategies for resolving prices, discounts, taxes, and delivery and payment costs used in cart (re)calculation.

Channel Specific Pricing

With Channel-Specific Pricing, you can set up different pricing models depending on how a user is accessing your store. Here you can find how to configure, use, and extend the SAP Commerce functionality to provide channel-specific prices for products.

Ordering Process

SAP Commerce has a built-in ordering process that automatically handles calculation of prices, taxes, and discounts for orders. The focus lies on the creation, calculation, recalculation, and lifetime of the carts and orders. Here you will find an overview of the phases of the ordering process and the technical processes in the background.

Payment Transaction and Delivery Mode Handling

SAP Commerce offers built-in support for management of fees for delivery of orders and payment methods. When a set-up of payment and delivery costs is done, the costs are automatically calculated and added to an order. SAP Commerce uses the best-matching elements of the set-up in terms of currency, user location and so on.

Commerce Quotes Items

Commerce Quotes enables buyers to create quotes and negotiate the final price of an order using the storefront. Learn about the items included in platform core and platform services that enable Commerce Quotes.

Related Information

deliveryzone Extension
Order Framework Extensibility
Price, Tax, and Discount Calculation
Channel Specific Pricing
Ordering Process
Payment Transaction and Delivery Mode Handling
Commerce Quotes Items

Order Framework Extensibility

The Order Framework is delivered with SAP Commerce. The framework focuses on handling subtypes of `AbstractOrder`, for example Orders and Carts.

The Order Framework provides several different ways for extending your system by relying on lower-level components that implement particular features responsible for the whole functionality. Additionally, you can also add new interceptors to introduce custom logic for order processing.

The order framework provides basic functionality for Carts and Order instances management:

- Cloning order instances
- Caving order instances
- Cloning with target type changing
- Managing order/cart entries

Order Framework Architecture

The Order Framework architecture is designed to provide two level separation in the framework - Separation of Concerns and Separation of `AbstractOrder` type.

1. **Separation of Concerns**: The main areas of the Order/Cart services are defined and encapsulated in the separate spring beans. Developers can exchange one of those beans with their customized implementations. Within those areas, you can find: order/cart cloning, order/carts saving, data fetching (DAO), order placement, entries management. Each of them is represented by a strategy, DAO, or a service.
2. **Separation of AbstractOrder Type**: The main interface `AbstractOrderService` is a template interface.

```
interface AbstractOrderService<O extends AbstractOrderModel, E extends AbstractOrderEntryMode
```

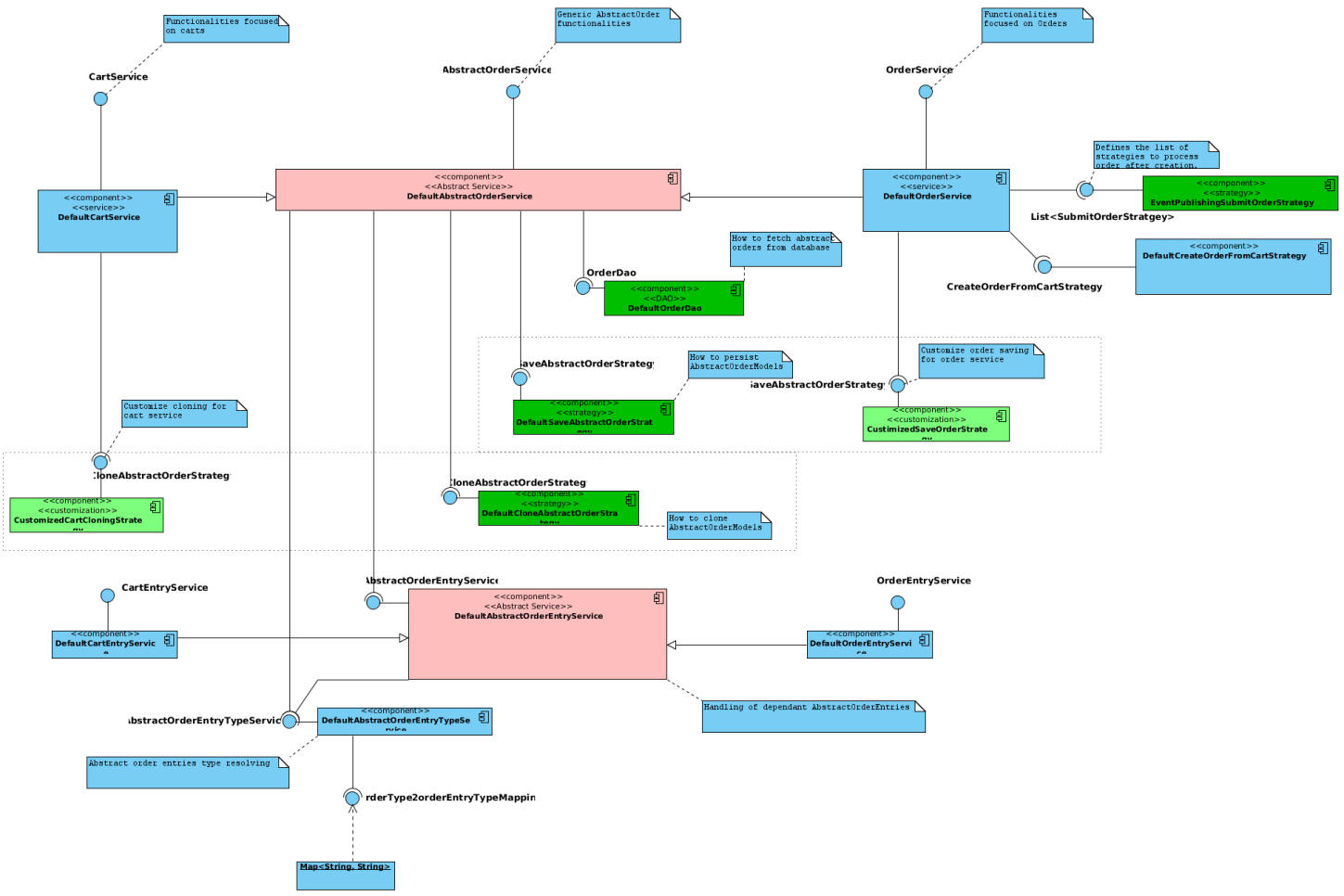
By extending it for a specific order type (`OrderModel`, `CartModel`) and order entry type (`OrderEntryModel`, `CartEntryModel`) you can obtain a set of its contract methods typed for the target order type. This is exactly how the `OrderService` and `CartService` are implemented. Apart from their own type dedicated logic, they share common functionality through the common `AbstractOrder` part. This separation is also reflected by the default implementations. There is:

- **Generic implementation**: For AbstractOrder type, there is `DefaultAbstractOrderService`.
- **More specific implementation**: For Carts, there is `DefaultCartService`. For Orders, there is `DefaultOrderService`.

⚠ Caution

SAP Commerce uses a specialised event that is sent by the enhanced `DefaultOrderService` class to monitor the use of different order metrics. To ensure that the data is sent and interpreted correctly, do not modify `DefaultOrderService` in your projects.

As you can see on the component diagram, the framework is based on the main service - `DefaultAbstractOrderService` - that binds all the related beans into one ecosystem.



`OrderService` and `CartService` provide main touch points for several basic business use cases, for example: order placement, obtaining session cart, order, or cart cloning. Those diversified cases involve many separated concerns. With the spring integration, the framework is decomposed into the smaller components, strategies, and data access object. This approach enables you to replace any of the framework components with another one that fits best into your order framework definition.

AbstractOrderService

The `AbstractOrderService` is the main component for the Order Framework extensibility. It delegates the particular method calls to the dedicated strategies, DAOs, and subservices.

The `AbstractOrderService` joins all the common functionalities that are applicable for all of the `AbstractOrder` subtypes in the system (for example, Order or Cart). Here, you can find methods to find order instances, order entries instances, but also to add new entries. Such functionality is common for any type of order or cart. But you can't find here the methods for order placement or session cart management as `AbstractOrder` type is not the type those methods belong to.

The following table groups the methods into the corresponding beans that handle them. This would give you the idea which parts of the main and shared functionality belong to which dependent spring bean's responsibilities.

AbstractOrderService Methods	Responsible Spring Bean	Spring Bean Role
`addNewEntry(...)`	`abstractOrderEntryService` `abstractOrderEntryTypeService`	Resolve the proper type of the entry for the given order instance. Create the order entry.
`clone(...)`	`cloneAbstractOrderStrategy` for its purpose, you can obtain the set of the functionalities for the order type of your choice. The order framework comes with two type-dedicated services:	Performs deep cloning of the target AbstractOrder instance. Uses cloning context to preset or skip attributes.
`getEntryForNumber(final O order, final int number)` `getEntriesForNumber(final O order, final int start, final int end)` `getEntriesForProduct`	`orderDao`	Constructs flexible search queries the persistence layer.
`saveOrder(...)`	`saveAbstractOrderStrategy`	Responsible for persisting order or cart models.

By extending the `AbstractOrderService` for a specific type and extending `DefaultAbstractOrderServiceOrderService` and `CartService`.

The following code snippet shows the spring configuration of the `AbstractOrderService`:

```
                        for its purpose, you can obtain the set of the functionalities for the orde
For Order and Cart there are services available out of the box. -->
<bean id="abstractOrderService" class="de.hybris.platform.order.impl.DefaultAbstractOrderService" a
                                parent="abstractBusinessService" >
        <property name="orderDao" ref="orderDao"/>
        <property name="abstractOrderEntryService" ref="abstractOrderEntryService"/>
        <property name="abstractOrderEntryTypeService" ref="abstractOrderEntryTypeService"/>
        <property name="saveAbstractOrderStrategy" ref="saveAbstractOrderStrategy"/>
        <property name="cloneAbstractOrderStrategy" ref="cloneAbstractOrderStrategy"/>
</bean>
```

OrderService

The `OrderService` extends the `AbstractOrderService`.

```
public interface OrderService extends AbstractOrderService<OrderModel, OrderEntryModel>
```

Its default implementation extends `DefaultAbstractOrderService`.

```
public class DefaultOrderService extends DefaultAbstractOrderService<OrderModel, OrderEntryModel>
                                 implements OrderService
```

It means that the order service can do whatever the `AbstractOrderService` defines, but it does it for:

- `OrderModel`: As an order type.
- `OrderEntryModel`: As an entry type.

The spring configuration enables you to inject one of the dependent beans for the `orderService` purpose only. If you want the order saving to work differently for the `orderService` than it works for the generic service, you need to redeclare the corresponding strategy in the spring configuration for the `orderService`.

The following spring configuration snippet represents the scenario.

```
<alias alias="orderService" name="defaultOrderService"/>
<bean id="defaultOrderService" class="de.hybris.platform.order.impl.DefaultOrderService"
                               parent="abstractOrderService" >
                <property name="createOrderFromCartStrategy" ref="createOrderFromCartStrategy"/>
                <property name="submitOrderStrategies">
                        <list>
                                <!-- implementation that sends SubmitOrderEvent -->
                                <!-- <ref bean="eventPublishingSubmitOrderStrategy"/> -->
                        </list>
                </property>
 </list>
 </property>
</bean>
```

In addition to the contract of the `AbstractOrderService`, `OrderService` defines also function strictly reserved for the service: order submission.

```
/**
 * Create the order for the given <code>cart</code>. This method focuses on creating an {@link Orde
 * instance from the given {@link CartModel} instance. The order instance remains unsaved
 * and not calculated.</br> This method does nothing with the cart member attributes (addresses, pa
 * It also leaves the target cart untouched.
 *
 * If you want to calculate cart or order, use {@link CalculationService}.
 *
 *
 * @param cart
 *           the target {@link CartModel}
 * @return a non persisted {@link OrderModel}
 * @throws InvalidCartException
 *            if the cart is invalid according to the used {@link CartValidator}.
 */
OrderModel createOrderFromCart(CartModel cart) throws InvalidCartException;
```

The method call is delegated to `CreateOrderFromCartStrategy`, replacing deprecated `PlaceOrderStrategy`, which defines the process of creating an actual order instance out of a cart instance. You can define your own strategy here in order to represent your business process.

ℹ Note

Some of the deprecated logic from `PlaceOrderStrategy` is now taken care by the interceptors dedicated for Order. The `DefaultOrderPrepareInterceptor` takes care of the contract members cloning: address and payment info.

For additional functionality focused on `OrderModel` type, you can extend the `OrderService` interface or `DefaultOrderService` implementation.

CartService

The `CartService` extends the `AbstractOrderService`.

```
public interface CartService extends AbstractOrderService<CartModel, CartEntryModel>
```

Its default implementation extends `DefaultAbstractOrderService`.

```
public class DefaultCartService extends DefaultAbstractOrderService<CartModel, CartEntryModel>
                                implements CartService
```

It means that the cart service can do whatever the `AbstractOrderService` defines, but it does it for `CartModel` as order type and `CartEntryModel` as entry type.

The spring configuration enables you to inject one of the dependent beans for the `cartService` purpose only. So, if you want the order cloning to work differently for the `cartService` than it works for the generic service, redeclare the corresponding strategy in the spring configuration for the `cartService`. The following spring configuration code example represents the scenario.

```
<alias alias="cartService" name="defaultCartService"/>
<bean id="defaultCartService" class="de.hybris.platform.order.impl.DefaultCartService" parent="abst
                              >
        <property name="cartFactory" ref="cartFactory"/>
        <property name="cloneAbstractOrderStrategy" ref="customizedCartCloningStrategy"/>
</bean>
```

For additional functionality focused on `CartModel` type, you can extend the `CartService` interface, or `DefaultCartService` implementation.

Common Functional Areas of Order Framework

See the common functional areas or the order framework.

The topics cover:

Cloning

You can clone `AbstractOrderModel` into the target instances of any type (for example, `Order`, `Cart`, `YourCustomOrder`) from any original instance type.

The common use case is creating order instance from cart instance. The following code example shows the method signature on the `AbstractOrderService`.

```
/**
 * Creates new order from given original and change its type and type for entry. Delegates to the s
```

```
   * {@link CloneAbstractOrderStrategy} injected strategy. Resulting order remains not persisted.
   *
   * @param orderType
   *           type of result order (OrderModel will be chosen if null is passed)
   * @param entryType
   *           type of entry (if null will use service to get proper type)
   * @param original
   *           original order
   * @param code
   *           code for new order
   * @return not persisted order model instance.
   */
  clone(final ComposedTypeModel orderType, final ComposedTypeModel entryType, final AbstractOrderMode
          final String code);
```

Since this belongs to the shared functionality for any order type - you can call it from `OrderService` or `CartService` and get the result of the type that the particular service focuses on. For example, when called from `OrderService`, the clone method always returns `OrderModel` instance.

```
...
        CartModel cart = cartService.getSessionCart();

        OrderModel order = orderService.clone(null, null, cart, "123");
...
```

If you are interested in the particular composed type (Order's subtype), you can give the desired `ComposedType` of the order and entry as method's arguments. Keep in mind that as you call the method from `OrderService`, the returned class is `OrderModel` and in result you may need to cast it.

The logic of the clone method delegates to the `clone` method of the internal strategy. You can change the service behavior when `clone()` method is called by replacing the responsible strategy. Implement your own cloning strategy that realizes the `de.hybris.platform.order.strategies.ordercloning.CloneAbstractOrderStrategy` contract.

CloneAbstractOrderHook

Instead of overwriting the clone behavior with your own strategy, you can also decorate it by implementing the before or after hooks of `CloneAbstractOrderHook`. As a result, you can use multiple independent hooks without overwriting the clone behavior. For example, you can use a clone hook when given fields of an order contain references to objects in other systems or external resources. When the order gets cloned, you need to implement a hook that clones the external resource or an object in a different system.

Methods for Adding Entries to AbstractOrder

You can use any of the `addNewEntry` methods to append or insert new order entries to `AbstractOrder` on specific `entryNumber`.

Appending is possible for any case, but the insert option is implementation-dependent. For example, you are always able to shuffle entries in the cart, but not the order. The entry number in the order is an important contract information and normally should never be changed once the order has been placed.

So, while the `CartService` directly inherits the generic implementation in `DefaultAbstractOrder`, the `OrderService` performs additional check if the new entry would cause entry numbers collision.

```
 @Override
 public OrderEntryModel addNewEntry(final OrderModel order, final ProductModel product, final long q
                                    final UnitModel unit, final int number, final boolean addToPresent)
 {
```

```
        ServicesUtil.validateParameterNotNullStandardMessage("order", order);
        checkOrderEntryNumberCollisions(order, number);
        return super.addNewEntry(order, product, qty, unit, number, addToPresent);
}
```

In terms of extensibility, there are two important players in the `addNewEntry()` method logic:

- `AbstractOrderEntryTypeService`: To resolve the entry type, if none was given. The bean relies on the spring configurable order type to order entry type mapping - `orderType2orderEntryTypeMapping`. If no such mapping was given, then it checks the atomic type of the ENTRIES attribute of the given order.
- `AbstractOrderEntryService`: To create the entry instance. If for some reason you want the `addNewEntry()` method to act differently, you can try to replace the `abstractOrderEntryTypeService` or `abstractOrderEntryService` injections. You can influence the type matching by plugging in your own type-mapping (`orderType2orderEntryTypeMapping` to the `DefaulAbstractOrderEntryTypeService`).

Methods for Fetching Order Entries

See the methods that allow you to fetch order entries by product, entry number, or entry number range.

In the default implementation, the methods calls are handled by the Order DAO.

```
/**
 * Returns an order entry with the given {@link AbstractOrderEntryModel#ENTRYNUMBER}. The method de
 * access object and checks the actual persisted order in the data base, not the order state repres
 * model.
 *
 * @param order
 * @param number
 * @throws UnknownIdentifierException
 *            if no entry with the requested number exists
 * @throws IllegalArgumentException
 *            if either order is null or number is negative
 *
 * @return {@link AbstractOrderEntryModel}
 */
E getEntryForNumber(final O order, final int number);

/**
 * Returns order entries with the given {@link AbstractOrderEntryModel#ENTRYNUMBER}. The method
 * delegates to data access object and checks the actual persisted order in the data base,
 * not the order state represented by the model.
 *
 * @param order
 * @param start
 *           start of the range
 * @param end
 *           end of the range
 * @throws UnknownIdentifierException
 *            if no entry from the requested range number exists
 * @throws IllegalArgumentException
 *            if either order is null or start is negative or start is greater than end
 *
 * @return {@link AbstractOrderEntryModel}
 */
List<E> getEntriesForNumber(final O order, final int start, final int end);

/**
 * Returns order entries having the target product. In case is no entry contains the requested proc
 * empty list is returned. The method delegates to data access object and
 * checks the actual persisted order in the data base, not the order state represented by the model
 *
 * @param order
 *           - target order
 * @param product
```

```
   *           - searched product
   * @throws IllegalArgumentException
   *             if either order or product is null
   *
   * @return matching order entries
   */
  List<E> getEntriesForProduct(final O order, final ProductModel product);
```

Those methods can be grouped as repository type methods. They fetch persisted data and delegate the calls to the dedicated order data access object. In case you want to change the functionality here, you can replace or overwrite it on your own. Customized DAO must realize `de.hybris.platform.order.daos.OrderDao` contract.

Methods for Adding or Removing Taxes and Global Discounts

`AbstractOrderService` consist a set of convenience methods to manage order's taxes and global discounts.

These methods add `TaxValues` or `DiscountValues` to an order model without persisting the change. They do not have their separate components to handle the logic. They simply attach the `taxValue` or `DiscountValue` to the order or remove them. After such operation, the order (or the cart) must be recalculated.

MyOwnOrder Type

Following the pattern of `OrderService` and `CartService`, you can introduce a dedicated service for your own `AbstractOrder` type.

By extending the generic classes, you can end up having the basic functionality contract for your `AbstractOrder` subtype:

```
public interface MyOrderService extends AbstractOrderService<MyOrderModel, MyOrderEntryModel>


public class DefaultMyOrderService extends DefaultAbstractOrderService<MyOrderModel, MyOrderEntryMo
                                implements MyOrderService
```

Remember to register the service with proper dependency on `abstractOrderService`.

```
<alias alias="myOrderService" name="defaultMyOrderService"/>
<bean id="defaultMyOrderService" class="de.hybris.platform.order.impl.DefaultMyOrderService"
                                parent="abstractOrderService" >
<!-- you can use dedicated strategies beans, or using the default ones by leaving the spot empty-->
</bean>
```

Order-Related Functionality

You can create an order for a given cart or specify what you want order numbers to consist of.

Create Order from Cart

By calling this method, you create an `OrderModel` instance out of the source `CartModel`. Then, after persisting the model you can have an actual order in the system, which represents a real selling contract.

```
/**
 * Create the order for the given <code>cart</code>. This method focuses on creating an {@link Orde
 * instance from the given {@link CartModel} instance. The order instance remains unsaved
```

```
   * and not calculated.</br> This method does nothing with the cart member attributes (addresses, pa
   * It also leaves the target cart untouched.
   *
   * If you want to calculate cart or order, use {@link CalculationService}.
   *
   *
   * @param cart
   *           the target {@link CartModel}
   * @return a non persisted {@link OrderModel}
   * @throws InvalidCartException
   *            if the cart is invalid according to the used {@link CartValidator}.
   */
  OrderModel createOrderFromCart(CartModel cart) throws InvalidCartException;
```

The default implementation delegates to the dedicated strategy. The spring bean defines the process of the order placement. Use the default one, or define your own order placement process by exchanging the current strategy with your own. To do so, implement the `de.hybris.platform.order.strategies.CreateOrderFromCartStrategy`.

Customizing Order Number Generation

Use the new `KeyGenerator` interface and its `PersistentKeyGenerator` implementation instead of `NumberSeriesManager`.

This allows users to customize the order generation process using values specified in the `local.properties` file. There is a number of available customization options. To use them, define the following set of properties in `local.properties`.

You can specify:

`local.properties`

- How many digits are in the order code
- Start order code
- Whether the code is numeric only
- Template to be used in order code generation

```
keygen.order.code.digits=8
keygen.order.code.start=00000000
keygen.order.code.type=numeric
keygen.order.code.template=$
```

The bean with properties is now in `core-spring.xml`.

```
<bean id="orderCodeGenerator" class="de.hybris.platform.servicelayer.keygenerator.impl.PersistentKe
<property name="key" value="${keygen.order.code.name}"/>
<property name="digits" value="${keygen.order.code.digits}"/>
<property name="start" value="${keygen.order.code.start}"/>
<property name="numeric" value="${keygen.order.code.numeric}"/>
<property name="template" value="${keygen.order.code.template}"/>
```

You can also use the symbol @, which stands for the system PK:

`local.properties`

```
keygen.order.code.template=ACC-@-$-DE
```

This generates order codes that contain a unique Platform number (return value of `MasterTenant.getClusterIslandPK()`), for example `123456`. The whole order code in this case looks like this:

```
ACC-123456-00000001-DE
```

Note

When you customize the order number using the method above while you already have orders in the system, you must restart Platform to apply your changes. To customize order generation without restarting Platform, use Administration Console groovy web console:

```
import de.hybris.platform.jalo.numberseries.*
NumberSeriesManager nm = NumberSeriesManager.getInstance()
def s = nm.getNumberSeries("order_code")
println "Before: current: ${s.currentNumber} type: ${s.type} template: ${s.template}"
nm.resetNumberSeries(s.key, "777", s.type, s.template )
def s2 = nm.getNumberSeries("order_code")
println "After: current: ${s2.currentNumber} type: ${s2.type} template: ${s2.template}"
```

Framework Components

The `AbstractOrderService` relies on several lower-level components that together support the order framework.

The following sections provide an overview of these components:

Services

See the services that AbstractOrderService relies on.

AbstractOrderEntryTypeService

The key functionality allows you to determine suitable order entry type, that means `ComposedType` for the target order.

```
public interface AbstractOrderEntryTypeService
{

/**
 * Returns {@link ComposedTypeModel} of order entry that best match the target order.
 *
 * @param order
 *           target {@link AbstractOrderModel}
 * @return {@link ComposedTypeModel} of given order's entries.
 */
public ComposedTypeModel getAbstractOrderEntryType(final AbstractOrderModel order);

/**
 * Convenience method for resolving {@link Class} of the given order entry {@link ComposedTypeModel
 *
 * @param entryType
 *           - {@link ComposedTypeModel} you want to check class for.
 * @return {@link Class} that corresponds to the type.
 */
public Class getAbstractOrderEntryClassForType(ComposedTypeModel entryType);
}
```

If you want to influence the `automatic type matching` of the `AbstractOrderEntryTypeService`, extend it and configure `orderType2orderEntryTypeMapping` mapping property in the extension's spring xml configuration. This is shown on the following example, where orders are expected to get some custom order entries and custom cart are based on standard cart entries.

```
...
<alias alias="abstractOrderEntryTypeService" name="customAbstractOrderEntryTypeService"/>
<bean id="customAbstractOrderEntryTypeService" parent="defaultAbstractOrderEntryTypeService" >
<!-- You can configure order to order entry type mapping of your choice. If this map is not configu
the service will resolve the type from atomic type of the order's 'entries' collection. -->
<property name="orderType2orderEntryTypeMapping">
        <map>
                <entry key="Order" value="CustomOrderEntry"/>
                <entry key="CustomCart" value="CartEntry"/>
        </map>
</property>
</bean>
```

AbstractOrderEntryService

Similarly to `AbstractOrderService`, this service is also template service, only defining a set of common functionality on the `AbstractOrderEntryModel` type level.

```
public interface AbstractOrderEntryService<E extends AbstractOrderEntryModel>
```

And similarly to `AbstractOrderService`, it is also extended by typed interfaces focused on the concrete order entry types. Example for the cart entries is shown on the code sample below.

```
public interface CartEntryService extends AbstractOrderEntryService<CartEntryModel>
```

Sample for order entries-related service.

```
public interface OrderEntryService extends AbstractOrderEntryService<OrderEntryModel>
```

At this point, the logic is implemented only on the generic `AbstractOrderEntry` level. The `OrderEntryService` and `CartEntryService` are only marker interfaces to provide consistency.

The `AbstractOrderEntryService` service is a framework's helper service oriented on order entries. It enables you to create such order entry for a given order using the following:

- **Automatic type matching**: Using `AbstractOrderEntryTypeService`.
- **Custom entry type on demand**

```
/**
 * Creates a new instance of order entry for a given abstract order instance. The entry type is cho
 * to best suit the order instance. New entry remains not persisted.
 *
 * @see AbstractOrderEntryTypeService#getAbstractOrderEntryType(AbstractOrderModel)
 *
 * @param abstractOrder
 *           target order
 * @return new order entry of the suitable runtime type.
 */
public E createEntry(final AbstractOrderModel abstractOrder);

/**
 * Creates a new instance of abstract order entry of the specific composed type for a given order i
 * New entry remains not persisted.
 *
 * @param abstractOrder
 *           target order
 * @param entryType
 *           create entry of this specific type
 * @return new abstract order entry of the specific runtime type.
```

```
   */
  public AbstractOrderEntryModel createEntry(final ComposedTypeModel entryType,final AbstractOrderMo
```

As it relies on the `AbstractOrderEntryTypeService`, you could inject different implementation of it in your extension's spring configuration.

If you want to provide only the order entry-related logic for a particular entry type (for example, `MyOrderEntry`), you can provide a new implementation of `AbstractOrderEntryService` for that type (`AbstractOrderEntryService <MyOrderEntry>`) and register it in the spring configuration. You can further use it in your corresponding `AbstractOrderService<MyOrder, MyOrderEntry>` service.

If `MyOrderEntry` extends `AbstractOrderEntry` and `MyOrder` extends `AbstractOrder`, then you can configure new services as shown in examples below.

- For `MyOrder`:

```
public interface MyOrderService extends AbstractOrderService<MyOrderModel>


public class DefaultMyOrderService extends DefaultAbstractOrderService<MyOrderModel> implemen
```

- For `MyOrderEntry`:

```
public interface MyOrderEntryService extends AbstractOrderEntryService<MyOrderEntryModel>


public class DefaultMyOrderEntryService extends DefaultAbstractOrderEntryService<MyOrderEntry
                                implements MyOrderEntryService


...
<bean id="myOrderEntryService"
        class="de.hybris.platform.order.impl.DefaultMyOrderEntryService"
                        parent="defaultAbstractOrderEntryService">
                <!-- inject whatever you need to use in your implementation -->
</bean>

<bean id="myOrderService" class="de.hybris.platform.order.impl.DefaultMyOrderService"
                        parent="defaultAbstractOrderService">
                <property name="abstractOrderEntryService" ref="myOrderEntryService"/>
                        <!-- inject whatever you need to use in your implementation -->
</bean>
...
```

Strategies

The `DefaultAbstractOrderService` has dedicated and exchangeable strategies for the following functionality provided by the order framework: cloning and saving.

SubmitOrderStrategy

This strategy submits the order after it is created.

Default implementation `EventPublishingSubmitOrderStrategy` sends `SubmitOrderEvent`. If you wish to do something else, then you can implement the interface and inject it into the Order Service that provides the list of strategies fired one after another.

```
public interface CreateOrderFromCartStrategy
{
 /**
 * Submits the order. One of strategies that is invoked by {@link OrderService#submitOrder(OrderMod
 * your own implementation(s) there.
 *
 * @param order
 *           order to submit.
 */
 void submitOrder(OrderModel order);
}
```

CloneAbstractOrderStrategy

Cloning is responsible for creating one `AbstractOrderModel` instance (clone) from any other `AbstractOrderModel` instance. For example, you can clone a real future order out of the customer cart after he or she has finally agreed to finalize the order.

```
public interface CloneAbstractOrderStrategy
{

/**
 * Make a clone of an abstract order (eventually change the type).
 *
 * @param orderType
 *           type of newly created order
 * @param entryType
 *           type of order entry of newly created order
 * @param original
 *           original order
 * @param code
 *           code of created order
 */
<T extends AbstractOrderModel> T clone(ComposedTypeModel orderType, ComposedTypeModel entryType,
                        AbstractOrderModel original, String code, final Class abstractOrderClassRes
                        final Class abstractOrderEntryClassResult);

/**
 * Make a clone of entries (eventually change their type). If entriesType is null, the type will be
 * according to {@link AbstractOrderEntryTypeService#getAbstractOrderEntryType(AbstractOrderModel)}
 *
 * @param entriesType
 *           type of cloned entries
 * @param original
 *           original abstractOrder
 * @return cloned order entries
 * @throws IllegalArgumentException
 *            if original is null
 */
<T extends AbstractOrderEntryModel> Collection<T> cloneEntries(ComposedTypeModel entriesType,
                        AbstractOrderModel original);
}
```

If you want to replace the cloning strategy in the framework with your own implementation (or the default extension of `DefaultCloneAbstractOrderStrategy`), you can provide an implementation of the `de.hybris.platform.order.strategies.ordercloning.CloneAbstractOrderStrategy` and configure it in your extension's spring configuration.

```
public class MyCloneOrderStrategy extends DefaultCloneAbstractOrderStrategy implements CloneAbstrac
{
   ...
}
```

```
<alias alias="cloneAbstractOrderStrategy" name="myCloneAbstractOrderStrategy"/>
<bean id="myCloneAbstractOrderStrategy"
                        class="de.hybris.platform.order.strategies.ordercloning.impl.MyCloneAbstrac
                          parent="defaultCloneAbstractOrderStrategy">
                <!-- inject whatever you need to use in your implementation -->
</bean>
```

CreateOrderFromCartStrategy

This strategy is responsible for taking a cart and making a real order from it.

```
public interface CreateOrderFromCartStrategy
{
/**
 * Validates the cart using {@link CartValidator} and performs cart to order cloning.
 *
 * @param cart
 *           - the target {@link CartModel}
 * @throws InvalidCartException
 *            according to {@link CartValidator} implementation.
 * @return an unsaved and not calculated {@link OrderModel} instance.
 */
OrderModel createOrderFromCart(CartModel cart) throws InvalidCartException;
}
```

This process usually differs in every project. It means that you can define your own logic. The default implementation additionally validates the cart by using injected `CartValidator`. One can define his or her own validator in order to prevent from creating orders from carts not fulfilling your business requirements.

Here is the spring configuration of the default strategy:

```
<alias alias="createOrderFromCartStrategy" name="defaultCreateOrderFromCartStrategy"/>
<bean id="defaultCreateOrderFromCartStrategy"
                        class="de.hybris.platform.order.strategies.impl.DefaultCreateOrderFromCartS
                        >
        <property name="cartValidator" ref="cartValidator"/>
        <property name="cloneAbstractOrderStrategy" ref ="cloneAbstractOrderStrategy"/>
</bean>
```

As you can see, it uses the validator mentioned earlier and the cloning strategy to clone the cart into a new order.

If you wish to exchange the order placement strategy in the framework with your own implementation (or extension of `DefaultCreateOrderFromCartStrategy`), you can provide an implementation of the `de.hybris.platform.order.strategies.CreateOrderFromCartStrategy` and configure it in your spring configuration. The following example presents the case where a customized strategy extends the default one.

```
public class MyPlaceOrderStrategy extends DefaultCreateOrderFromCartStrategy implements CreateOrder
{
   ...
}
```

```
<alias alias="createOrderFromCartStrategy" name="myCreateOrderFromCartStrategy"/>
<bean id="myCreateOrderFromCartStrategy" class="de.hybris.platform.order.strategies.impl.MyPlaceOrd
                    parent="defaultCreateOrderFromCartStrategy">
          <!-- inject whatever you need to use in your implementation -->
</bean>
```

SaveAbstractOrderStrategy

`SaveAbstractOrderStrategy` is responsible for persisting the given `AbstractOrderModel` instances.

```
public interface SaveAbstractOrderStrategy<O>
{
/**
 * Saves the given order model and the order entries.
 *
 * @param order
 *           order model
 * @return saved and refreshed order model
 */
O saveOrder(final O order);
}
```

Usually, you could persist the order models using standard method like calling `save()` method from the `ModelService`. But for such important data like orders it might happen that there is a need of some special saving logic. The default implementation tries to save the order and the entries within one transaction and refresh the models.

If you want to replace the saving strategy in the framework with your own implementation (or the default extension of `DefaultSaveAbstractOrderStrategy`), you can provide an implementation of the `de.hybris.platform.order.strategies.saving.SaveAbstractOrderStrategy` and configure it in the spring configuration of your extension.

```
public class MySaveAbstractOrderStrategy extends DefaultSaveAbstractOrderStrategy
                        implements SaveAbstractOrderStrategy
{
   ...
}
```

```
<alias alias="saveAbstractOrderStrategy" name="mySaveAbstractOrderStrategy"/>
<bean id="mySaveAbstractOrderStrategy"
                        class="de.hybris.platform.order.strategies.saving.impl.MySaveAbstractOrderS
                          parent="defaultSaveAbstractOrderStrategy">
                <!-- inject whatever you need to use in your implementation -->
</bean>
```

DataAccessObjects

In addition to the services and strategies determining the basic flow in the framework, there is also a component used to fetch the order-related data from the persistence layer.

OrderDao - Order Data Access Object

```
public interface OrderDao
{
/**
 * Returns order entries with the matching order entry number
 *
 * @param entryTypeCode
 *           - entries of this specific type will be searched. I.e 'OrderEntry', 'CartEntry'
 * @param number
 *           - requested entry number
 * @param order
 *           - target order
 *
 * @return List of matching order entries, or {@link Collections#EMPTY_LIST} in case if no entries
 */
List<AbstractOrderEntryModel> findEntriesByNumber(String entryTypeCode, AbstractOrderModel order, i

/**
 * Returns order entries with the order entry number from the requested range
```

```
   *
   * @param entryTypeCode
   *           - entries of this specific type will be searched. I.e 'OrderEntry', 'CartEntry'
   *
   * @param order
   *           - target order
   * @param start
   *           lower range limit
   * @param end
   *           upper range limit
   *
   * @return List of matching order entries, or {@link Collections#EMPTY_LIST} in case if no entries
   */
  List<AbstractOrderEntryModel> findEntriesByNumber(String entryTypeCode, AbstractOrderModel order, i

  /**
   * Returns order entries with the matching product
   *
   * @param entryTypeCode
   *           - entries of this specific type will be searched. I.e 'OrderEntry', 'CartEntry'
   * @param product
   *           - requested {@link ProductModel}
   * @param order
   *           - target order
   *
   * @return List of matching order entries, or {@link Collections#EMPTY_LIST} in case if no entries
   */
  List<AbstractOrderEntryModel> findEntriesByProduct(String entryTypeCode, final AbstractOrderModel o
                  final ProductModel product);

  /**
   * Returns orders of the type specified with the given currency.
   *
   * @param currency
   *           the target currency
   *
   * @return {@link List} of {@link AbstractOrderModel} - matched orders
   * @throws IllegalArgumentException
   *            if currency is null
   * @throws IllegalStateException
   *            if currency is not persisted.
   */
  List<AbstractOrderModel> findOrdersByCurrency(CurrencyModel currency);

  /**
   * Returns orders with the given delivery mode.
   *
   * @param deliveryMode
   *           target {@link DeliveryModeModel}
   *
   * @return {@link List} of {@link AbstractOrderModel} - matched orders
   * @throws IllegalArgumentException
   *            if deliveryMode is null
   * @throws IllegalStateException
   *            if deliveryMode is not persisted.
   */
  List<AbstractOrderModel> findOrdersByDelivereMode(DeliveryModeModel deliveryMode);

  /**
   * Returns orders with the given payment mode.
   *
   * @param paymentMode
   *           target {@link PaymentModeModel}
   *
   * @return {@link List} of {@link AbstractOrderModel} - matched orders
   * @throws IllegalArgumentException
   *            if paymentMode is null
   * @throws IllegalStateException
   *            if paymentMode is not persisted.
   */
  List<AbstractOrderModel> findOrdersByPaymentMode(PaymentModeModel paymentMode);
  }
```

If you wish to change something in a way that these order and order entry models are fetched from the database, then you can extend the default implementation. Implement the `OrderDao` interface and register it in your extension's spring configuration. The example presents the case where customized DAO extends the default one.

```
public class MyOrderDao extends DefaultOrderDao implements OrderDao
{
   ...
}
```

```
<alias alias="orderDao" name="myOrderDao"/>
<bean id="myOrderDao" class="de.hybris.platform.order.daos.impl.MyOrderDao"
                      parent="defaultOrderDao">
        <!-- inject whatever you need to use in your implementation -->
</bean>
```

Order-Related Interceptors

Modifying services and strategies is not the only way you can influence how the order framework behaves. Have a look at the interceptors defined for orders and order entries.

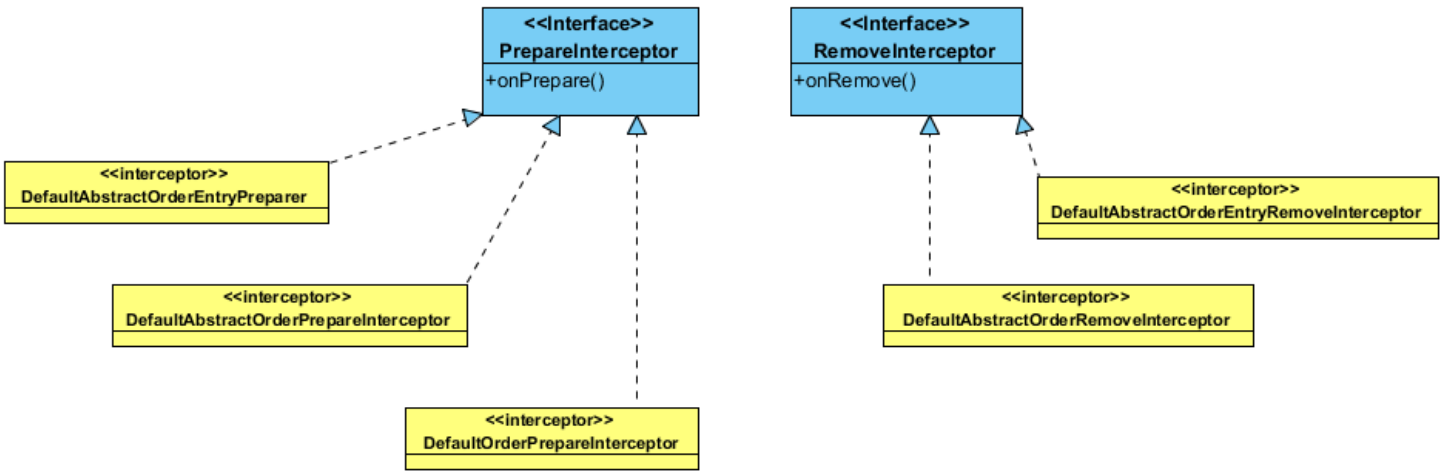


Fig. Interfaces for Order-Related Interceptors

By adding new interceptors or modifying the existing ones, you can hook up custom logic whenever orders, in other words order entries, are prepared, validated, loaded, or removed. For more details on interceptors, see Interceptors documentation.

DefaultAbstractOrderEntryPreparer

This interceptor resets the calculated flag on the entry and on the owning order, in case one of the predefined attributes has been modified. This provides some automation in determining if the order and the order entry are to be calculated.

This interceptor is activated by default. It prepares:

- **order entry info field**: According to the contained product.
- **orderentry.infofield.orderEntry**: Property that defines the syntax of the information.

```
<alias name="defaultAbstractOrderEntryPreparer" alias="abstractOrderEntryPreparer"/>
<bean id="defaultAbstractOrderEntryPreparer"
                class="de.hybris.platform.order.interceptors.DefaultAbstractOrderEntryPreparer" >
        <property name="typeService" ref="typeService"/>
        <property name="modelService" ref="modelService"/>
```

```
        <property name="configurationService" ref="configurationService"/>
        <!-- optionally, define attributes that, when modified, will reset CALCULATED flag
        <property name="attributesForOrderRecalculation">
                <list>
                        <value>product</value>
                        <value>quantity</value>
                        <value>basePrice</value>
                        ...
                        </list>
                </property>
                -->
</bean>
```

In terms of extensibility you can:

1. Deactivate the interceptor.
2. Replace it with your own implementation of `PrepareInterceptor`.
3. Change the syntax of the order entry `info` field and modify `orderentry.infofield.orderEntry` property.
4. Decide which attributes, when modified, should reset CALCULATED flags. By default the following set is used: PRODUCT, QUANTITY, UNIT, BASEPRICE, TAXVALUES, DISCOUNTVALUES, GIVEAWAY, REJECTED.

DefaultAbstractOrderEntryRemoveInterceptor

Before an order entry is removed, the interceptor marks the owning order as not calculated.

```
<alias name="defaultAbstractOrderEntryRemoveInterceptor" alias="abstractOrderEntryRemoveInterceptor
<bean id="defaultAbstractOrderEntryRemoveInterceptor"
                class="de.hybris.platform.order.interceptors.DefaultAbstractOrderEntryRemoveInterce
                >
        <property name="modelService" ref="modelService"/>
</bean>
```

In terms of extensibility, you can:

1. Deactivate the interceptor.
2. Replace it with your own implementation of `RemoveInterceptor`.

DefaultAbstractOrderPrepareInterceptor - AbstractOrderModel Type

The interceptor provides order code value in case it has not been already set in the model directly, and controls the **CALCULATED** flag on the order and its entries.

```
<alias name="defaultAbstractOrderPrepareInterceptor" alias="abstractOrderPrepareInterceptor"/>
<bean id="defaultAbstractOrderPrepareInterceptor"
                        class="de.hybris.platform.order.interceptors.DefaultAbstractOrderPrepareInt
                        >
        <property name="keyGenerator" ref="orderCodeGenerator"/>
        <!-- define qualifiers of order attributes, which, when modified, the order and orderEntrie
        need to be recalculated.
        <property name="attributesForOrderEntriesRecalculation">
                <list>
                        <value>date</value>
                        <value>user</value>
                        <value>currency</value>
                        <value>net</value>
                </list>
        </property>
```

```
        -->
        <!-- define qualifiers of order attributes, which, when modified, the order needs to be rec
        <property name="attributesForOrderRecalculation">
                <list>
                        <value>deliveryModel</value>
                        <value>deliveryCost</value>
                        <value>paymentModel</value>
                        <value>paymentCost</value>
                                ...
                </list>
        </property>
        -->
</bean>
```

In terms of extensibility you can:

1. Deactivate the interceptor.
2. Replace it with your own implementation of `PrepareInterceptor`.
3. Provide your own order key generator: `keyGenerator`.
4. Decide which attributes, when modified, should reset CALCULATED flag on order. By default the following set is used: DELIVERYMODE, DELIVERYCOST, PAYMENTMODE, PAYMENTCOST, TOTALTAXVALUES, DISCOUNTS, DISCOUNTSINCLUDEDELIVERYCOST, DISCOUNTSINCLUDEPAYMENTCOST, DELIVERYADDRESS, PAYMENTADDRESS.
5. Decide which attributes, when modified, should reset CALCULATED flag of an order and order entries. By default, the following set is used: NET, USER, CURRENCY, DATE.

DefaultAbstractOrderRemoveInterceptor - AbstractOrderModel Type

This type removes the payment and delivery address and payment info, which have been cloned for the order.

```
<bean id="defaultAbstractOrderRemoveInterceptor"
                class="de.hybris.platform.order.interceptors.DefaultAbstractOrderRemoveInterceptor"
        <property name="modelService" ref="modelService"/>
</bean>
```

It removes the following data that have been cloned for the order:

1. PAYMENTADDRESS
2. DELIVERYADDRESS
3. PAYMENTINFO

In terms of extensibility you can:

1. Deactivate the interceptor.
2. Replace it with your own implementation of `RemoveInterceptor`.

DefaultOrderPrepareInterceptor - Particular Type of AbstractOrderModel: OrderModel Type

This type clones the order contract data such as: `addresses` and `payment info`.

```
<alias name="defaultOrderPrepareInterceptor" alias="orderPrepareInterceptor"/>
        <bean id="defaultOrderPrepareInterceptor"
                class="de.hybris.platform.order.interceptors.DefaultOrderPrepareInterceptor" >
```

```
        <property name="orderPartOfMembersCloningStrategy" ref="orderPartOfMembersCloningStrategy",
</bean>
```

The cloning bases on a strategy that can be replaced with a customized implementation. In terms of extensibility you can:

1. Deactivate the interceptor.
2. Replace it with your own implementation of `PrepareInterceptor`.
3. Replace the strategy, which decides when cloning is necessary and how the cloning operation proceeds.

OrderPartOfMembersCloningStrategy

```
public interface OrderPartOfMembersCloningStrategy
{

/**
 * Clones an address and sets the order as the clone's owner.
 *
 * @param address
 *           address to clone
 * @param order
 *           owning order
 * @return cloned but <b>not persisted</b> {@link AddressModel} instance.
 */
AddressModel cloneAddressForOrder(AddressModel address, OrderModel order);

/**
 * Checks according to business strategies whether given address needs to be cloned
 * as a part of order's contract.
 *
 * @param address
 *
 * @param order
 *
 * @return true if this address needs to be cloned.
 * @throws IllegalArgumentException
 *            if order is null
 */
boolean addressNeedsCloning(AddressModel address, OrderModel order);

/**
 * Clones a payment info and sets the order as the clone's owner.
 *
 * @param paymentInfo
 *           payment info to clone
 * @param order
 *           owning order
 * @return cloned but <b>not persisted</b> {@link PaymentInfoModel} instance.
 */
PaymentInfoModel clonePaymentInfoForOrder(PaymentInfoModel paymentInfo, OrderModel order);

/**
 * Checks according to business strategies whether given payment info needs to be cloned as a part
 * order's contract.
 *
 * @param paymentInfo
 *
 * @param order
 *
 * @return true if this address needs to be cloned.
 * @throws IllegalArgumentException
 *            if order is null
 */
boolean paymentInfoNeedsCloning(PaymentInfoModel paymentInfo, OrderModel order);
}
```

You can extend default implementation of the `DefaultOrderPartOfMembersCloningStrategy` interface if you wish to influence how and when the `address` and `payment info` are cloned for the purpose of the order placement.

```
public class MyOrderPartOfMembersCloningStrategy extends DefaultOrderPartOfMembersCloningStrategy :
        OrderPartOfMembersCloningStrategy
{
   ...
}
```

Configure the strategy in the spring configuration file. Be advised not to use the `order-spring.xml` file for that purpose, but to use your own file, as any update of SAP software could overwrite your changes.

```
<alias name="myOrderPartOfMembersCloningStrategy" alias="orderPartOfMembersCloningStrategy"/>
<bean id="myOrderPartOfMembersCloningStrategy"
        class="de.hybris.platform.order.strategies.ordercloning.impl.MyOrderPartOfMembersCloningStr
        parent="defaultOrderPartOfMembersCloningStrategy"/>
```

Price, Tax, and Discount Calculation

The `europe1` extension handles all price, tax, and discount calculations in SAP Commerce.

Price Calculation

SAP Commerce stores all possible prices for products and customers as price row objects. Every price row represents a certain price constellation, which is optionally related to a certain date range, customer, product, group of customers, or group of products.

Tax Calculation

SAP Commerce stores all possible taxes for products and customers as TaxRow objects. Every TaxRow object represents a certain tax constellation, which is optionally related to a certain date range, customer, product, group of customers, or a group of products.

Discount Calculation

SAP Commerce stores all possible discounts for products and customers as DiscountRow objects. EveryDiscountRow represents a certain discount constellation, which is optionally related to a certain date range, customer, product, group of customers, or group of products.

Time-Dependent Prices and Discounts

The types that define prices, taxes, and discounts have a common supertype, the abstract PriceDiscountTaxRow, or `PDTRow` type.

Overriding and Testing Settings

You can override customer-related and product-related group settings.

Decoupling PDTRows from Product

Find out how decoupling `PDTRows` from the `Product` affects synchronization.

Editing Tax and Discount Values of an Order in Backoffice

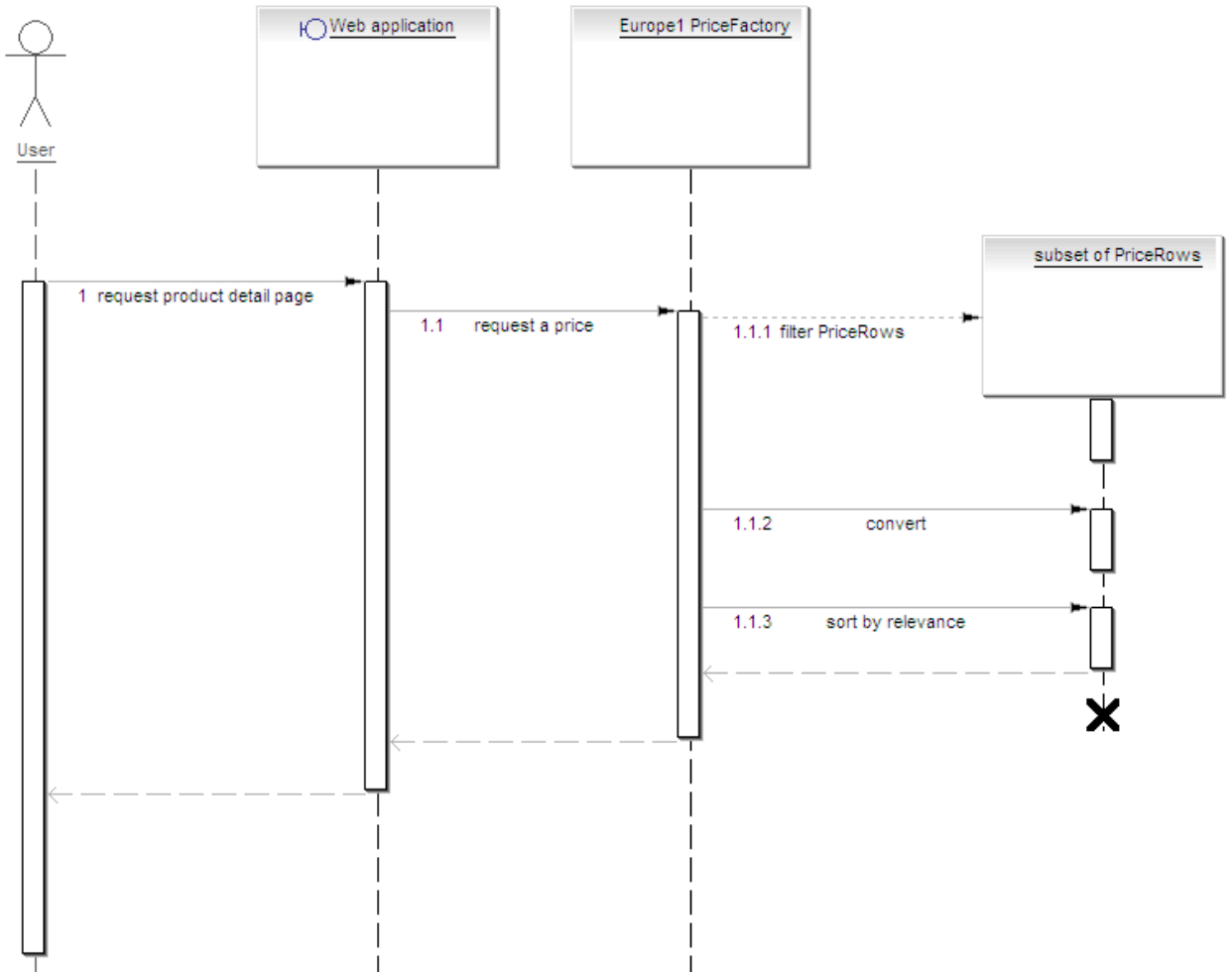
You can edit tax and discount values of an order in Backoffice. We assume that you apply taxes per line item - at order entry level. It means a tax is calculated on the total selling price of an individual item. Our examples show discounts applied both at order entry, and order levels.

Price Calculation

SAP Commerce stores all possible prices for products and customers as price row objects. Every price row represents a certain price constellation, which is optionally related to a certain date range, customer, product, group of customers, or group of products.

Every PriceRow object is a potential price. When a price is required for display, the Europe1 PriceFactory retrieves all PriceRow objects related to the combination of customer and products. This is because unless an order is placed, the number of products is not determined and therefore scale prices might apply. When a price is required for an order, the Europe1 PriceFactory retrieves all PriceRow objects related to the combination of customer and products and then chooses the best matching PriceRow.

The following diagram shows the basic course of action for the ordering case.



To handle a request for a price for a product, the Europe1 PriceFactory retrieves all PriceRow objects that match the combination of user and product. The remaining subset of PriceRows is converted so that the PriceRows are comparable. The converted PriceRows are sorted by the relevance to the combination of customer and product. Finally, the best matching combination is returned as the final price.

Attributes of a PriceRow

The following table provides an overview of the attributes of a PriceRow object.

Attribute Name	Attribute Type	Mandatory	Description
channel	PriceRowChannel	no	Channel for which the price is effective (for example, mobile or desktop).
currency	Currency	yes	Currency of the price. No default value.
dateRange	StandardDateRange	no	Date range for the PriceRow object's validity.
giveAwayPrice	java.lang.Boolean	yes	Specifies whether a price row has a give-away price. The default value is `false`.

Attribute Name	Attribute Type	Mandatory	Description
endTime	java.util.Date	no	Specifies the end date of a price's validity.
matchValue	java.lang.Integer	no	Internally calculated value that is based on user and product information. The value is used for sorting PriceRow records by their relevance.
minqtd	java.lang.Long	yes	Scale price definition. A customer needs to order at least this number of units for this price to be effective. Default value is 1.
net	java.lang.Boolean	no	Specifies whether the price specified is net or gross. Default value is gross.
pg	ProductPriceGroup	no	Product Pricelist for which the price is effective. The price is applied to all members of this Product Pricelist.
price	java.lang.Double	yes	Price to be set.
product	Product	no	Product for which the price is set.
productMatchQualifier	java.lang.Long	no	Internally calculated value based on product information. The value is used to filter out the PriceRow records that are considered when a price is being retrieved.
startTime	java.util.Date	no	Specifies the start date of a price's validity.
ug	UserPriceGroup	no	Customer Pricelist for which the price is effective. The price is applied to all members of this Customer Pricelist.
unit	Unit	yes	The unit this price is measured in. No default value.
unitFactor	java.lang.Integer	yes	The number of individual units the price refers to. If you set the price to 50 EUR and the unitFactor to 5, the individual unit's price is 10 EUR (50 EUR / 5). Default value is 1.
user	User	no	Customer for which the price is set.
userMatchQualifier	java.lang.Long	no	Internally calculated value that is based on user information.

Attribute Name	Attribute Type	Mandatory	Description
			The value is used to filter out the PriceRow records that are considered when a price is being retrieved.

If neither the customer nor a UserPriceGroup is given for a certain PriceRow, the specified price applies to all customers in SAP Commerce. If neither the product nor ProductPriceGroup is given for a certain PriceRow, the specified price applies to all products in SAP Commerce.

Filtering PriceRows

During the filtering phase, the Europe1 PriceFactory selects all PriceRows matching the combination of customer and product. A PriceRow must match all of the following criteria to apply:

- customer or customer group or no customer-related value (blank field)
- product or product group or no product-related value (blank field)
- date range
- scale price

If a PriceRow fails to match all the criteria, it is not applied and therefore not considered by the Europe1 PriceFactory.

Customer Matching

To match a customer, a PriceRow must be one of the following:

- empty
- set to match the customer, or the customer's user group.

To match the customer, the PriceRow's **user** attribute must be set to contain the customer.

To match the customer's user group, the PriceRow's **ug** (user price group) must be set to contain a user group (an instance of **UserPriceGroup**) which contains the customer.

If there is a PriceRow that has no Customer or UserPriceGroup assigned to it, that PriceRow is applied to all customers and therefore be effective by default.

The following table lists some possible combinations and whether the respective PriceRow is applied to the customer.

PriceRow Assigned To	Current Customer	PriceRow Applies to the Current Customer?
kotal	kotal	✅
kotal	abel	⛔
customergroup	kotal (member of customergroup)	✅
customergroup	abel (not a member of customergroup)	⛔
(no assignment)	kotal	✅

PriceRow Assigned To	Current Customer	PriceRow Applies to the Current Customer?
(no assignment)	abel	✅
(no assignment)	kotal (member of customergroup)	✅
(no assignment)	abel (not a member of customergroup)	✅

In other words:

- If the current Customer is different from the Customer the PriceRow is assigned to, the PriceRow is not applied.
- If the PriceRow is assigned to a UserPriceGroup and the current Customer is not a member of this UserPriceGroup, the PriceRow is not applied either.

Product Matching

To match a product, a PriceRow must be one of the following:

- empty
- set to match the product or the product's price group.

To match the product, the PriceRow's **product** attribute must be set to contain the product.

To match the product's price group, the PriceRow's **pg** (product price group) must be set to contain a price group (an instance of **ProductPriceGroup**) which contains the product.

If there is a PriceRow that has no Product or ProductPriceGroup assigned to it, that PriceRow is applied to all products and therefore be effective by default.

The following table shows you some possible combinations and whether the respective PriceRow is applied to the product.

PriceRow Assigned To	Current Product	PriceRow Applies to the Current Product?
B00005LJ7N-1	B00005LJ7N-1	✅
B00005LJ7N-1	C232134_0	➖
cameragroup	B00005LJ7N-1 (member of cameragroup)	✅
cameragroup	C232134_0 (not a member of cameragroup)	➖
(no assignment)	B00005LJ7N-1	✅
(no assignment)	C232134_0	✅
(no assignment)	B00005LJ7N-1 (member of cameragroup)	✅
(no assignment)	C232134_0	✅

PriceRow Assigned To	Current Product	PriceRow Applies to the Current Product?
	(not a member of cameragroup)	

In other words:

- If the current Product is different from the Product the PriceRow is assigned to, the PriceRow is not applied.
- If the PriceRow is assigned to a ProductPriceGroup and the current Product is not a member of this ProductPriceGroup, the PriceRow is not applied either.

Scale Prices

A PriceRow object can define a scale price via its **minqtd** attribute. If the value is greater than 1 (one), the PriceRow defines a scale Price. Usually, higher order volumes result in a lower price per individual unit.

A PriceRow with a scale price is only applied if the customer buys the minimum units required for the scale price to be triggered. When you define scale prices, you need to pay attention to the effects of multiple PriceRow objects or you may run into a situation where it benefits the customer to order smaller amounts of the product (e.g., 1 for 7 EUR instead of 4 for 8 EUR each).

Below is an example of prices set for numerous scale price quantities. The right table displays the price per unit that is applied based on the number of units the customer purchases.

Scale Price Quantity	Price per Unit
1	100 €
5	95 €
20	90 €
50	75 €
100	50 €
1000	30 €

Ordered Quantity	Effective Price per Unit
5	95 €
7	95 €
20	90 €
49	90 €
50	75 €
99	75 €
100	50 €
999	50 €
1000	30 €

Converting PriceRows

Net/Gross Prices

For each PriceRow, you can define whether it should be net or gross. The net price is the "pure" price of the product and does not include taxes, whereas the gross price is the effective cost for the customer and includes all relevant taxes. PriceRows in SAP Commerce have an attribute called **net** that defines whether its value is given in net or in gross format. Use the **isNet()** and **setNet(Boolean net**) methods to retrieve and specify a price's net or gross status (**true** = net, **false** = gross).

Currency Conversion

A PriceRow is always specified in a certain currency. If a PriceRow is specified in a currency different from the current customer's session, the PriceRow's value is converted automatically based on the conversion rates in SAP Commerce. The following table provides an example of how this works.

Price Value in PriceRow	Session Currency	Conversion Rate	Effective Price
10 USD	USD	1:1	10 USD
10 USD	EUR	1:1	10 EUR
10 USD	GBP	1.3:1	7.70 GBP

Using unitFactor

The **unitFactor** attribute specified with a PriceRow object allows you to scale prices with units. Effectively, when calculating a price on a PriceRow, the Europe1 PriceFactory divides the value of the price attribute by the value of the **unitFactor**. In other words, if you define a PriceRow object with a price of 1000 EUR and a **unitFactor** of 500, each single unit costs 2 EUR. Two possible applications for this:

- You can scale small prices up by a manageable factor
- You can base prices on the lot size you receive the goods in

The following are a couple of examples:

- In your cheese shop 100 g of cheddar has a price of 2.50 EUR. A single gram of cheddar would then cost 0.025 EUR. In your PriceRow object, you could specify the price for cheddar in EUR per grams (0.025 EUR). Alternatively, you could specify the price in EUR per kilogram and add a unitFactor of 1000, resulting in (25,00 EUR / kg) / 1000 effectively.
- Your shop sells pre-assembled computers, 16 of which arrive on an europalette each. Every computer has a price of 250 EUR. Instead of setting up a price of 250 EUR per computer in your PriceRow object, you could alternatively set up the price for an entire palette and specify a **unitFactor** of 16.

In order to configure the number of digits at the base price, use the following property:

advanced.properties

```
#scale of the BigDecimal value to be returned. Used during a conversion of Number value to the SQL
jdbcmappings.big_decimal_scale=5
```

Sorting PriceRows and Selecting a Final Price

After the Europe1 PriceFactory has gathered all PriceRows that might apply, those PriceRows are sorted by product and customer matches. As a rule of thumb, the customer ranks higher than the product and matching individual settings rank higher than group settings. The priorities are in the following order:

1. P + C
2. PG + C

3. P + CG
4. PG + CG
5. P + C*
6. PG + C*
7. P* + C
8. P* + CG
9. P* + C*

P	Product matches exactly
C	Customer matches exactly
PG	Productgroup matches exactly
CG	Customergroup matches exactly
P*	Price matches for all products
C*	Price matches for all customers

In other words:

- Customer-specific PriceRows override product-specific PriceRows of equivalent priority.
- Customer-specific PriceRows override UserPriceGroup-specific PriceRows, which override global PriceRows.
- Product-specific PriceRows override ProductPriceGroup-specific PriceRows, which override global PriceRows.

Enabling Legacy Query Strategy

Use the following property with the `true` value to enable the legacy query strategy for fetching user group and product ID prices parameters in `DefaultPDTRowsQueryBuilder`:

```
europe1.use.legacy.productid.query.strategy=
```

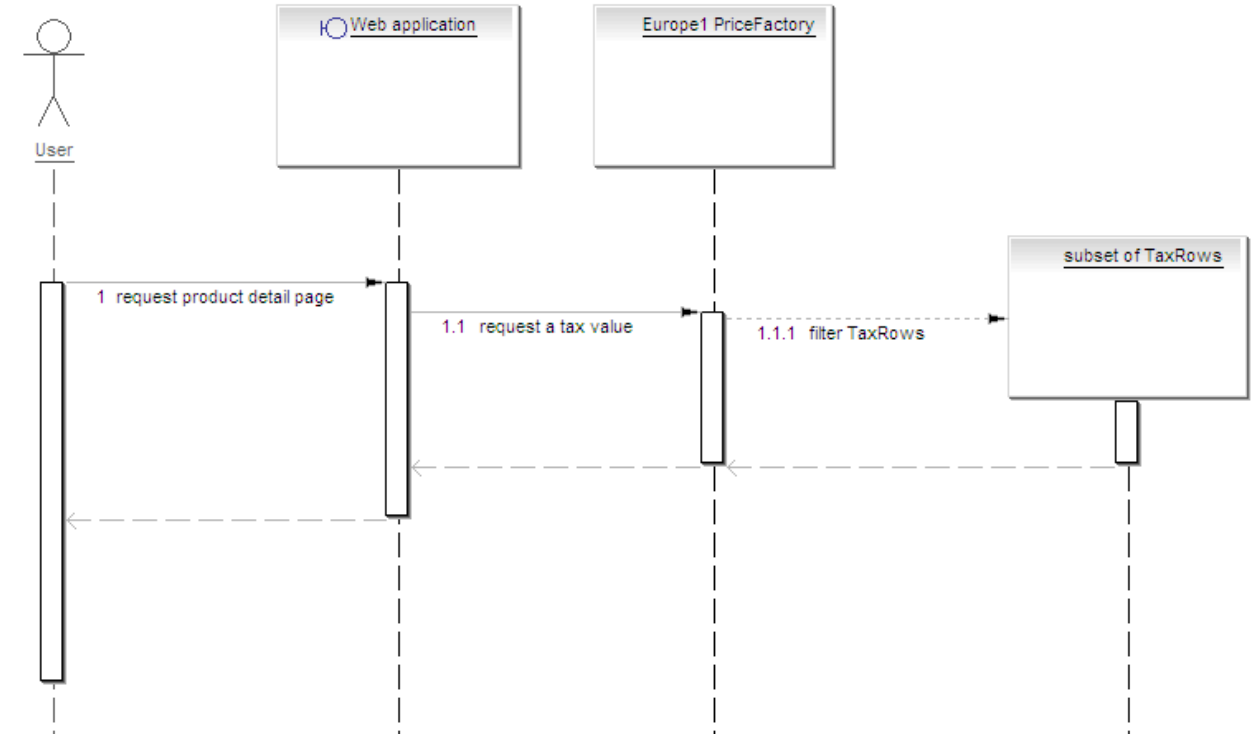
The property is set to `false` by default.

Tax Calculation

SAP Commerce stores all possible taxes for products and customers as TaxRow objects. Every TaxRow object represents a certain tax constellation, which is optionally related to a certain date range, customer, product, group of customers, or a group of products.

Every TaxRow object is a potential tax. Unlike prices, which can only be applied one at a time, taxes and discounts accumulate. If more than one tax or discount is eligible for a certain order entry or an entire order, every single of those taxes and discounts will take effect. For the case of taxes, let's assume the customer **cust** is assigned an individual tax rate of 20% and the product **prod** is also assigned an individual tax rate of 20%. If **cust** orders **prod**, the order entry for **prod** contains two tax entries of 20% - one from **cust** and one from **prod**. If **cust** were to order the product **otherprod** (which has no individual tax rate), the order entry for **otherprod** would contain one tax entry of 20% - the one from **cust**.

When a tax is required, the Europe1 PriceFactory retrieves all TaxRow objects related to the combination of customer and products. The following diagram shows the basic course of action.



To handle a request for a tax for a product, the Europe1 PriceFactory retrieves all TaxRow objects that match the combination of user and product.

Attributes of a TaxRow

The following table provides an overview of the attributes of a TaxRow object.

Attribute Name	Attribute Type	Mandatory	Description
absolute	java.lang.Boolean	no	
currency	Currency	no	
dateRange	StandardDateRange	no	Specify a date range for the TaxRow object's validity.
endTime	java.util.Date	no	
pg	ProductTaxGroup	no	Select a ProductTaxGroup the tax is effective for. The discount is applied to all members of this ProductTaxGroup.
product	Product	no	Specify the product the tax is set for.
productMatchQualifier	java.lang.Long	no	
startTime	java.util.Date	no	
tax	Tax	yes	
ug	UserTaxGroup	no	Select a UserTaxGroup the tax is effective for. The discount is applied to all members of this UserTaxGroup.

Attribute Name	Attribute Type	Mandatory	Description
user	User	no	Specify the customer the tax is set for.
userMatchQualifier	java.lang.Long	no	
value	java.lang.Double	no	

Filtering TaxRows

During the filtering phase, the Europe1 PriceFactory selects all TaxRows matching the combination of customer and product. A TaxRow must match all of the following criteria to apply:

- customer, or customer group, or no customer-related value (blank field)
- product, or product group, or no product-related value (blank field)
- date range

If a TaxRow fails to match all criteria, it is not applied and therefore not considered by the Europe1 PriceFactory. The following sections provide details on these criteria.

Customer Matching

If there is a TaxRow that has no Customer or UserTaxGroup assigned to it, that TaxRow is applied to all customers and is therefore effective by default.

To match a customer, a TaxRow must be one of the following:

- Empty
- Set to match the customer
- Set to match the customer's user group.

To match the customer, the TaxRow's **user** attribute must be set to contain the customer.

To match the customer's user group, the TaxRow's **ug** (user tax group) must be set to contain a user group (an instance of **UserTaxGroup**) which contains the customer.

The following table lists some possible combinations and whether the respective TaxRow is applied to the customer.

TaxRow Assigned To	Current Customer	TaxRow Applies to the Current Customer?
kotal	kotal	✅
kotal	abel	➖
customergroup	kotal (member of customergroup)	✅
customergroup	abel (not a member of customergroup)	➖
(no assignment)	kotal	✅
(no assignment)	abel	✅

TaxRow Assigned To	Current Customer	TaxRow Applies to the Current Customer?
(no assignment)	kotal (member of customergroup)	✅
(no assignment)	abel (not a member of customergroup)	✅

In other words:

- If the current Customer is different from the Customer the TaxRow is assigned to, the TaxRow is not applied.
- If the TaxRow is assigned to a UserTaxGroup and the current Customer is not a member of this UserTaxGroup, the TaxRow is not applied.

Product Matching

If there is a TaxRow that has no Product or ProductTaxGroup assigned to it, that TaxRow is applied to all products and is therefore effective by default.

To match a product, a TaxRow must be one of the following:

- Empty
- Set to match the product
- Set to match the product's tax group

To match the product, the TaxRow's **product** attribute must be set to contain the product.

To match the product's tax group, the TaxRow's **pg** (product tax group) must be set to contain a tax group (an instance of **ProductTaxGroup**) which contains the product.

The following table presents some possible combinations and whether the respective TaxRow is applied to the product.

TaxRow Assigned to	Current Product	TaxRow Applies to the Current Product?
B00005LJ7N-1	B00005LJ7N-1	✅
B00005LJ7N-1	C232134_0	➖
cameragroup	B00005LJ7N-1 (member of cameragroup)	✅
cameragroup	C232134_0 (not a member of cameragroup)	➖
(no assignment)	B00005LJ7N-1	✅
(no assignment)	C232134_0	✅
(no assignment)	B00005LJ7N-1 (member of cameragroup)	✅
(no assignment)	C232134_0 (not a member of cameragroup)	✅

In other words:

- If the current Product is different from the Product the TaxRow is assigned to, the TaxRow is not applied.
- If the TaxRow is assigned to a ProductTaxGroup and the current Product is not a member of this ProductTaxGroup, the TaxRow is not applied.

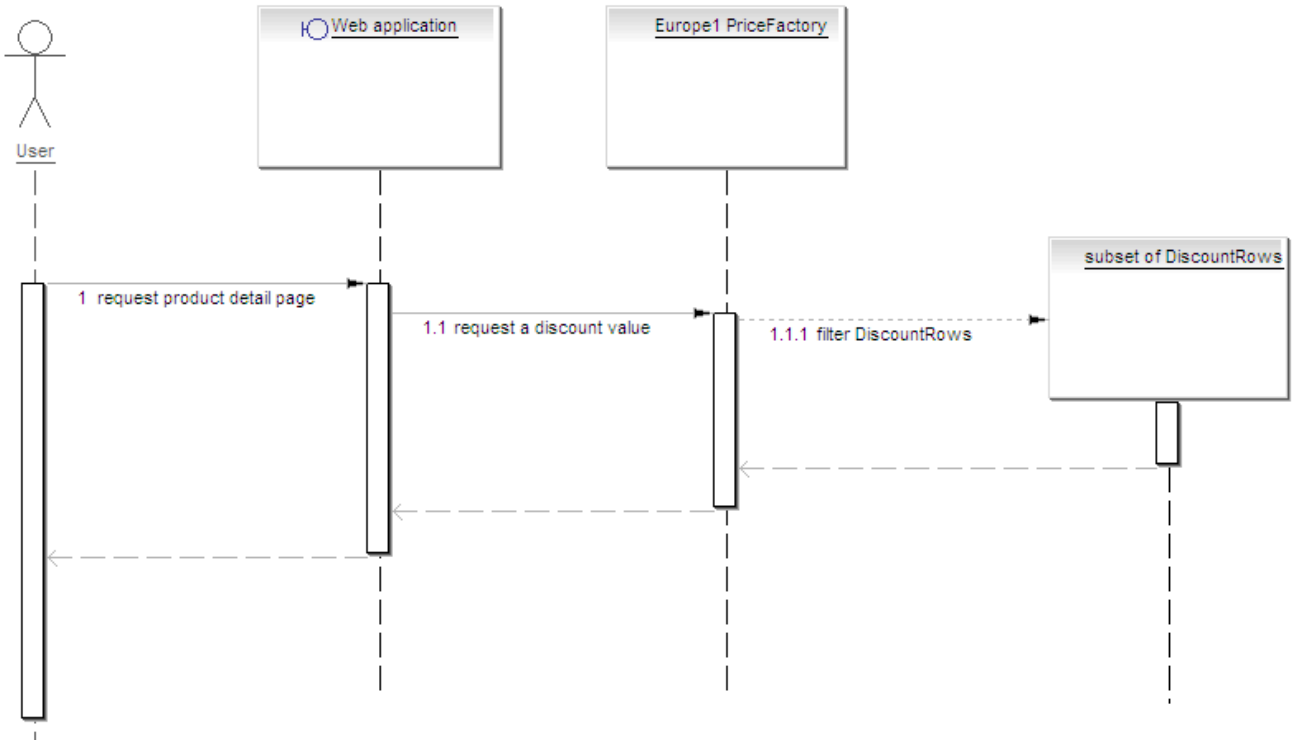
Discount Calculation

SAP Commerce stores all possible discounts for products and customers as DiscountRow objects. EveryDiscountRow represents a certain discount constellation, which is optionally related to a certain date range, customer, product, group of customers, or group of products.

Every DiscountRow object is a potential discount. Unlike prices, which can only be applied one at a time, taxes and discounts accumulate. If more than one tax or discount is eligible for a certain order entry or an entire order, every single one of those taxes and discounts are taken into effect. In the case of discounts, let's assume the customer **cust** is assigned an individual discount rate of 20% and the product **prod** is also assigned an individual discount rate of 20%. If **cust** orders **prod**, the order entry for **prod** contains two discount entries for 20% - one from **cust** and one from **prod**. If **cust** orders the product **otherprod** (which has no individual discount rate), the order entry for **otherprod** contains one discount entry for 20% - the one from **cust**.

If you want to reduce or remove a discount assigned to a product or a customer, you have to assign another discount with a negative value to that product or customer. For example, a discount of -15% increases the price by 15%.

When a discount is required, the Europe1 PriceFactory retrieves all DiscountRow objects related to the combination of customer and products. The following diagram shows the basic course of action.



To handle a request for a discount for a product (**1** and **1.1**), the Europe1 PriceFactory retrieves all DiscountRow objects that match the combination of user and product (**1.1.1**).

Attributes of a DiscountRow

The following is an overview of the DiscountRow object attributes.

Attribute Name	Attribute Type	Mandatory	Description
absolute	java.lang.Boolean	no	Jalo-only attribute
currency	Currency	no	
dateRange	de.hybris.platform.util.StandardDateRange	no	Jalo-only attribute
discount	Discount	yes	
discountString	java.lang.String	no	Jalo-only attribute
endTime	java.util.Date	no	
pg	ProductDiscountGroup	no	
product	Product	no	
productMatchQualifier	java.lang.Long	no	
startTime	java.util.Date	no	
user	User	no	
userMatchQualifier	java.lang.Long	no	
ug	UserDiscountGroup	no	
value	java.lang.Double	no	

If neither a customer nor a UserDiscountGroup is given for a certain DiscountRow, the specified discount applies to all customers in SAP Commerce. If neither a product nor a ProductDiscountGroup is given for a certain DiscountRow, the specified discount applies to all products in SAP Commerce.

The following is an overview of the GlobalDiscountRow object attributes.

Attribute Name	Attribute Type	Mandatory	Description
absolute	java.lang.Boolean	no	Jalo-only attribute
currency	Currency	no	
discount	Discount	yes	
discountString	java.lang.String	no	Jalo-only attribute
dateRange	de.hybris.platform.util.StandardDateRange	no	Jalo-only attribute
endTime	java.util.Date	no	
pg	ProductDiscountGroup	no	
product	Product	no	
productMatchQualifier	java.lang.Long	no	
startTime	java.util.Date	no	
ug	UserDiscountGroup	no	
user	User	no	

Attribute Name	Attribute Type	Mandatory	Description
userMatchQualifier	java.lang.Long	no	
value	java.lang.Double	no	

If neither a customer nor a UserDiscountGroup is given for a certain GlobalDiscountRow, the specified discount applies to all customers in SAP Commerce. If neither a product nor a ProductDiscountGroup is given for a certain GlobalDiscountRow, the specified discount applies to all products in SAP Commerce.

Filtering DiscountRows

During the filtering phase, the Europe1 PriceFactory selects all DiscountRows matching the combination of customer and product. A DiscountRow must match all of the following criteria to apply:

- customer or customer group or no customer-related value (blank field)
- product or product group or no product-related value (blank field)
- date range

If a DiscountRow fails to match all the criteria, it is not applied and therefore not considered by the Europe1 PriceFactory. The following sections provide details on these criteria.

Customer Matching

If there is a DiscountRow that has no Customer or UserDiscountGroup assigned to it, that DiscountRow is applied to all the customers and is therefore effective by default.

To match a customer, a DiscountRow must be one of the following:

- Empty
- Set to match the customer
- Set to match the customer's user group

To match the customer, the DiscountRow's **user** attribute must be set to contain the customer.

To match the customer's user group, the DiscountRow's **ug** (user discount group) must be set to contain a user group (an instance of **UserDiscountGroup**) which contains the customer.

The following table lists some possible combinations and whether the respective DiscountRow is applied to the customer or not.

DiscountRow assigned to	Current Customer	DiscountRow applies to the current Customer?
kotal	kotal	✅
kotal	abel	⛔
customergroup	kotal (member of customergroup)	✅
customergroup	abel (not a member of customergroup)	⛔

DiscountRow assigned to	Current Customer	DiscountRow applies to the current Customer?
(no assignment)	kotal	✅
(no assignment)	abel	✅
(no assignment)	kotal (member of customergroup)	✅
(no assignment)	abel (not a member of customergroup)	✅

In other words:

- If the current Customer is different from the Customer the DiscountRow is assigned to, the DiscountRow is not applied.
- If the DiscountRow is assigned to a UserDiscountGroup and the current Customer is not a member of this UserDiscountGroup, the DiscountRow is not applied.

Product Matching

If there is a DiscountRow that has no Product or ProductDiscountGroup assigned to it, that DiscountRow is applied to all products and is therefore effective by default.

To match a product, a DiscountRow must be one of the following:

- Empty
- Set to match the product
- Set to match the product's Discount group

To match the product, the DiscountRow's **product** attribute must be set to contain the product.

To match the product's discount group, the DiscountRow's **pg** (product discount group) must be set to contain a discount group (an instance of **ProductDiscountGroup**) which contains the product.

The following table displays some possible combinations and whether the respective DiscountRow is applied to the product.

DiscountRow Assigned To	Current Product	DiscountRow Applies to the Current Product?
B00005LJ7N-1	B00005LJ7N-1	✅
B00005LJ7N-1	C232134_0	➖
cameragroup	B00005LJ7N-1 (member of cameragroup)	✅
cameragroup	C232134_0 (not a member of cameragroup)	➖
(no assignment)	B00005LJ7N-1	✅
(no assignment)	C232134_0	✅
(no assignment)	B00005LJ7N-1	✅

DiscountRow Assigned To	Current Product	DiscountRow Applies to the Current Product?
	(member of cameragroup)	
(no assignment)	C232134_0 (not a member of cameragroup)	✅

In other words:

- If the current Product is different from the Product the DiscountRow is assigned to, the DiscountRow is not applied.
- If the DiscountRow is assigned to a ProductDiscountGroup and the current Product is not a member of this ProductDiscountGroup, the DiscountRow is not applied.

Time-Dependent Prices and Discounts

The types that define prices, taxes, and discounts have a common supertype, the abstract PriceDiscountTaxRow, or `PDTRow` type.

The `PDTRow` type optionally accepts an attribute named `daterange` of type `de.hybris.platform.util.DateRange` (time span), whose range defines when the respective PriceRow, TaxRow, or DiscountRow instance applies. If the current date is earlier than the start date or later than the end date, respectively, the price, tax, or discount isn't applied. A given date starts by 00:00:00 that day and ends by 23:59:59 that same day. You can't define a DateRange to end by 15:23:34, for example.

Europe1 doesn't allow one of the fields of the date range to be empty. You need to specify a value for both the start and end date, or leave both fields blank.

The following table provides an example.

Start Date	End Date	Daterange Valid
2005-03-16	2005-04-19	March 16, 2005, 00:00:00 through April 19, 2005, 23:59:59
(blank)	2005-04-19	***(not possible)***
2005-03-16	(blank)	***(not possible)***
(blank)	(blank)	Always valid

Defining Date Ranges

To assign a DateRange to a price or tax definition, create an instance of a DateRange, get the instance of the current PriceFactory, and refer the instantiated DateRange as a parameter in the call for the **createPriceRow(...)** or **createDiscountRow(...)** method, respectively, as in the following code snippet:

```
Date startDate = new Date( 2005, 06, 12 );
                                    Date endDate = new Date( 2006, 02, 01 );
                                    DateRange dateRange = new StandardDateRange( startDate, enc
                                    Europe1PriceFactory pf = (Europe1PriceFactory) OrderManage
                                    pf.createPriceRow( product, productPriceGroup, user, userP
                                    currency, unit, unitFactor, net, dateRange, price );
                                    pf.createDiscountRow( product, productPriceGroup, user, use
```

When you create price rows with start and end dates, make sure that they don't overlap. Otherwise, it's not guaranteed which price is chosen when Europe1 is asked for prices within the overlapping period.

As Dates in Java are precise to the millisecond, you should declare the end time and the start time of a successive price row to be at least one millisecond apart, as shown in this example:

```
PriceRowModel beforePrice = modelService.create(PriceRowModel.class);
                                                    beforePrice.setStartTime( new Date(1L) );
                                                    beforePrice.setEndTime( new Date(9999L) );
                                                    ...
                                                    PriceRowModel afterPrice = modelService.cre
                                                    afterPrice.setStartTime( new Date(10000L) )
                                                    afterPrice.setEndTime( new Date(20000L) );
                                                    ...
                                                    modelService.saveAll(beforePrice, afterPric

                                                    // or using DateRange


                                                    PriceRowModel beforePrice = modelService.cr
                                                    beforePrice.setDateRange( new StandardDateR
                                                    ...
                                                    PriceRowModel afterPrice = modelService.cre
                                                    afterPrice..setDateRange( new StandardDateR
                                                    ...
                                                    modelService.saveAll(beforePrice, afterPric
```

Defining Open-Ended Date Ranges

To allow for a daterange to be open-ended, for example "since February 1, 2016 until forever", or have no defined start date, for example "since forever in the past until February 1, 2017 ", set a fixed date and then set the other date to a point in time far in the future or far in the past. As in the following example, you could set it ten years or more from now.

```
                                                // setting a fixed date as a start date for an "ope
                                                Date startDate = new Date( 2016, 02, 01 );
                                                // setting an "open end" date as a time point 10 ye
                                                Date endDate = new Date( System.currentTimeMillis()
                                                DateRange dateRange = new StandardDateRange( startD
```

In this case, you need to make sure that there are no two dateranges of this kind for a given product as the Europe1 PriceFactory wouldn't be able to determine which is the better match.

Overriding and Testing Settings

You can override customer-related and product-related group settings.

Overriding Group Settings

It is possible to override customer-related and product-related group settings on both the session and cart level.

Session-Level Customer Group Settings

For customer-related group settings, that is customer tax, discount, and price groups, there's another way to influence `Europe1`. To override the grouping attributes stored with the customer itself, use `SessionService` to set them for the enclosing session:

```
// setting customer price group
                                                    sessionService.setAttribute(de.hybris.platf
                                                    // setting customer tax group
                                                    sessionService.setAttribute(de.hybris.platf
                                                    // setting customer discount group
                                                    sessionService.setAttribute(de.hybris.platf
```

Once there are values set at session level, `Europe1` will ignore the values at the customer items. This is particularly helpful for technical users (Anonymous) when influencing prices, taxes or discounts because they can represent various real customer groups.

Cart-Level Product Group Settings

You can also override product-related group settings. However, this is done at order and cart entry levels instead of a session level as is the case with customer groups. This way each cart or order entry can be given individual product price, tax and discount groups.

```
// setting product price group
                                                    cartEntry.setEurope1PriceFactory_PPG(..);
                                                    // setting product tax group
                                                    cartEntry.setEurope1PriceFactory_PTG(..);
                                                    // setting product tax group
                                                    cartEntry.setEurope1PriceFactory_PDG(..);
```

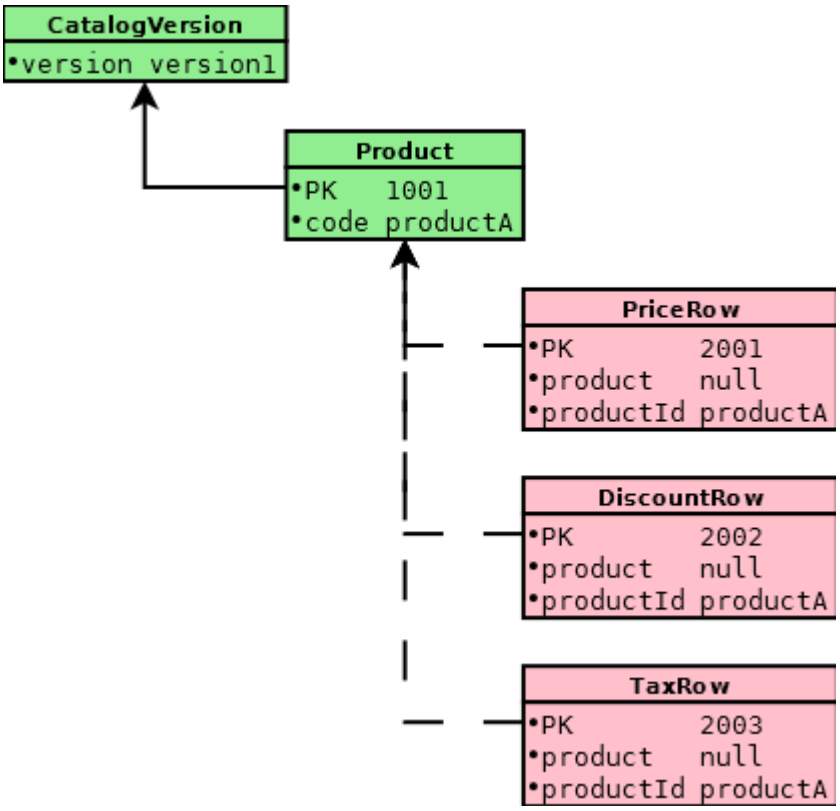
Related Information

Ordering Process
payment Extension

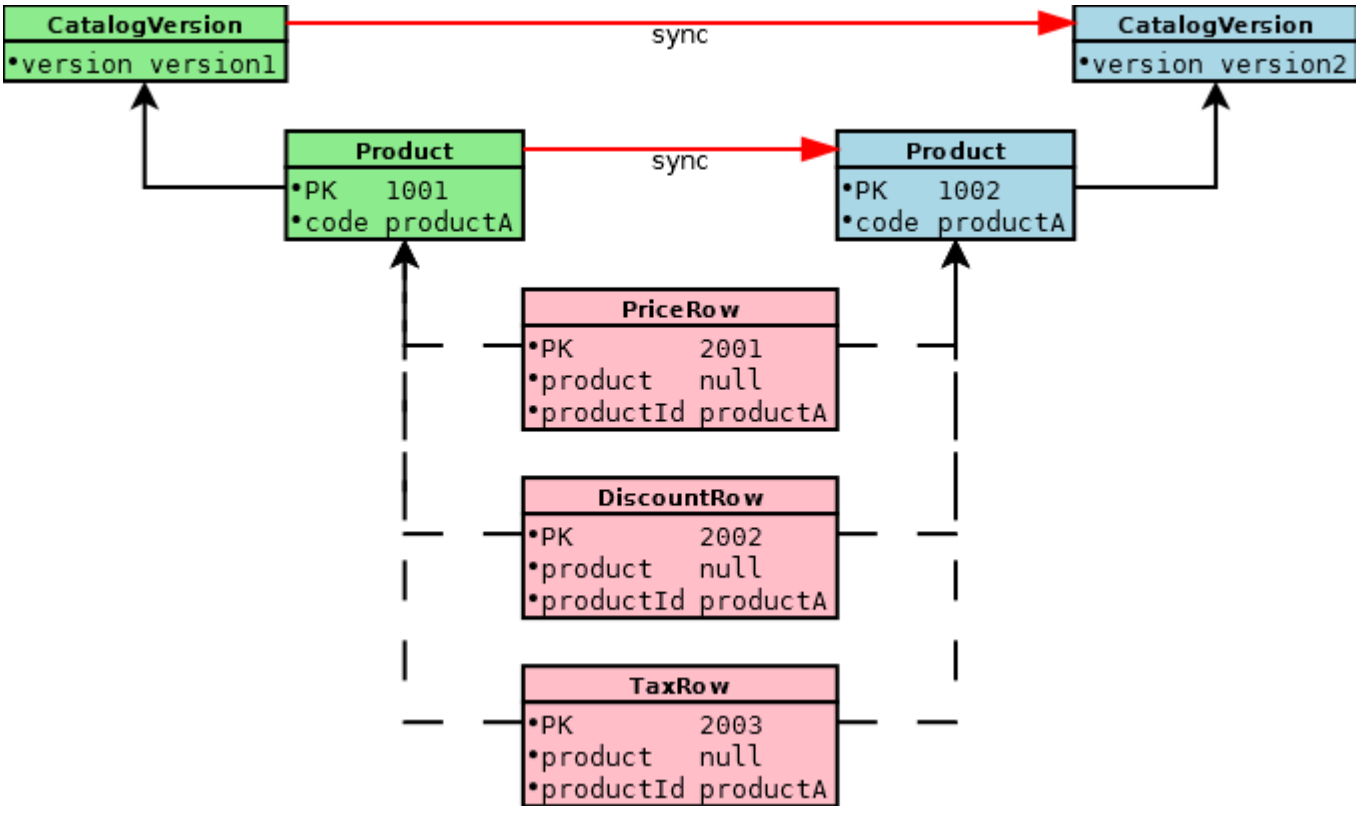
Decoupling PDTRows from Product

Find out how decoupling `PDTRows` from the `Product` affects synchronization.

It is possible to create `PDTRow` as separate entities and connect them with a given `Product` not by a hard reference using PK but rather through a kind of soft reference using the `Product` identifier, which in the default implementation means `Product` code. This diagram shows `PDTRow` referencing `Product` by `productId`:



After the synchronization it looks accordingly:



ℹ Note

Bear in mind that none of the PDTRows has been synchronized, and products, even in different catalog versions share the same instance of PDTRow.

Impact on the Synchronization Process

Let's first look at the old way of importing products using your `product.impex` file:

product.impex

```
$catalog-id=Default
$catalog-version=Staged

$catalogversion=catalogversion(catalog(id),version)[unique=true,default=$catalog-id:$catalog-versio
$prices=europe1prices[translator=de.hybris.platform.europe1.jalo.impex.Europe1PricesTranslator]

INSERT_UPDATE Product;code[unique=true];$catalogversion;name[lang=en];unit(code);$prices;approvalSt
;product1;;product1;pieces;1 EUR;approved;admin;0
;product2;;product2;pieces;2 EUR;approved;admin;1
etc...
```

This import file follows the old convention. If you have a file with 10000 rows, after syncing catalogs (staged to online) make a query for the number of products and prices; you can find that there is 20000 products and 20000 prices. Synchronization takes about 5 minutes.

Let's try with an import file that follows the new convention.

product.impex

```
$catalog-id=Default
$catalog-version=Staged

$catalogversion=catalogversion(catalog(id),version)[unique=true,default=$catalog-id:$catalog-versio

INSERT_UPDATE Product;code[unique=true];$catalogversion;name[lang=en];unit(code);approvalStatus(coc
;product1;;product1;pieces;approved;admin;0
;product2;;product2;pieces;approved;admin;1
etc...

INSERT_UPDATE PriceRow;productId[unique=true];price;unit(code);currency(isocode)
;product1;1;pieces;EUR
;product2;2;pieces;EUR
etc...
```

If you import this file and synchronize the catalogs, the synchronization is much faster. The query for the number of products yields 20000 but the number of prices is only 10000. This is because only products are synchronized, not prices.

Editing Tax and Discount Values of an Order in Backoffice

You can edit tax and discount values of an order in Backoffice. We assume that you apply taxes per line item - at order entry level. It means a tax is calculated on the total selling price of an individual item. Our examples show discounts applied both at order entry, and order levels.

Requirements

To follow the tutorial, create the following items:

- Product
- Order
- Order entry

Below you can find instructions how to do it.

Creating a Product

Procedure

1. Log into Backoffice.
2. In the Explorer tree, click Catalog > Products to open the product collection browser.
3. In the product collection browser, click + Product to open the Create New Product window.
4. Complete all required fields for your product.
5. Click Done to complete creating a new product.

Creating an Order

Procedure

1. Log into Backoffice.
2. In the Explorer tree, click Order > Orders to open the order collection browser.
3. In the order collection browser, click + Order to open the Create New Order window.
4. Complete all required fields for your order.
5. Click Done to complete creating an order.

Creating an Entry for Your Order

Context

When creating an order entry, you assign it an order and a product. Create an order and a product in advance. You can start creating an entry from within an order editor.

Procedure

1. Log into Backoffice.
2. In the Explorer tree, click Order > Orders to open the order collection browser.
3. In the order collection browser, click the name of the order for which you want to create an entry to open the order editor.
4. Make sure that the order editor view is switched to POSITIONS AND PRICES.
5. In the Entries field, click Create new Order Entry to open the Create new Order Entry window:

6. Complete all required fields for your order entry. Point to the product and the order you created earlier to assign them to the entry.
7. Click Done to complete creating your order entry.

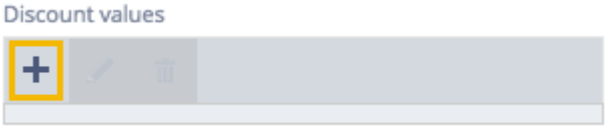
Assigning a Tax Rate and Discount Value for your Entry

Context

Apart from assigning a tax rate or a discount value, set a base price for the product assigned to your entry. You can access your entry through the order editor.

Procedure

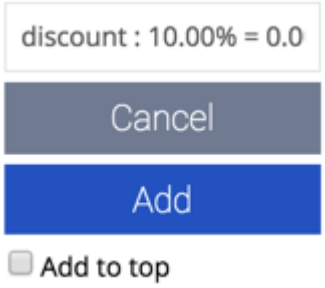
1. Log into Backoffice.
2. In the **Explorer tree**, click **Order** > **Orders** to open the order collection browser.
3. Click the order that has the entry you want to edit to open the order editor.
4. Make sure that the order editor view is switched to **POSITIONS AND PRICES**.
5. In the **Entries** field, click your entry to open the entry editor.
6. Set the base price value for your product in the **Base Price** field.
7. Click **Add new item** .

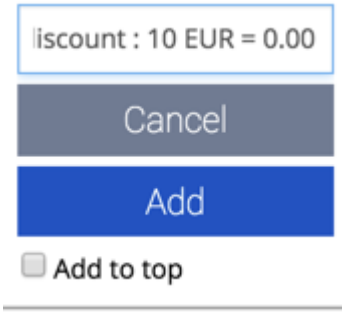
   A window opens where you can provide your values to define discounts or taxes - see the next step.
8. Define your discounts or taxes.

   When providing a value for discounts or taxes, follow the `<code> : <value>(%| <CUR>)[ = <appliedValue>]` pattern, for example, `taxName : 10.00% = 0.10`, or `taxName : 10 EUR`. Provide and edit the parameters before the = sign only - the `appliedValue` is calculated automatically.

   This example shows how to set a 10% discount, or 10% tax:

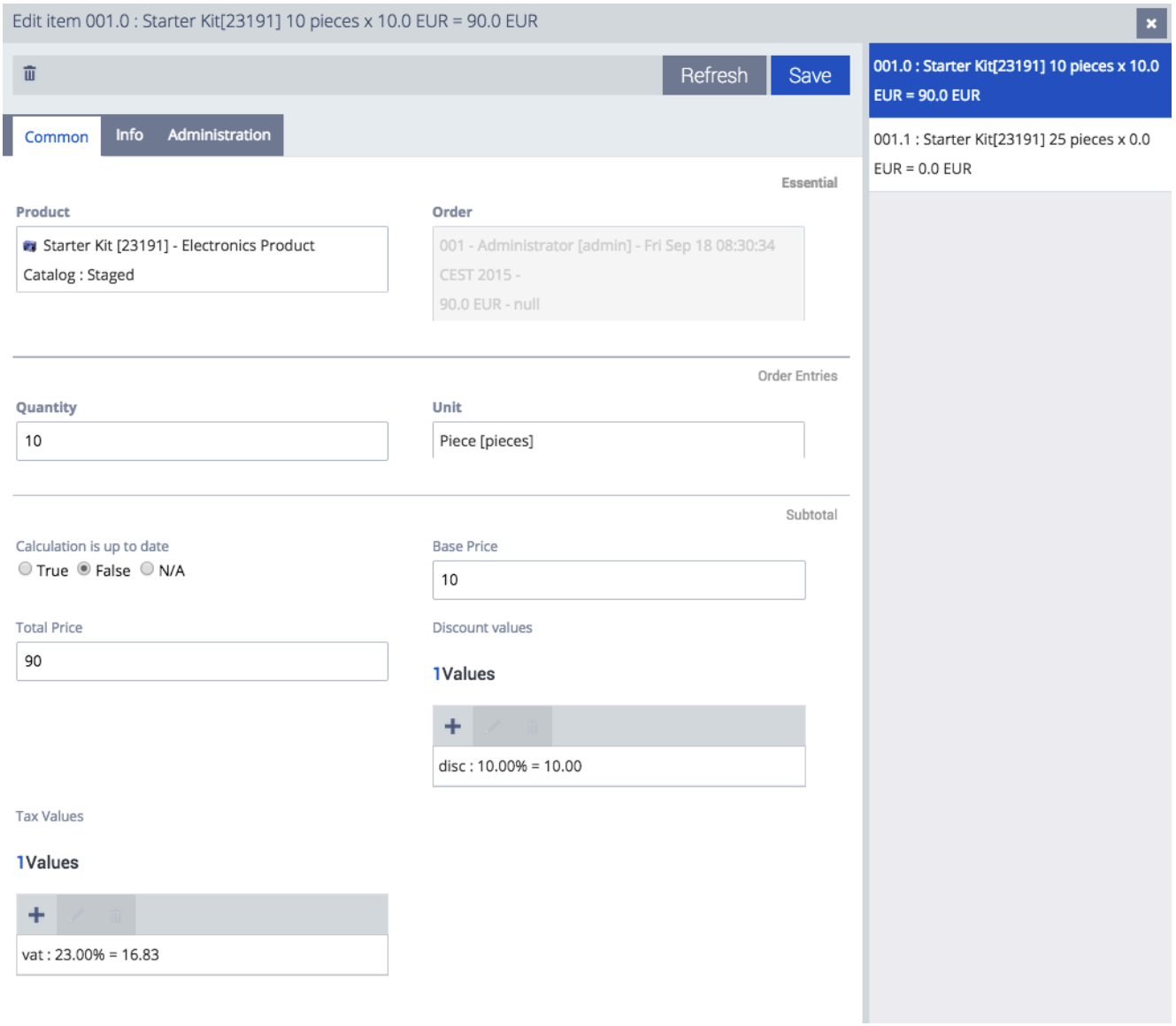
   This example shows how to set a 10 EUR discount:

9. Click **Add** to complete adding discounts or taxes.
10. Click **Save** in the entry editor. Close it to switch back to the order collection browser.
11. In the order collection browser, check the **False** option in the **Calculation is up to date** field

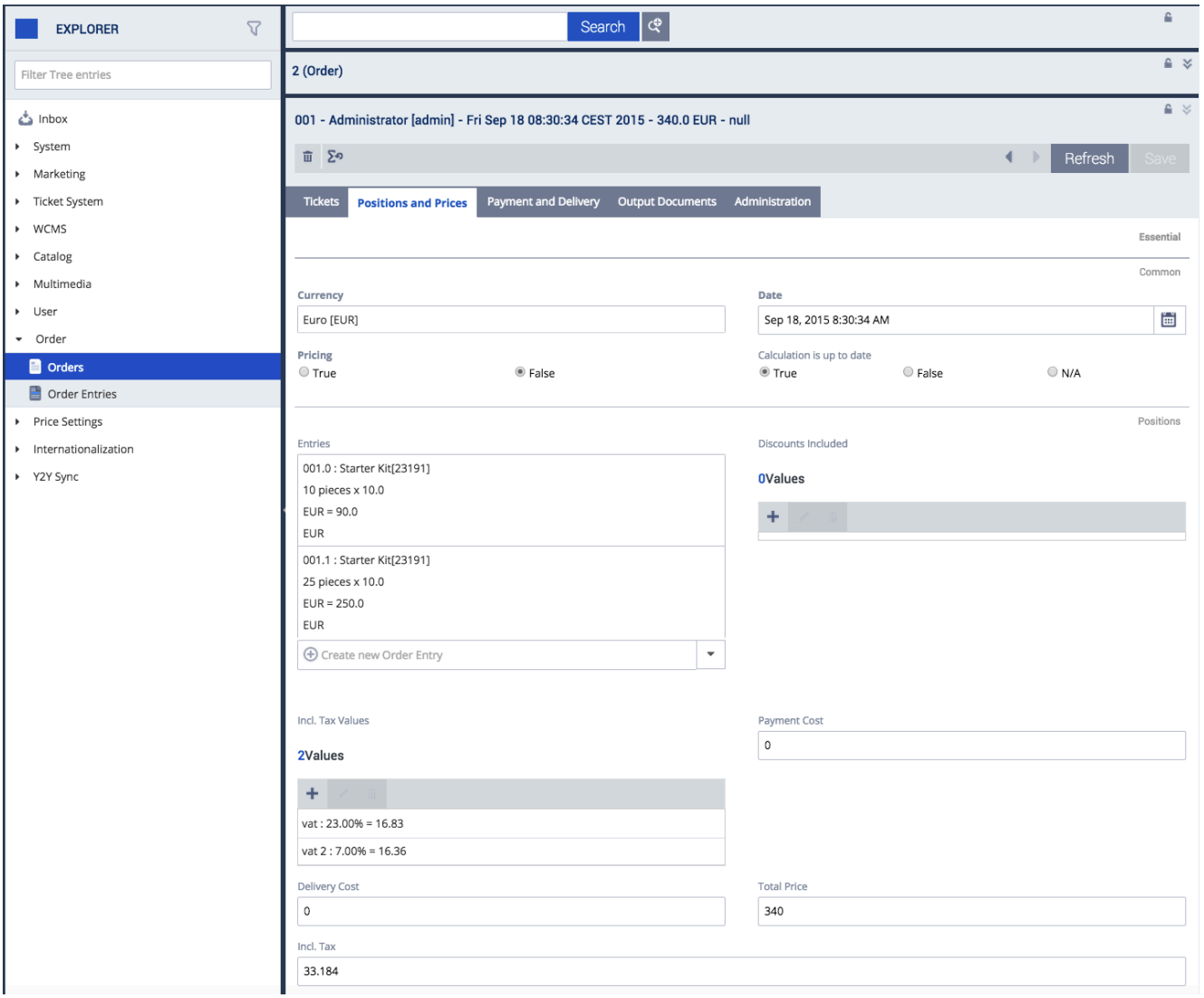
12. Click Save.
13. Click  to recalculate your order.
14. Refresh the view.
15. Open your order entry through the order editor to make sure that the values have been saved and recalculated.

    Below you can see an example entry with a discount and tax set at 10% and 23% respectively. You can also see that the total price has been recalculated too:



16. Switch the view to the order editor to see the final calculations at an order level.

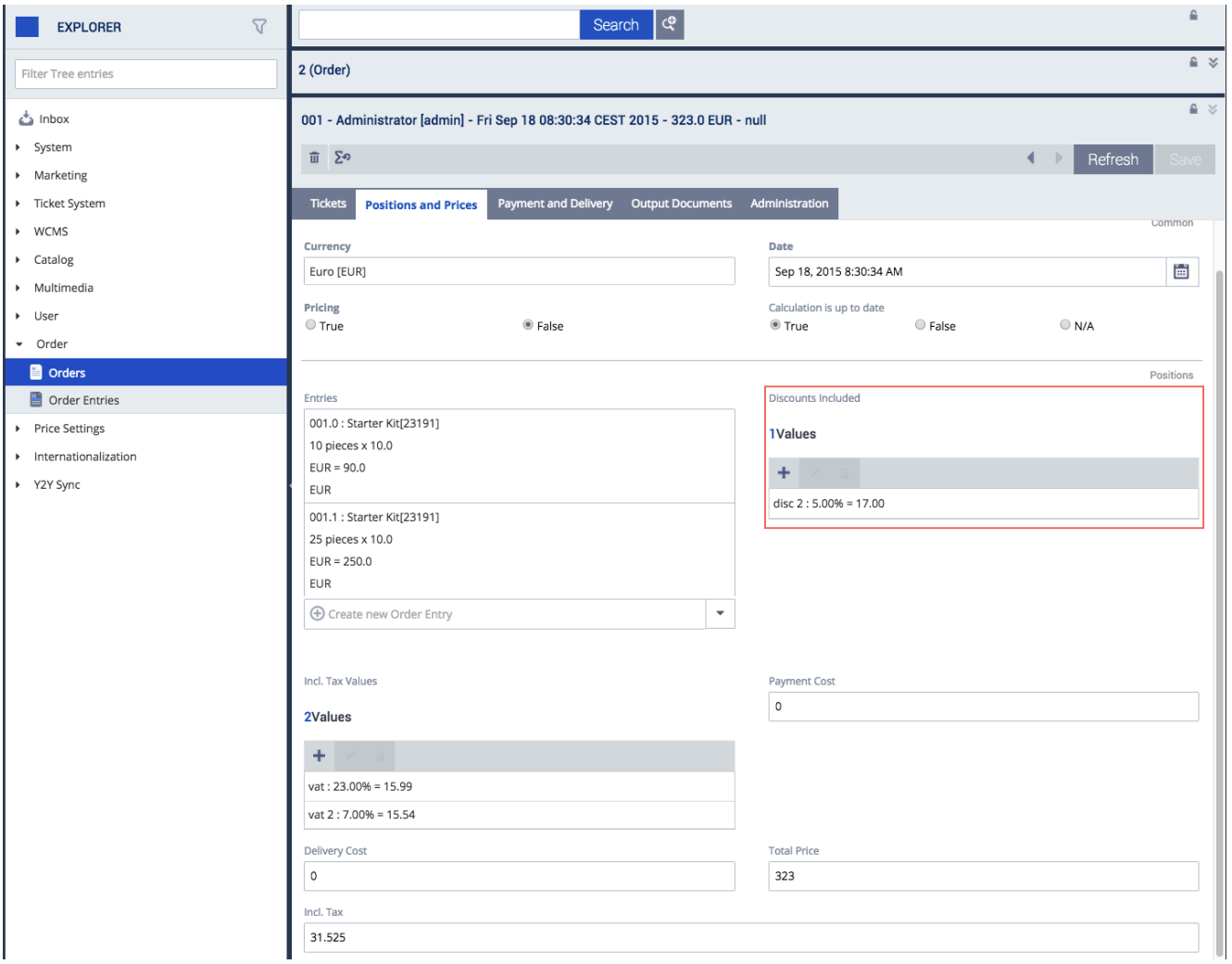
    Below you can see an example order that consists of 2 entries. In both these entries tax rates have been applied and you can see them listed in the order editor. Discounts applied at entry level are not listed in the order editor. They are, however, applied and reflected in an entry Total Price.



Notice that when **Pricing** in the order collection browser is set to **True**, values are calculated as net values.

Assigning a Discount at Order Level

Unlike tax rates, you can apply discounts at order level - such a discount affects the entire order. To apply a discount, use the order editor.



Related Information

Price, Tax, and Discount Calculation

Target Price Discount

When you apply a target price discount, you set its value as a requested target price for a unit of a given product.

A target price discount value specifies a target selling price as opposed to a typical price reduction as is the case with relative or absolute discounts. A target price discount allows you to easily implement use cases such as "Sell for €9.99 regardless of the current base price". It thus relieves project developers from calculating discounts manually. Note that not only the calculation was extended to support a target price discount. Also `Europe1` allows you to set up `DiscountRow` items containing target price discounts.

First see our guidelines for setting up a test order. We need it only to show you how a target price discount works. In our examples we create a target price discount using `Europe1`, and then manually.

Setting Up a Test Order

To create a test order, you need the following items:

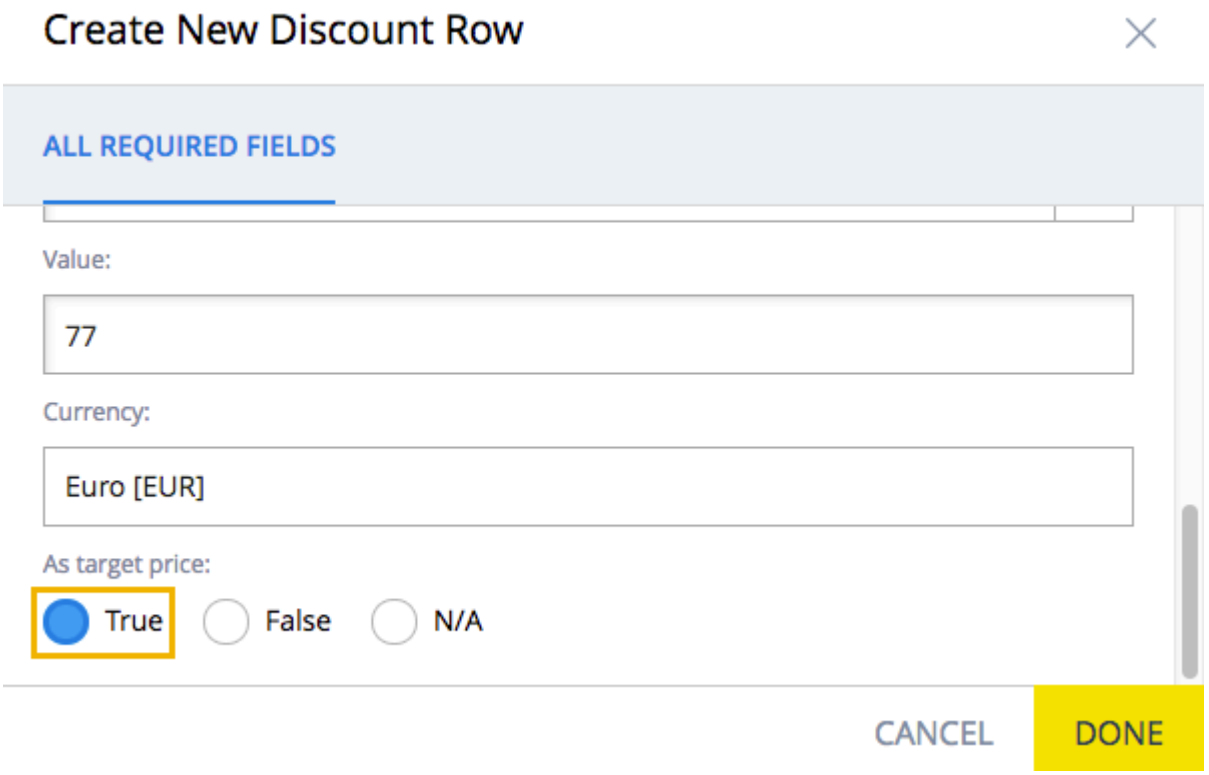
- `Product`: When creating a product, provide product identification information such as for example an article number, a catalog version, or a product description.

- `PriceRow`: Through a price row you provide a price value (**Price**) for a product. When creating a price row, point to your product. In this way you bind it to the price row and thus to the defined price value. Choose `Anonymous` for **Customer**, and complete the **Unit**, and **Currency** fields.
- `Order`, and `OrderEntry`: When creating an order, use `Anonymous` for **User**. When creating an order entry, point to your order and product. In this way you bind your product to the order. Complete the **Unit** and **Quantity** fields.

Creating a Target Price Discount Using Europe1

To create a discount, you need `Discount` and `DiscountRow` items.

Create a discount item first. When creating a discount row, point to your discount and product items. In this way you bind all those items together. Additionally, complete the **Value**, and **Currency** fields. Set the **As target price** attribute to **True** so that the discount value you just provided is treated as a target price discount:



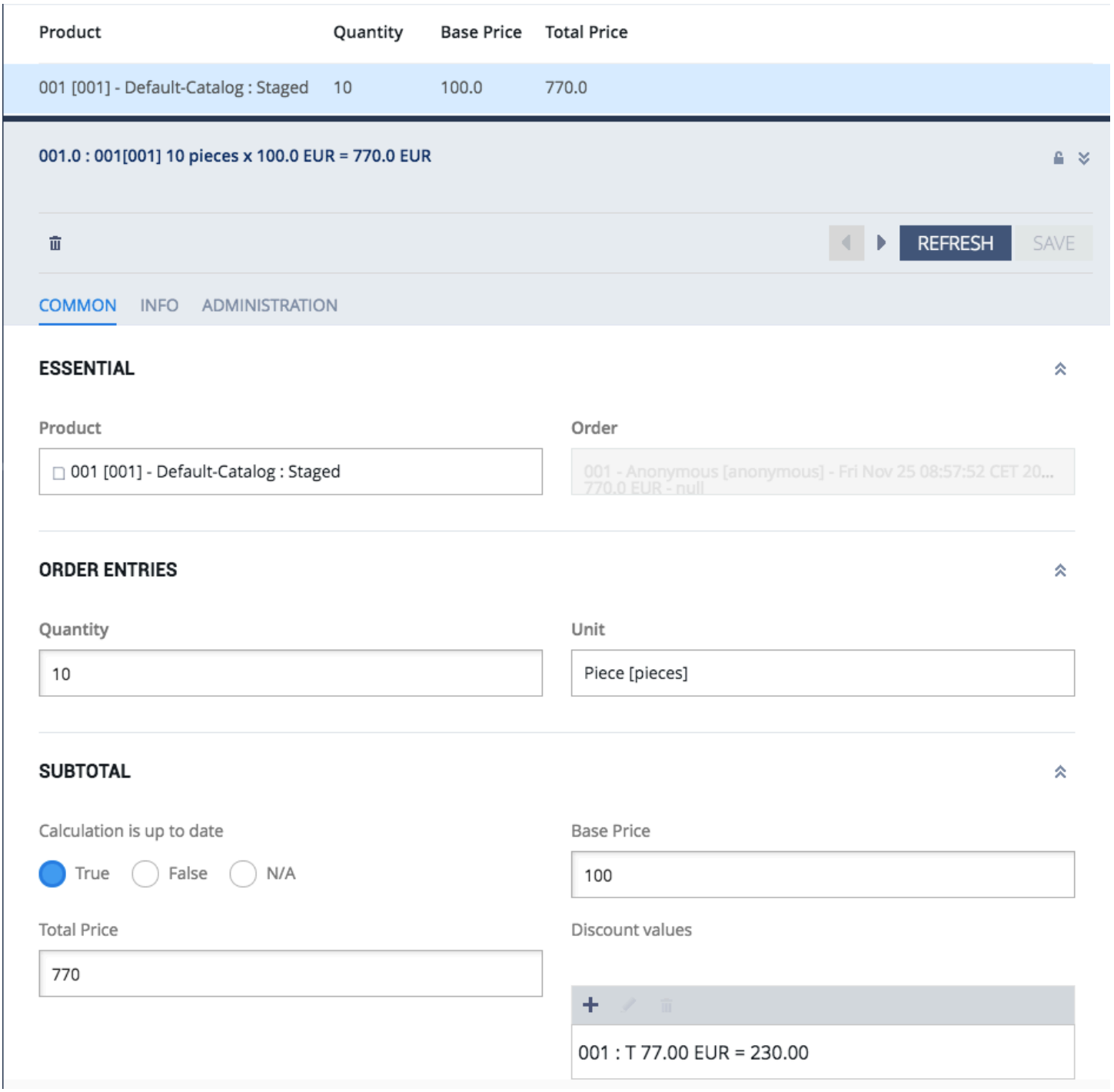
Recalculating Your Order

If you have configured your order properly, your product is bound to it through the order entry. You have already bound the price to the product, and your order entry defines a number of pieces of the product. You are ready to recalculate your order. To recalculate your order, use the following script in Administration Console:

```
import de.hybris.platform.core.model.order.OrderModel

OrderModel order=[code:"001"]
order=flexibleSearchService.getModelByExample(order)

calculationService.recalculate(order)

println "Recalculated order $order.code - new total is $order.totalPrice"
```

Remember to provide a proper value for your order code. In our example it is `001`.

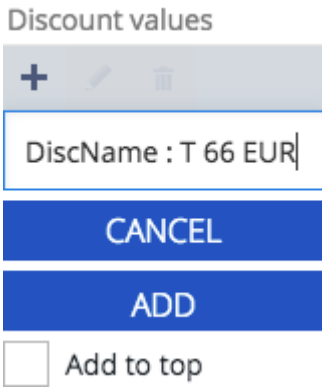
In the screenshot you can see our recalculated order entry. The entry includes 10 pieces of product 001 at the base price of EUR 100 a piece. However, our target price discount sets a target price to EUR 77 a piece:



Creating a Target Price Discount Manually

You can create a target price discount manually in your order entry using the **Discount Values** feature. Use a special expression for adding discounts. When inserting a value for a discount, follow this expression pattern: **<code> : T <value>(<CUR>)[= <appliedValue>]**, for example `DiscountName : T 10 EUR`. Leave the **appliedValue** parameter as is, or delete it. It is calculated automatically when you recalculate your order. Letter **T** in the expression marks your discount value as a target price discount. For more information, see Editing Tax and Discount Values of an Order in Backoffice.

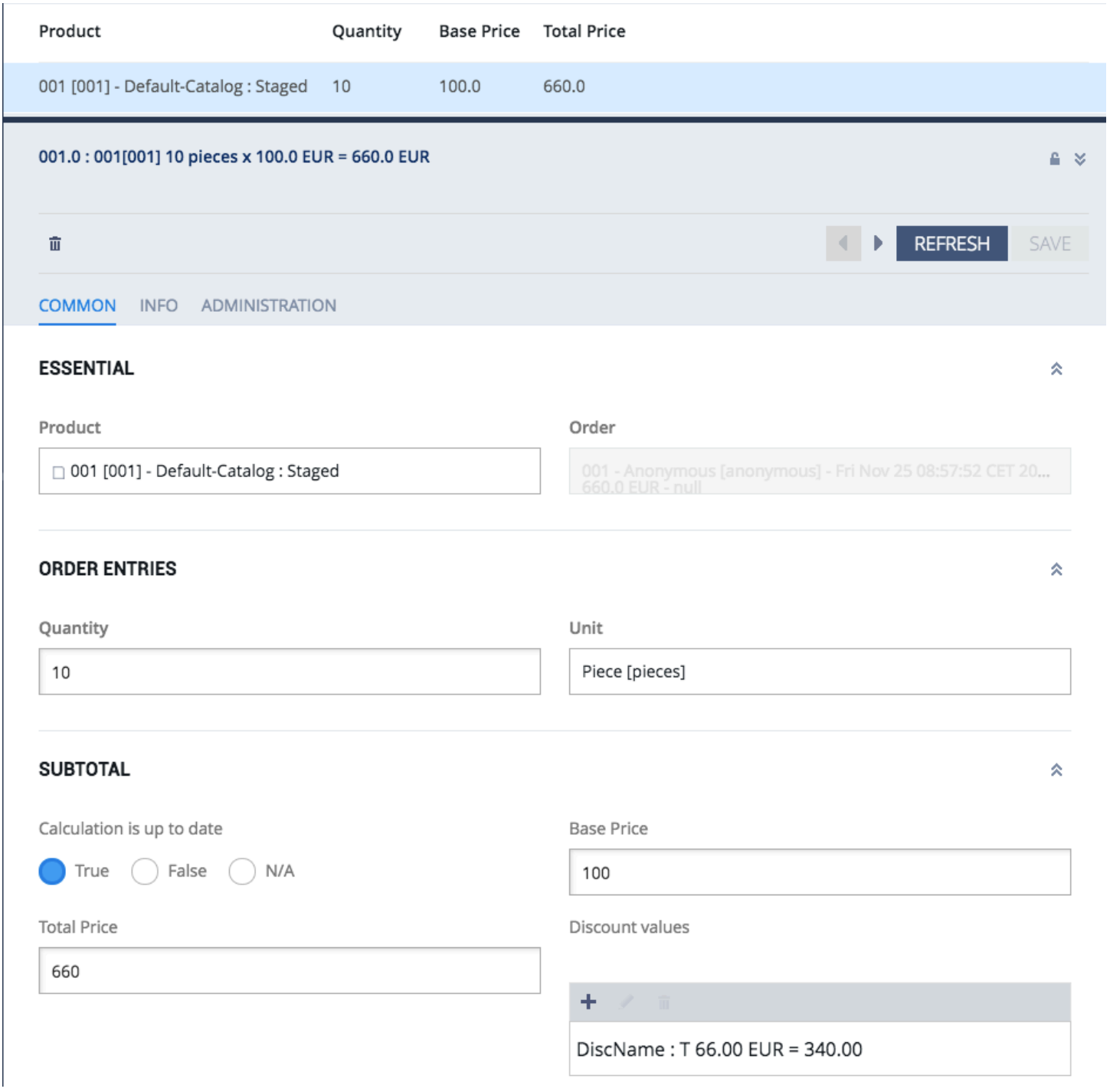
This example shows how to set a 66 EUR target price discount:



Recalculating Your Order

Recalculate your order using the **Recalculate Order Totals** action button available in the order editor for your discount to take effect.

As a result our target price discount sets a target price to EUR 66 a piece:



Extensible Cart Calculation

The extensible cart calculation is a solution based on service layer that can replace the default implementation of PriceFactory (Europe1PriceFactory). It provides an API and default implementations of the services and strategies for resolving prices, discounts, taxes, and delivery and payment costs used in cart (re)calculation.

Extensible cart calculation introduces common, flexible, and easily customizable methods for retrieving information required for cart calculation. On the business logic level, the default extensible cart calculation implementation is equivalent to the default implementation of the `PriceFactory` interface.

For more information about the `PriceFactory`, see Price, Tax, and Discount Calculation.

With the extensible cart calculation, you can implement your own custom pricing strategies. The solution allows every custom extension to contribute to resolving prices, discounts, and taxes.

Service Layer API

The extensible cart calculation is flexible and customizable thanks to its modular architecture. As opposed to the default implementation of PriceFactory (Europe1PriceFactory), it implements the price, discount, and tax resolving strategies as separate strategies.

Design Concepts

The extensible cart calculation uses the generic API and the processor chain design concepts.

Price, Discount, and Tax Services

See the list of Platform Services extended with additional methods to show all prices, taxes, and discounts for given product set in criteria.

Operation Modes

The extensible cart calculation API is enabled by default, in the SMART mode. This mode uses the service layer but under certain circumstances it falls back to jalo.

Migration Guide

See the information related to migrating your cart calculation logic from Jalo to the extensible cart calculation.

Service Layer API

The extensible cart calculation is flexible and customizable thanks to its modular architecture. As opposed to the default implementation of PriceFactory (Europe1PriceFactory), it implements the price, discount, and tax resolving strategies as separate strategies.

`DefaultCalculationService` (the default Platform implementation for `CalculationService`) resolves the information required for cart calculation (prices, taxes, discounts, payment and delivery costs) through respective `Find[Price,Discount,Tax,Delivery,Payment]..Strategies`.

The diagram shows the jalo and the service layer strategies. Here is how they differ:

Jalo

In the jalo `find` strategies, Platform provides a default implementation of the price, discount, and tax strategies in the single `FindPricingWithCurrentPriceFactoryStrategy` class. This class implements the `FindPriceStrategy`, `FindDiscountValuesStrategy`, `FindTaxValuesStrategy` resolver strategy interfaces, and uses a configured `PriceFactory` implementation.

The default implementations for `FindDeliveryCostStrategy(DefaultFindDeliveryCostStrategy)` and `FindPaymentCostStrategy(DefaultFindPaymentCostStrategy)` use jalo items to retrieve delivery and payment cost

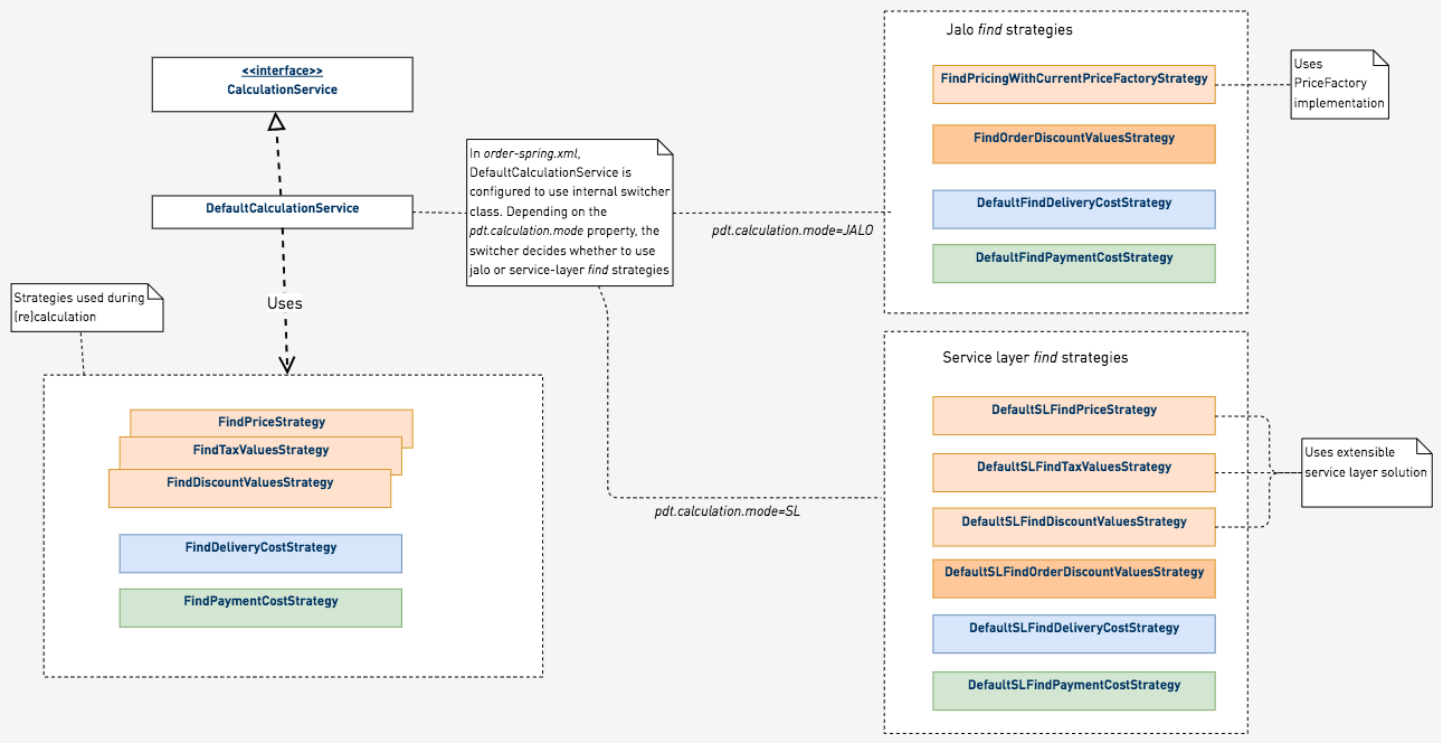
information.

Service layer

In the service layer solution, the implementations of all the strategies used by `DefaultCalculationService` use only the service layer logic and service layer models. Those strategies have the `DefaultSL` prefix in the names of the classes.

The service layer approach provides the same logic as the default implementations of the methods from the `PriceFactory` interface (`Europe1PriceFactory`).

The diagram shows the interfaces/classes and the interdependencies for both approaches:



As opposed to the jalo approach, the service layer solution provides implementations for the interfaces separately:

Interface	Service layer implementation
`FindPriceStrategy`	`DefaultSLFindPriceStrategy`
`FindTaxValuesStrategy`	`DefaultSLFindTaxValuesStrategy`
`FindDiscountValuesStrategy`	`DefaultSLFindDiscountValuesStrategy`
`FindDeliveryCostStrategy`	`DefaultSLFindDeliveryCostStrategy`
`FindPaymentCostStrategy`	`DefaultSLFindPaymentCostStrategy`

Those strategies are located in the `de.hybris.platform.order.strategies.calculation.impl.servicelayer` package.

ℹ Note

If you would like to provide your own implementation of any of the above strategies and you use only service layer logic and models, then override the `boolean isSLOnly()` method and return `true`.

The `boolean isSLOnly()` method is inherited from the `ServiceLayerOnlyCalculationVerifier` interface. As a result of all the strategies returning `true` in `boolean isSLOnly()`, `DefaultCalculationService` optimizes the number of save operations.

`DefaultCalculationService` has a configured switcher that allows you to still use the jalo approach. This switcher decides whether to use either jalo strategies or service layer strategies based on the `pdt.calculation.mode` property. For more information, see Operation Modes.

Extending DefaultSLFindPriceStrategy through the FindPriceHook Interface

You can extend `DefaultSLFindPriceStrategy` through an implementation of the `FindPriceHook` interface. It allows you to modify the default price based on the default price value and its entry. You can use these modifications only when the `isApplicable` method returns `true`. Any customs implementations of the `FindPriceHook` interface need to be defined as Spring beans and registered to the `findPriceHooks` list.

As only the first implementation of an applicable registered price hook is evaluated at run time, it is impossible to implement chains of multiple modifications. As a consequence, price hook implementations can't overlap with regards to their domain of applicability.

Design Concepts

The extensible cart calculation uses the generic API and the processor chain design concepts.

Generic API

The logics responsible for resolving price, tax, and discount values have a shared part (retrieving and matching takes place for prices, taxes, and discounts). It allows the extensible cart calculation to use a generic interface - `FindPDTValueInfoStrategy`. The interface provides two unified methods:

- `List<Value> getPDTValues(Criteria criteria)` resolves price/discount/tax **values**
- `List<INFO> getPDTInformation(Criteria criteria)` resolves price/discount/tax **information**

The interface allows the generic implementation to serve the shared process flow for resolving prices, discounts, and taxes. It splits the task responsible for resolving the values and information into multiple parts, and executes the methods from the dedicated single-method interfaces.

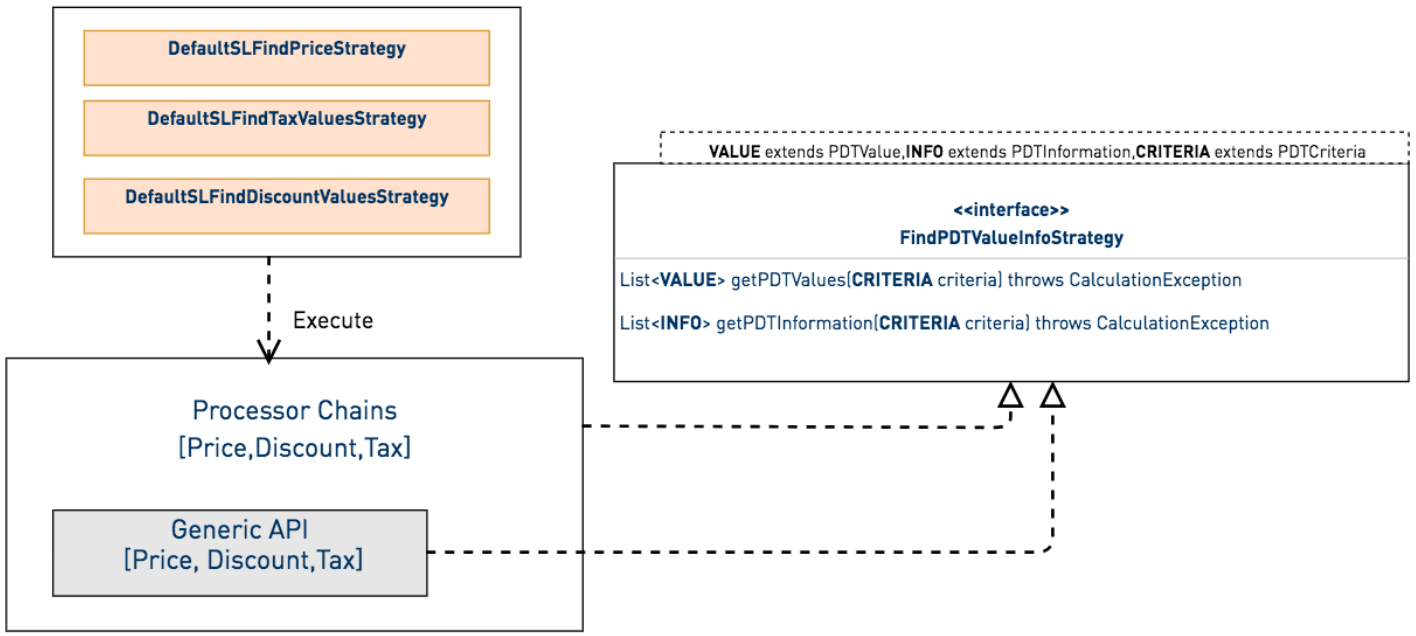
For more information, see Generic API.

Processor Chains

You can create a custom price or/and tax or/and discount processor in which you can decide when your logic should be applied depending on a passed context.

For more information, see Processor Chains.

The diagram shows the connection between the two concepts:



You can see that the default service layer implementations of price, discount, and tax strategies are where the execution of resolving prices, discounts, and taxes begins. The strategies are:

- `DefaultSLFindPriceStrategy`
- `DefaultSLFindDiscountValuesStrategy`
- `DefaultSLFindTaxValuesStrategy`

Those strategies call the `DefaultPDTProcessorChainExecutor` class that implements the `FindPDTValueInfoStrategy` interface (**Processor Chains** in the diagram). `DefaultPDTProcessorChainExecutor` executes a configured list of processors - classes that implement the `PDTProcessor` interface. It also creates a context from a passed criteria, and calls `doProcess()` on the first processor from the processor chain list.

Every list of processors is already configured by default with one default processor. You can extend it with custom processors. In appropriate conditions, the default `DefaultPDTProcessor` executes the default implementation of `FindPDTValueInfoStrategy` (**Generic API** in the diagram).

Below are snippets of a Spring configuration (`order-spring.xml` defined in Platform) for the default implementation of the service layer discount strategy (`DefaultSLFindDiscountValuesStrategy`). The configurations for price and tax strategies look similar.

Code sample corresponding to `DefaultSLFindDiscountValuesStrategy` from the diagram:

```
<alias alias="slFindDiscountValuesStrategy" name="defaultSLFindDiscountValuesStrategy" />
    <bean id="defaultSLFindDiscountValuesStrategy"
            class="de.hybris.platform.order.strategies.calculation.impl.servicelayer.DefaultSLFindD
        <property name="findDiscountValueInfoStrategy" ref="findDiscountValueInfoStrategyExecutable
        <property name="pdtCriteriaFactory" ref="PDTCriteriaFactory"/>
    </bean>
```

Code sample corresponding to Processor Chains from the diagram:

```
<alias alias="findDiscountValueInfoStrategyExecutable" name="defaultDiscountProcessorChainExecutor
    <bean id="defaultDiscountProcessorChainExecuctor" class="de.hybris.platform.order.strategies.ca
        <property name="pdtProcessors" ref="discountProcessors" />
    </bean>

    <alias alias="discountProcessors" name="defaultDiscountProcessors" />
    <util:list id="defaultDiscountProcessors" value-type="de.hybris.platform.order.strategies.calcu
```

```
    <bean id="discountProcessorsMergeDirective" depends-on="defaultDiscountProcessors" parent="list
       <property name="add" ref="findDiscountValuesStrategyProcessor" />
    </bean>

    <alias alias="findDiscountValuesStrategyProcessor" name="defaultFindDiscountValuesStrategyProces
     <bean id="defaultFindDiscountValuesStrategyProcessor"
             class="de.hybris.platform.order.strategies.calculation.pdt.processor.impl.DefaultPDTPro
         <property name="findPDTValueInfoStrategy" ref="findDiscountValueInfoStrategy"/>
     </bean>
```

Code sample corresponding to Generic API from the diagram:

```
<alias alias="findDiscountValueInfoStrategy" name="defaultFindDiscountValueInfoStrategy"/>
    <bean id="defaultFindDiscountValueInfoStrategy" class="de.hybris.platform.order.strategies.calc
        ...
    </bean>
```

Generic API

See the details of the extensible cart calculation generic API.

The generic API of the extensible cart calculation is based on an interface, and a class that implements it. They are:

- the `FindPDTValueInfoStrategy<VALUE extends PDTValue, INFO extends PDTInformation, CRITERIA extends PDTCriteria>` interface
- the `GenericPDTFindValueInfoStrategy<VALUE extends PDTValue, INFO extends PDTInformation, CRITERIA extends PDTCriteria, MODEL extends PDTRowModel>` class

The `FindPDTValueInfoStrategy` interface provides two generic methods that are used to resolve prices, taxes and discounts:

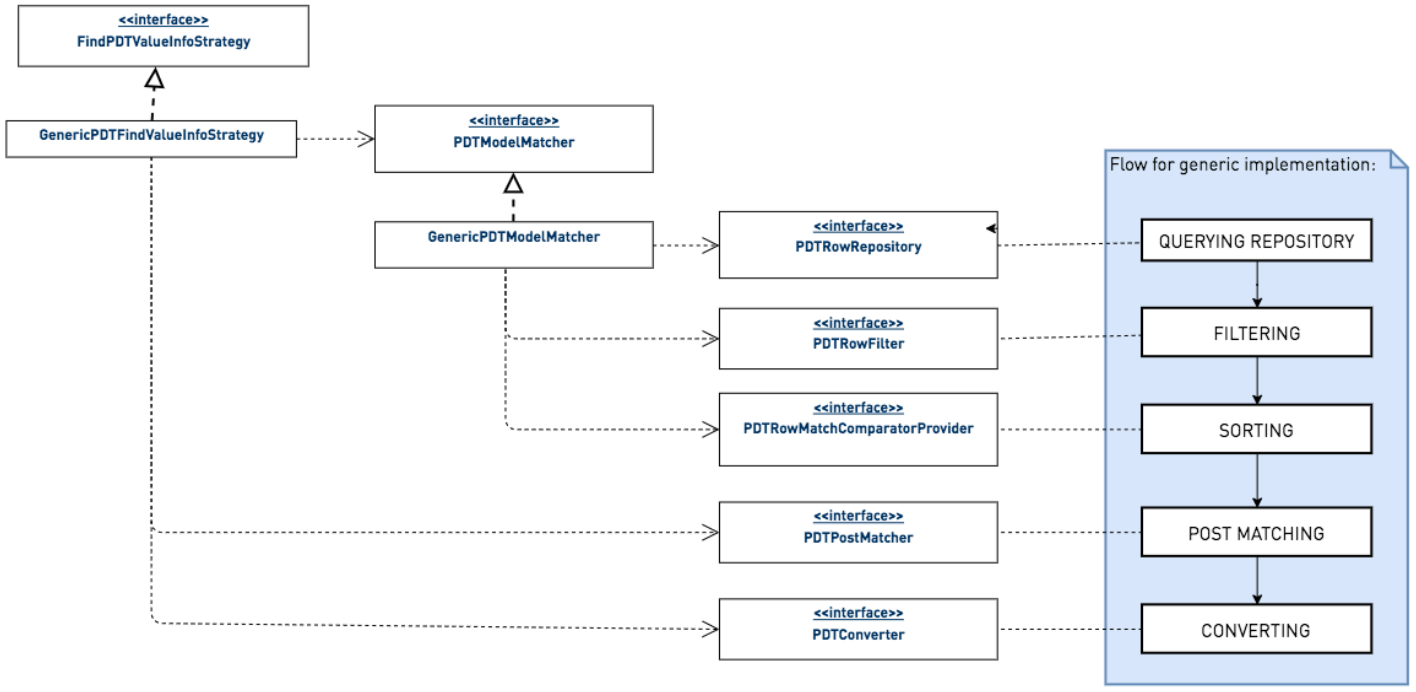
- `List<VALUE> getPDTValues(CRITERIA)` resolves price, discount, and tax **values** for given criteria. The method is used during cart (re)calculation.
- `List<INFO> getPDTInformation(CRITERIA)` resolves price, discount, and tax **information** for given criteria.

The generic `GenericPDTFindValueInfoStrategy` implementation is based on the `FindPDTValueInfoStrategy` interface. The implementation executes the shared flow of retrieving and matching prices, discounts, and taxes to resolve values and information. It splits the task responsible for resolving values and information into multiple parts, and executes methods from the dedicated single-method interfaces in the following order:

Order		Method	Interface	Method Description
1		`List<MODEL> matchRows(final CRITERIA criteria)`	`PDTModelMatcher<CRITERIA extends PDTCriteria, MODEL extends PDTRowModel>` (*)	Matches rows (including querying repository, filtering rows, and sorting rows)
	1.1	`Collection<MODEL> findRows(CRITERIA criteria)`	`PDTRowRepository<CRITERIA extends PDTCriteria, MODEL extends PDTRowModel>`	Queries repository
	1.2	`Collection<MODEL> filter(Collection<MODEL> collection, CRITERIA criteria)`	`PDTRowFilter<CRITERIA extends PDTCriteria, MODEL extends PDTRowModel>`	Filters rows

Order		Method	Interface	Method Description
	1.3	`Comparator<MODEL> comparator(CRITERIA criteria)PDTRowFilter<CRITERIA extends PDTCriteria, MODEL extends PDTRowModel>`	`PDTRowMatchComparatorProvider<CRITERIA extends PDTCriteria, MODEL extends PDTRowModel>`	Compares rows
2		`Collection<MODEL> process(Collection<MODEL> models, CRITERIA criteria)`	`PDTPostMatcher<CRITERIA extends PDTCriteria, MODEL extends PDTRowModel>`	Posts matching rows
3		`TARGET convert(SOURCE source, CONTEXT context)`	`PDTConverter<SOURCE, TARGET, CONTEXT>`	Converts rows into value/information

(*) For the `PDTModelMatcher` interface, the generic `GenericPDTModelMatcher` implementation is also provided. The implemented `matchRows()` method finds rows using `PDTRowRepository`, filters them using `PDTRowFilter`, and sorts them using `PDTRowMatchComparatorProvider`.



The following code snippet shows the implementation of the `getPDTValues()` method of `GenericPDTFindValueInfoStrategy`:

```
private PDTModelMatcher<CRITERIA, MODEL> valueModelMatcher;
private PDTPostMatcher<CRITERIA, MODEL> valuePostMatcher;
private PDTConverter<MODEL, VALUE, CRITERIA> valueConverter;


public List<VALUE> getPDTValues(final CRITERIA criteria) throws CalculationException{
    Collection<MODEL> rowModels = valueModelMatcher.matchRows(criteria);
    rowModels = valuePostMatcher.process(rowModels, criteria);
    return valueConverter.convertAll(rowModels, criteria);
}
```

The following code snippet shows the implementation of the `getPDTInformation()` method:

```
private PDTModelMatcher<CRITERIA, MODEL> informationModelMatcher;
private PDTPostMatcher<CRITERIA, MODEL> informationPostMatcher;
private PDTConverter<MODEL, INFO, CRITERIA> informationConverter;

public List<INFO> getPDTInformation(final CRITERIA criteria) throws CalculationException{
    Collection<MODEL> rowModels = informationModelMatcher.matchRows(criteria);
    rowModels = informationPostMatcher.process(rowModels, criteria);
    return informationConverter.convertAll(rowModels, criteria);
}
```

The `GenericPDTFindValueInfoStrategy` class is a base class for default implementations responsible for resolving price/tax/discounts values and information in the `de.hybris.platform.order.strategies.calculation.pdt.impl` package as follows:

- `DefaultFindPriceValueInfoStrategy` extends `GenericPDTFindValueInfoStrategy<PriceValue, PriceInformation, PriceValueInfoCriteria, PriceRowModel>`
- `DefaultFindDiscountValueInfoStrategy` extends `GenericPDTFindValueInfoStrategy<DiscountValue, DiscountInformation, DiscountValueInfoCriteria, DiscountRowModel>`
- `DefaultFindTaxValueInfoStrategy` extends `GenericPDTFindValueInfoStrategy<TaxValue, TaxInformation, TaxValueInfoCriteria, TaxRowModel>`

The following XML configuration snippet shows part of a Spring configuration for the default `DefaultFindDiscountValueInfoStrategy` discount strategy (for `DefaultFindPriceValueInfoStrategy`, the `DefaultFindTaxValueInfoStrategy` XML configuration looks similar):

```
<alias alias="discountValueModelMatcher" name="defaultDiscountValueModelMatcher"/>

<bean id="defaultDiscountValueModelMatcher" class="de.hybris.platform.order.strategies.calculation.
    <property name="rowRepository" ref="discountRowValueRepository"/>
    <property name="rowFilter" ref="discountRowValueFilter"/>
    <property name="rowMatchComparatorProvider" ref="discountRowValueMatchComparatorProvider"/>
</bean>

<alias alias="discountInfoModelMatcher" name="defaultDiscountInfoModelMatcher"/>
<bean id="defaultDiscountInfoModelMatcher" class="de.hybris.platform.order.strategies.calculation.p
    <property name="rowRepository" ref="discountRowInfoRepository"/>
    <property name="rowFilter" ref="discountRowInfoFilter"/>
    <property name="rowMatchComparatorProvider" ref="discountRowInfoMatchComparatorProvider"/>
</bean>

<alias alias="findDiscountValueInfoStrategy" name="defaultFindDiscountValueInfoStrategy"/>
<bean id="defaultFindDiscountValueInfoStrategy" class="de.hybris.platform.order.strategies.calculat
    <property name="valueModelMatcher" ref="discountValueModelMatcher"/>
    <property name="informationModelMatcher" ref="discountInfoModelMatcher"/>
    <property name="valuePostMatcher" ref="discountValuePostMatcher"/>
    <property name="informationPostMatcher" ref="discountInfoPostMatcher"/>
    <property name="valueConverter" ref="discountValueConverter"/>
    <property name="informationConverter" ref="discountInfoConverter"/>
</bean>
```

The configuration is used for resolving discount values and information. `DefaultFindDiscountValueInfoStrategy` has these two sets of bean properties:

- The following set is used when the `getPDTInformation()` method is called:

  ```
  <property name="valueModelMatcher" ref="discountValueModelMatcher"/>
  <property name="valuePostMatcher" ref="discountValuePostMatcher"/>
  <property name="valueConverter" ref="discountValueConverter"/>
  ```

- The following set it used when the `getPDTValues()` method is called:

```
    <property name="informationModelMatcher" ref="discountInfoModelMatcher"/>
    <property name="informationPostMatcher" ref="discountInfoPostMatcher"/>
    <property name="informationConverter" ref="discountInfoConverter"/>
```

Processor Chains

Processor chains allow you to use the default implementation for resolving price, discount, and tax values/information without using inheritance. Processors are executed whenever the values or information are needed in a cart calculation based on service layer.

A single processor based on a passed context can decide whether to take part in the process of resolving price, discount, and tax values/information.

`DefaultSLFindPriceStrategy`, `DefaultSLFindDiscountValuesStrategy`, and `DefaultSLFindTaxValuesStrategy` are the service layer implementations of the `FindPriceStrategy`, `FindDiscountValuesStrategy`, `FindTaxValuesStrategy` strategies, respectively. The implementations are the starting point of cart calculation process. Those strategies call `DefaultPDTProcessorChainExecutor` - a special implementation of `FindPDTValueInfoStrategy`. The `DefaultPDTProcessorChainExecutor` class manages the execution of a processor list. A processor is an object of a class that implements the `PDTProcessor` interface and performs processing for a passed `PDTContext`. `PDTContext` contains criteria (as a request) created based on `AbstractOrderModel` or `AbstractOrderEntryModel`, and values/information that may be completed by a processor (as a response). Each of the mentioned strategies calls its own executor with a dedicated processor list.

The code snippet is a Spring configuration in `order-spring.xml` for `DefaultSLFindPriceStrategy`, which uses a processor list called `priceProcessors`:

```
<alias alias="slFindPriceStrategy" name="defaultSLFindPriceStrategy" />
<bean id="defaultSLFindPriceStrategy" class="de.hybris.platform.order.strategies.calculation.impl.s
    <property name="findPriceValueInfoStrategy" ref="findPriceValueInfoStrategyExecutable"/>
    <property name="pdtCriteriaFactory" ref="PDTCriteriaFactory"/>
</bean>
<alias alias="findPriceValueInfoStrategyExecutable" name="defaultPriceProcessorChainExecuctor" />
<bean id="defaultPriceProcessorChainExecuctor" class="de.hybris.platform.order.strategies.calculati
    <property name="pdtProcessors" ref="priceProcessors" />
</bean>
```

The following is a configuration for `DefaultSLFindDiscountValuesStrategy`, which uses a processor list named `discountProcessors`:

```
<alias alias="slFindDiscountValuesStrategy" name="defaultSLFindDiscountValuesStrategy" />
<bean id="defaultSLFindDiscountValuesStrategy" class="de.hybris.platform.order.strategies.calculati
    <property name="findDiscountValueInfoStrategy" ref="findDiscountValueInfoStrategyExecutable"/>
    <property name="pdtCriteriaFactory" ref="PDTCriteriaFactory"/>
</bean>

<alias alias="findDiscountValueInfoStrategyExecutable" name="defaultDiscountProcessorChainExecutor
<bean id="defaultDiscountProcessorChainExecuctor" class="de.hybris.platform.order.strategies.calcul
    <property name="pdtProcessors" ref="discountProcessors" />
</bean>
```

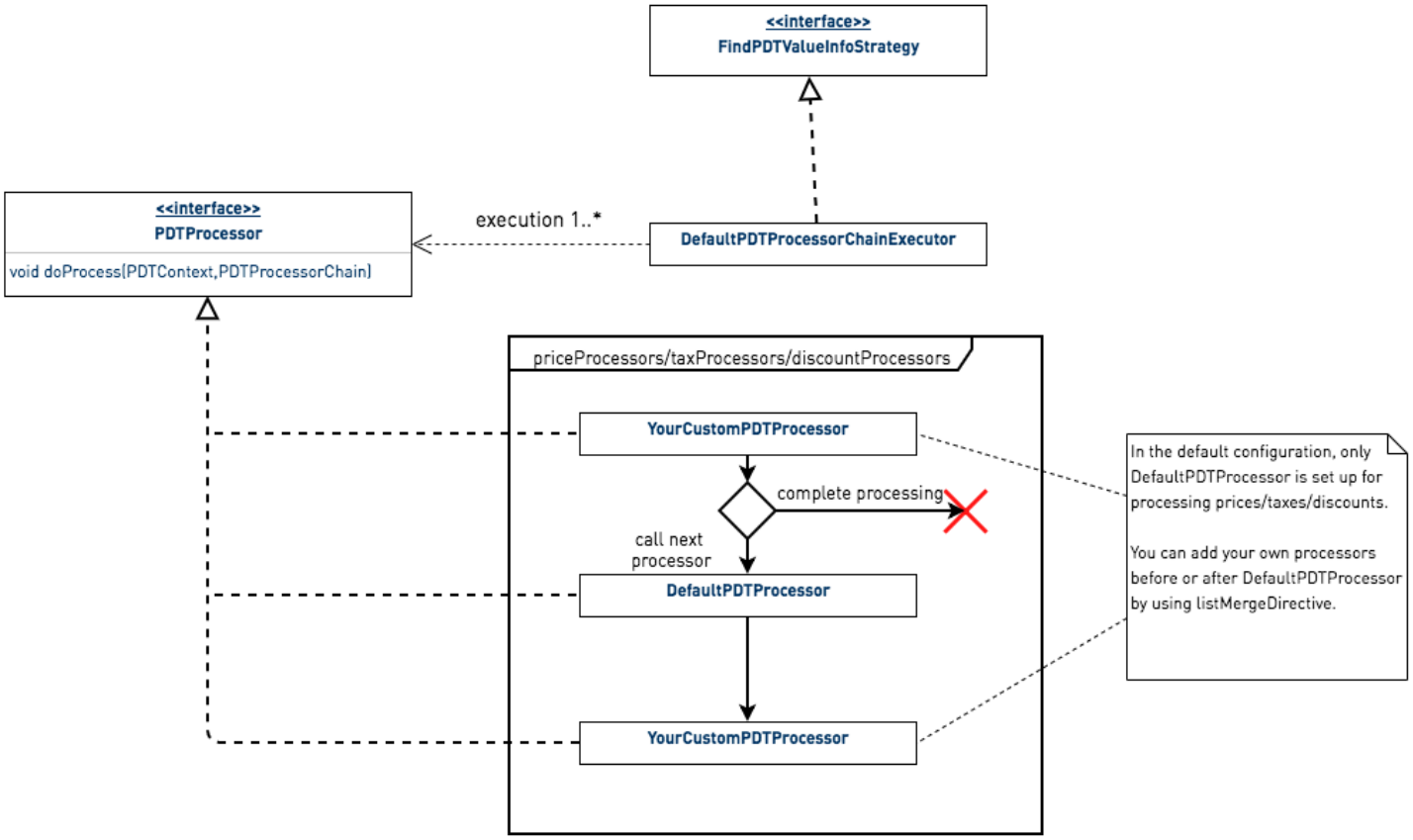
The following is a configuration for `DefaultSLFindTaxValuesStrategy`, which uses a processor list named `taxProcessors`:

```
<alias alias="slFindTaxValuesStrategy" name="defaultSLFindTaxValuesStrategy" />
<bean id="defaultSLFindTaxValuesStrategy" class="de.hybris.platform.order.strategies.calculation.in
    <property name="findTaxValueInfoStrategy" ref="findTaxValueInfoStrategyExecutable"/>
    <property name="pdtCriteriaFactory" ref="PDTCriteriaFactory"/>
```

```
</bean>

<alias alias="findTaxValueInfoStrategyExecutable" name="defaultTaxProcessorChainExecuctor" />
<bean id="defaultTaxProcessorChainExecuctor" class="de.hybris.platform.order.strategies.calculatior
    <property name="pdtProcessors" ref="taxProcessors" />
</bean>
```

`DefaultPDTProcessorChainExecutor` executes configured processors in a provided order. An executed processor may set values/information in the context and decide whether to invoke the next processor in the chain by calling the `doProcess()` method from `PDTProcessorChain`, or complete processing by not calling this method. For every list of processors, one `DefaultPDTProcessor` processor is provided by default. For more information about `DefaultPDTProcessor`, see section below.

The diagram depicts the chain configuration mechanism:



The XML snippet shows the `order-spring.xml` configuration for a list of processors responsible for handling `discountProcessors` discounts:

```
<alias alias="slFindDiscountValuesStrategy" name="defaultSLFindDiscountValuesStrategy" />
<bean id="defaultSLFindDiscountValuesStrategy"
        class="de.hybris.platform.order.strategies.calculation.impl.servicelayer.DefaultSLFindDisco
    <property name="findDiscountValueInfoStrategy" ref="findDiscountValueInfoStrategyExecutable"/>
    <property name="pdtCriteriaFactory" ref="PDTCriteriaFactory"/>
</bean>

<alias alias="findDiscountValueInfoStrategyExecutable" name="defaultDiscountProcessorChainExecutor
<bean id="defaultDiscountProcessorChainExecuctor" class="de.hybris.platform.order.strategies.calcul
    <property name="pdtProcessors" ref="discountProcessors" />
</bean>

<alias alias="discountProcessors" name="defaultDiscountProcessors" />
<util:list id="defaultDiscountProcessors" value-type="de.hybris.platform.order.strategies.calculati

<bean id="discountProcessorsMergeDirective" depends-on="defaultDiscountProcessors" parent="listMerg
    <property name="add" ref="findDiscountValuesStrategyProcessor" />
</bean>
```

```
<alias alias="findDiscountValuesStrategyProcessor" name="defaultFindDiscountValuesStrategyProcesso
<bean id="defaultFindDiscountValuesStrategyProcessor"
        class="de.hybris.platform.order.strategies.calculation.pdt.processor.impl.DefaultPDTProces
    <property name="findPDTValueInfoStrategy" ref="findDiscountValueInfoStrategy"/>
</bean>
```

For `priceProcessors` and `taxProcessors`, the configurations look similar.

Extending the Processor Chains

`PDTProcessor` is an interface implemented by processor components executed by `DefaultPDTProcessorChainExecutor`. To affect the price/discount/tax values/information resolving mechanism, create a custom implementation of the `PDTProcessor` interface.

Your implementation of the `doProcess(PDTContext context,PDTProcessorChain chain)` method from `PDTProcessor` could follow this pattern:

1. Examine `PDTCriteriaTarget` from the context (`context.getPDTCriteriaTarget()`)
    - if `PDTCriteriaTarget.VALUE` is set, then process for resolving **values** takes place
    - if `PDTCriteriaTarget.INFO` is set, then process for resolving **information** takes place
2. Examine the `CRITERIA` (criteria is based on `AbstractOrderModel` or `AbstractOrderEntryModel`) from the context (`context.getCriteria()`)
3. Based on `PDTCriteriaTarget` and `CRITERIA`, decide whether to resolve the specified values or information, or not
4. If you resolved values or information in the previous step, set up those values or information in the context (`context.setInformations()` / `context.setValues()`)
5. Based on the previous step, decide whether you want to invoke the next processor in the chain. To do it, use the `PDTProcessorChain` object (by calling `chain.doProcess(PDTContext context)`). If you don't invoke the next processor, you will complete the process.

When the process is complete, the `DefaultPDTProcessorChainExecutor` returns the values or the information set in the context (`context.getValues()` / `context.getInformations()`).

You have to add custom processor to a proper processor list: price processors / discount processors / tax processors, depending on whether the processor is dedicated to handling price / discount / tax values and information.

The bean with the `defaultPriceProcessors` id and the `priceProcessors` alias is defined in `order-spring.xml` as follows:

```
<alias alias="priceProcessors" name="defaultPriceProcessors" />
<util:list id="defaultPriceProcessors" value-type="de.hybris.platform.order.strategies.calculation.
```

The bean with the `defaultDiscountProcessors` id and the `discountProcessors` alias is defined in `order-spring.xml` as follows:

```
<alias alias="discountProcessors" name="defaultDiscountProcessors" />
<util:list id="defaultDiscountProcessors" value-type="de.hybris.platform.order.strategies.calculati
```

The bean with the `defaultTaxProcessors` id and the `taxProcessors` alias is defined in `order-spring.xml` as follows:

```
<alias alias="taxProcessors" name="defaultTaxProcessors" />
<util:list id="defaultTaxProcessors" value-type="de.hybris.platform.order.strategies.calculation.pd
```

You can use `ListMergeDirective` to add processors to `defaultPriceProcessors` / `defaultDiscountProcessors` / `defaultTaxProcessors`, and control their order if necessary.

For example, the following scenario assumes that you want to use the provided default implementation after executing your custom processor (your processor doesn't handle all cases and they need to be handled by the basic processor). The configuration adds the `customPriceProcessor` bean to the `defaultPriceProcessors` list before the default `findPriceValuesStrategyProcessor` processor bean:

```
<bean id="customPriceProcessorMergeDirective" depends-on="defaultPriceProcessors" parent="listMerge
    <property name="add" ref="customPriceProcessor" />
    <property name="beforeBeanNames">
      <list>
         <value>findPriceValuesStrategyProcessor</value>
      </list>
    </property>
</bean>
<bean id="customPriceProcessor" class="your.package.name.YourCustomPriceProcessor"/>
```

If you don't want to have the default implementation in the list of processors, you can override the Spring configuration as follows:

```
<alias alias="priceProcessors" name="customPriceProcessors" />
<util:list id="customPriceProcessors" value-type="de.hybris.platform.order.strategies.calculation.p


<bean id="customPriceProcessorMergeDirective" depends-on="customPriceProcessors" parent="listMergeD
    <property name="add" ref="customPriceProcessor"/>
</bean>
<bean id="customPriceProcessor" class="your.package.name.YourCustomPriceProcessor"/>
```

Those examples also apply to the tax and discount processors.

DefaultPDTProcessor

Platform provides the default `DefaultPDTProcessor` processor implementation. This implementation is added to each of the default processor chains (`defaultPriceProcessors` , `defaultDiscountProcessors`, `defaultTaxProcessors`). The processor calls the default implementations of `FindPDTValueInfoStrategy` for prices, discounts, and taxes. The default processor sets values/information when the current values/information forwarded in `PDTContext` are/is null or set to a `DefaultPDTProcessor.NO_RESULT` value. The processor always calls the `doProcess()` method from `PDTProcessorChain` (the next processor configured in the chain is always executed if it exists).

Below is a code snippet from the `DefaultPDTProcessor` class:

```
public void doProcess(final PDTContext context, final PDTProcessorChain chain) throws CalculationEx
        Objects.requireNonNull(context, "context is required");
        Objects.requireNonNull(context.getCriteria(), "context.getCriteria() is required");
        Objects.requireNonNull(context.getPDTCriteriaTarget(), "context.getPDTCriteriaTarget() is r
        final CRITERIA criteria = (CRITERIA) context.getCriteria();

        //it get values only if response object is empty
        if (PDTCriteriaTarget.VALUE.equals(context.getPDTCriteriaTarget()) && valuesNotSet(context)
        {
            final List<VALUE> actualResponse = findPDTValueInfoStrategy.getPDTValues(criteria);
            context.setValues(actualResponse);
        }
        else if (PDTCriteriaTarget.INFORMATION.equals(context.getPDTCriteriaTarget()) && informatic
        {
            final List<INFO> actualResponse = findPDTValueInfoStrategy.getPDTInformation(criteria);
            context.setInformations(actualResponse);
        }
        //always call doProcess from PDTProcessorChain
```

```
        chain.doProcess(context);
}
```

The configuration for `DefaultPDTProcessor` responsible for handling prices is added to `defaultPriceProcessors` that uses the `findPriceValueInfoStrategy` bean. The configuration is defined in `order-spring.xml` as follows:

```
<bean id="priceProcessorsMergeDirective" depends-on="defaultPriceProcessors" parent="listMergeDirec
    <property name="add" ref="findPriceValuesStrategyProcessor" />
</bean>

<alias alias="findPriceValuesStrategyProcessor" name="defaultFindPriceValuesStrategyProcessor" />
<bean id="defaultFindPriceValuesStrategyProcessor" class="de.hybris.platform.order.strategies.calcu
        <property name="findPDTValueInfoStrategy" ref="findPriceValueInfoStrategy"/>
</bean>
```

The configuration for `DefaultPDTProcessor` responsible for handling discounts is added to `defaultDiscountProcessors` that uses the `findDiscountValueInfoStrategy` bean. The configuration is defined in `order-spring.xml` as follows:

```
<bean id="discountProcessorsMergeDirective" depends-on="defaultDiscountProcessors" parent="listMerg
     <property name="add" ref="findDiscountValuesStrategyProcessor" />
</bean>

<alias alias="findDiscountValuesStrategyProcessor" name="defaultFindDiscountValuesStrategyProcessor
<bean id="defaultFindDiscountValuesStrategyProcessor" class="de.hybris.platform.order.strategies.ca
    <property name="findPDTValueInfoStrategy" ref="findDiscountValueInfoStrategy"/>
</bean>
```

The configuration for `DefaultPDTProcessor` responsible for handling taxes is added to `defaultTaxProcessors` that uses the `findTaxValueInfoStrategy` bean. The configuration is defined in `order-spring.xml` as follows:

```
<bean id="taxProcessorsMergeDirective"  depends-on="defaultTaxProcessors" parent="listMergeDirectiv
    <property name="add" ref="findTaxValueInfoStrategyProcessor" />
</bean>
<alias alias="findTaxValueInfoStrategyProcessor" name="defaultFindTaxValueInfoStrategyProcessor" />
<bean id="defaultFindTaxValueInfoStrategyProcessor" class="de.hybris.platform.order.strategies.calc
    <property name="findPDTValueInfoStrategy" ref="findTaxValueInfoStrategy"/>
</bean>
```

Price, Discount, and Tax Services

See the list of Platform Services extended with additional methods to show all prices, taxes, and discounts for given product set in criteria.

ℹ Note

The default implementations of `PriceService(DefaultPriceService)`, `TaxService(DefaultTaxService)`, `DiscountService(DefaultDiscountService)` rely on `FindPriceStrategy`, `FindTaxValuesStrategy`, and `FindDiscountValuesStrategy`, respectively.

The same strategies are used by `DefaultCalculationService`.

`PriceService` includes:

- `List<PriceInformation> getPriceInformations(final PriceCriteria priceCriteria)`
- `ProductPriceInformations getAllPriceInformation(final PriceCriteria priceCriteria)`

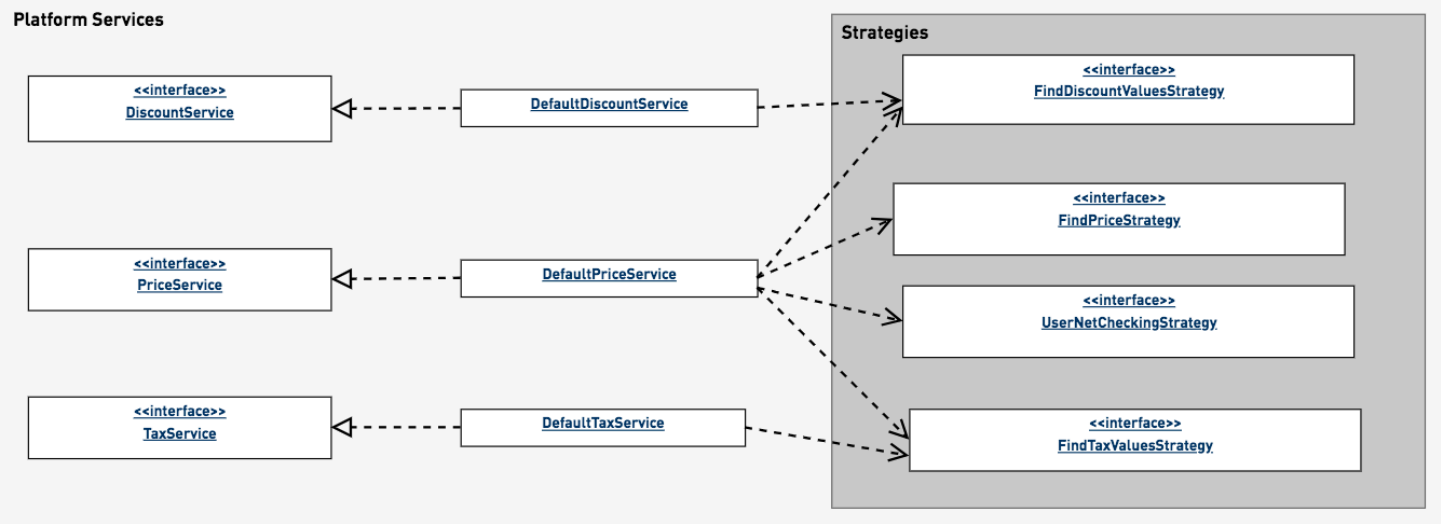
`TaxService` includes:

- `List<TaxInformation> getTaxInformation(final BaseCriteria baseCriteria)`

`DiscountService` includes:

- `List<DiscountInformation> getDiscountInformation(final BaseCriteria baseCriteria)`

The diagram shows how `[Price/Tax/Discount]Services` rely on `Find[Price,Discount,Tax]...Strategies` when the service evaluates the method for retrieving information:



Operation Modes

The extensible cart calculation API is enabled by default, in the SMART mode. This mode uses the service layer but under certain circumstances it falls back to jalo.

The default operation mode setting is `pdt.calculation.mode=SMART`.

You can activate a chosen mode using the following options:

Setting	Explanation
`pdt.calculation.mode=SL`	Enforces the use of the service layer cart calculation.
`pdt.calculation.mode=JALO`	Enforces the use of `Europe1PriceFactory` and jalo for processing cart calculation logic (deprecated).
`pdt.calculation.mode=SMART`	Enforces the use of the service layer cart calculation but falls back to jalo if a custom `currentFactoryFindPricingStrategy` bean is defined or a custom `priceFactory` is set in the JaloSession.

Migration Guide

See the information related to migrating your cart calculation logic from Jalo to the extensible cart calculation.

ⓘ Note

You can only either enable the service layer solution to operate in full, or not at all.

The service layer cart calculation is enabled by default in the SMART mode but this mode shouldn't be used for production environment. You should decide between the JALO and SL modes based on the information in this topic.

SMART Mode

During startup, you are informed in a log that this mode is enabled. This mode uses service layer but under certain circumstances it falls back to jalo. The information about falling back to jalo is logged at the INFO level to the console. Fallbacks to jalo take place in the following situations (be aware that not all customizations are recognized by the smart mode):

Customization Type	Customization Detection	Fallback to Jalo
Bean definition for `currentFactoryFindPricingStrategy` (defined in `order-spring.xml` in the Platform core) is overridden	After application startup	Permanent
Custom price factory is configured in the application	After application startup	Permanent
Custom price factory is set in the JaloSession	In runtime	Temporary

The SMART mode is more intended to detect scenarios where the code related to price API is customized using jalo.

JALO Mode

This mode doesn't require any adjustment. It doesn't bring any changes in the price/calculation API. Platform and extensions work with the calculation based on the implementation of the `PriceFactory` interface.

SL Mode

This mode enforces the use of the service layer calculation. If you use a customization code, you should adjust your code to this approach.

Below you can find scenarios of how you can migrate the JALO customization into the service layer solution. Be aware that in some cases it is not always possible to transition into the service layer solution.

The scenarios are applicable if you use `DefaultCalculationService` (the default Platform implementation for cart calculation):

Jalo Customization Scenario
Jalo `PaymentMode/StandardPaymentMode` item is extended and the `PriceValue getCost(AbstractOrder)` method is overridden.
Jalo `DeliveryMode/ZoneDeliveryMode` item is extended and the `PriceValue getCost(AbstractOrder)` method is overridden.

Jalo Customization Scenario
Jalo `Discount` item is extended and the `DiscountValue getDiscountValue(AbstractOrder)` method is overridden.
The bean definition for `currentFactoryFindPricingStrategy` (defined in `order-spring.xml` in the Platform core) is overridden in SAP Commerce or third-party extensions that you can't change. **→ Tip** If this scenario happens, you will see this message in a log during the application startup: `Detected custom bean 'currentFactoryFindPricingStrategy' implementation: className (not: de.hybris.platform.order.strategies.calculation.impl.FindPricingWithCurrentPriceFactoryStrategy) - falling back to jalo strategy).`
The bean definition for `currentFactoryFindPricingStrategy` (defined in `order-spring.xml` in the Platform core) is overridden in your custom extension. **→ Tip** If this scenario happens, you will see this message in a log during the application startup: `Detected custom bean 'currentFactoryFindPricingStrategy' implementation: className (not: de.hybris.platform.order.strategies.calculation.impl.FindPricingWithCurrentPriceFactoryStrategy) - falling back to jalo strategy).`
You use or extend a price factory other than the default implementation from extensions of SAP Commerce or third-party extensions that you can't change. **→ Tip** If this scenario happens, you will see this message in a log during the application startup: `Customized PriceFactory detected: className (not: ...) - falling back to jalo strategy`
You are implementing the `PriceFactory` interface or extending the default implementation `Europe1PriceFactory` class - the methods related to resolving values/informations for price/discount/tax. **→ Tip** If this scenario happens, you will see this message in a log during the application startup: `Customized PriceFactory detected: className (not: ...) - falling back to jalo strategy.`
You are implementing the PriceFactory interface or extending the default implementation `Europe1PriceFactory` class and the `boolean isNetUser(User user)` method is implemented/overridden. **→ Tip** If this scenario happens, you will see this message in a log during the application startup: `Customized PriceFactory detected: className (not: ...) - falling back to jalo strategy.`

Channel Specific Pricing

With Channel-Specific Pricing, you can set up different pricing models depending on how a user is accessing your store. Here you can find how to configure, use, and extend the SAP Commerce functionality to provide channel-specific prices for products.

About Channels

The concept of a channel is able to identify the different ways an end user can interact with the system. For example, if an end user navigates in a browser using his laptop, the system identifies that it is a laptop and filters the prices for that specific channel. If an end user navigates in a browser using his mobile phone, a different channel is associated with his session, and different prices may apply. This filtering mechanism also uses the concept of default prices. If the request is coming from a mobile browser but the system does not have any prices associated to that particular channel, the default price is returned.

Example of Possible Channel Scenarios

To understand how the price filtering is implemented, consider the following database example:

SKU	Channel	Price
001		$10.00
001	desktop	$15.00
001	mobile	$20.00
002		$5.00
002	mobile	$10.00
003		$35.00
004	desktop	$50.00

The following price is returned for each scenario:

1. The user requests the price for SKU 001 using the desktop channel. The price returned is $15.00.
2. The user requests the price for SKU 001 using the mobile channel. The price returned is $20.00.
3. The user requests the price for SKU 001 using the storefront channel. The price returned is $10.00 since a specific price has not been established for the storefront channel and $10.00 has been set as the default price.
4. The user requests the price for SKU 002 using the desktop channel. The price returned is $5.00 since a specific price has not been established for the desktop channel and $5.00 has been set as the default price.
5. The user requests the price for SKU 002 using the mobile channel. The price returned is $10.00.
6. The user requests the price for SKU 002 using an unspecified channel. The price returned is $5.00.
7. The user requests the price for SKU 003 using an unspecified channel. The price returned is $35.00.
8. The user requests the price for SKU 003 using the mobile channel. The price returned is $35.00 since a specific price has not been established for the mobile channel and $35.00 has been set as the default price.
9. The user requests the price for SKU 004 using the desktop channel. The price returned is $50.00.
10. The user requests the price for SKU 004 using the mobile channel. A price is not returned because a price has not been set for the mobile channel. Also, a default price for an unspecified channel has not been established.
11. The user requests the price for SKU 004 using an unspecified channel. A price is not returned because a default price for an unspecified channel has not been established.

Related Information

europe1 Pricing System Guide

Technical Overview

See the technical overview about channel-specific pricing.

Once the system is initialized, a new dynamic enumeration is created (`PriceRowChannel`). This enumeration is defined in `europe1-items.xml`:

`europe1-items.xml`

```
<enumtypes>
        <enumtype code="PriceRowChannel" autocreate="true" generate="true" dynamic="true">
                <!-- Default Values to support Mobile channel and desktop channel -->
                <value code="desktop"/>
                <value code="mobile"/>
        </enumtype>
</enumtypes>
```

The `europe1` extension uses the values defined in this enumeration to match with the new `PriceRow` channel attribute. If you are using an impex to load data into the system, the channel attribute needs to be consistent with the `PriceRowChannel` enumeration.

→ Tip

If you do not want to use the channel functionality, leave the `PriceRow` channel field blank (null) in the database.

The following impex examples create PriceRows with the channel information directly in the database.

Impex Example Using PriceRow Attribute

```
#
# Macro definitions
#
$catalogVersion=catalogVersion(catalog(id[default='electronicsProductCatalog']),version[default='Or

#
# Some pricerows created using the PriceRow entity
#
INSERT_UPDATE PriceRow;$catalogVersion;unit(code);price;currency(isocode)[unique=true];channel(code
;;pieces;20.00;USD;;107701
;;pieces;12.34;USD;"mobile";107701
;;pieces;15.00;USD;"desktop";107701
```

Impex Example Using the Product Managed Object

```
#
# Macro definitions
#
$catalogVersion=catalogVersion(catalog(id[default='electronicsProductCatalog']),version[default='St
$prices=europe1Prices[translator=de.hybris.platform.europe1.jalo.impex.Europe1PricesTranslator]

#
# Some pricerows created using the Product entity
#
INSERT_UPDATE Product;code[unique=true];$catalogVersion;$prices;
;1934793;electronicsProductCatalog:Online;"1 pieces = 10.00 USD mobile, 1 pieces = 20.00 USD deskto
```

How to Extend the Channel Functionality

See how to extend the channel functionality.

When an application calls the `Europe1PriceFactory` class to retrieve the prices, the channel attribute can be added to the `SessionContext`. The `Europe1PriceFactory` analyzes the `SessionContext` to retrieve the correct channel. This contract is defined by the `RetrieveChannelStrategy` interface. Refer to the following `DefaultRetrieveChannelStrategy` implementation to see how the channel is retrieved based on the Accelerator's `UIExperience` session attribute.

`DefaultRetrieveChannelStrategy.java (europe1 extension)`

```
public PriceRowChannel getChannel(final SessionContext ctx)
{
        PriceRowChannel priceRowChannel = null;
        if (ctx == null || ctx.getAttribute(DETECTED_UI_EXPERIENCE_LEVEL) == null)
        {
                //If the channel is null, the client caller will treat this as default price row.
                return priceRowChannel;
        }
        else
        {
                priceRowChannel = ctx.getAttribute(CHANNEL);
                if (priceRowChannel == null)
                {
                        final EnumerationValue enumUIExpLevel = ctx.getAttribute(DETECTED_UI_EXPERI
                        priceRowChannel = getEnumValueForCode(enumUIExpLevel.getCode().toLowerCase(
                        if (priceRowChannel != null)
                        {
                                ctx.setAttribute(CHANNEL, priceRowChannel);
                        }
                }
        }
        return priceRowChannel;
}
```

To customize this behavior, you have to create your own implementation of the interface `RetrieveChannelStrategy` and declare it as a spring bean with the name `retrieveChannelStrategy`. See the bean definition in the following spring configuration file.

`europe1-spring.xml`

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

        <alias alias="retrieveChannelStrategy" name ="defaultRetrieveChannelStrategy"/>
        <bean id="defaultRetrieveChannelStrategy"
                class="de.hybris.platform.europe1.channel.strategies.impl.DefaultRetrieveChannelStr
                <property name="enumerationService" ref="enumerationService"/>
        </bean>
</beans>
```

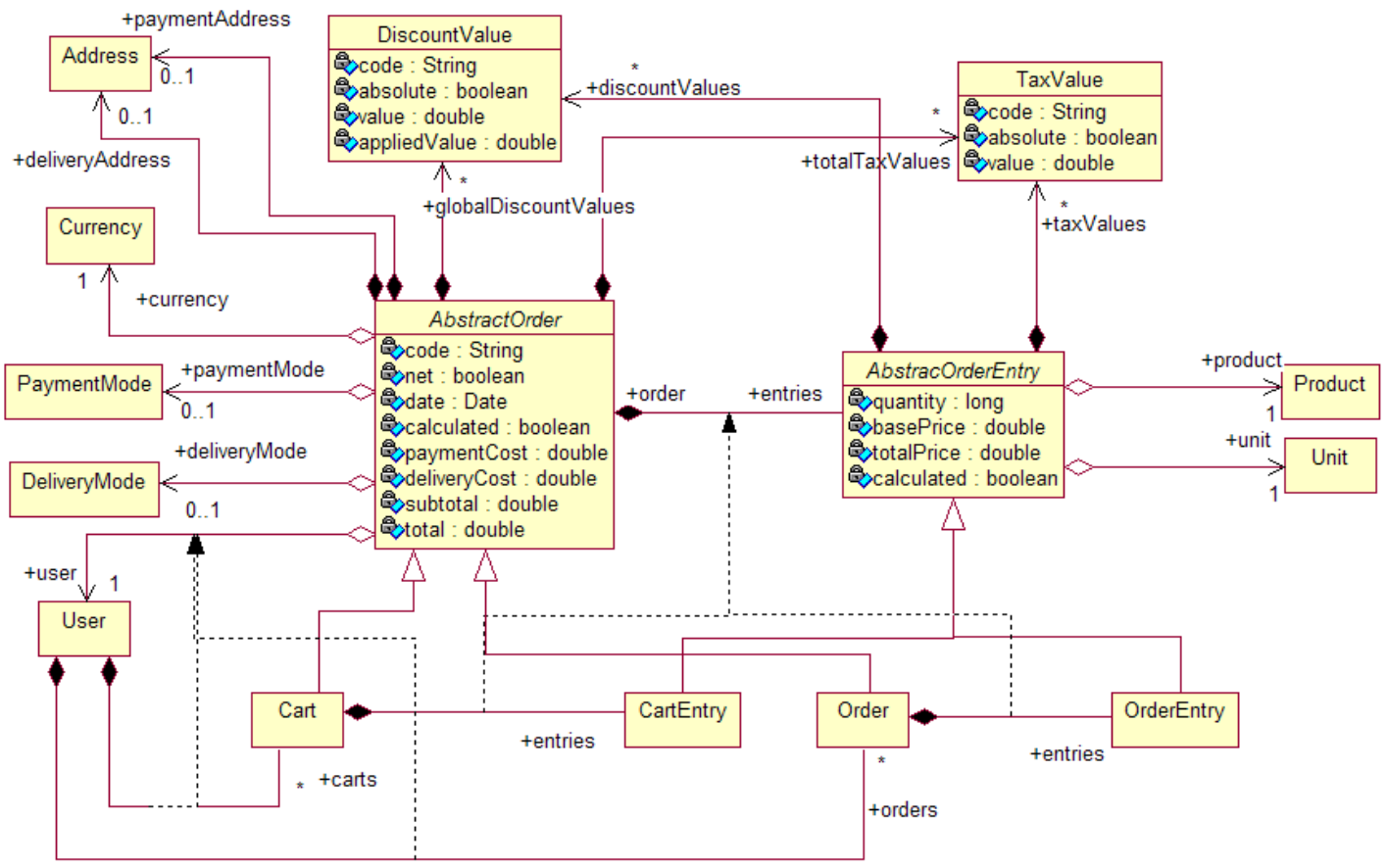
The `Europe1PriceFactory.retrieveChannelStrategy` attribute is going to be injected with the custom implementation putting the new strategy logic in place.

Ordering Process

SAP Commerce has a built-in ordering process that automatically handles calculation of prices, taxes, and discounts for orders. The focus lies on the creation, calculation, recalculation, and lifetime of the carts and orders. Here you will find an overview of the phases of the ordering process and the technical processes in the background.

Type Overview

The SAP Commerce order process by factory default uses the following types:



Basic Concepts of Ordering Process

SAP Commerce comes with an implementation of the ordering process which you can use for application development. The basic process looks like this:

1. Customer visits the webshop application by using any web browser

   Initialization and preparation steps. Transparent to the customer, fully automatic and no customer interaction required.

2. Shopping phase
   - Customer adds products to the virtual shopping cart. Each adding of a product creates a cart entry that holds the product, the quantity, and the unit
   - Cart is calculated to remain up-to-date
3. Customer logs in

   Prior to the login, the customer cannot be identified and an actual order cannot be triggered.

4. Customer provides ordering data (check-out process), such as:
   - delivery address
   - delivery mode
   - payment mode
5. Customer triggers order

The factory default ordering process uses **Carts** and **Orders**:

- **Cart**: Temporary and possibly volatile object, represents a potential order. A Cart is volatile in the sense that it is discarded when the user session times out.

- **Order**: Persistent, represents an actual order placed by a customer.

Both Carts and Orders can have entries, that represent a number of products being selected. A **Cart** with no entries indicates that the customer did not put anything into the cart. A **CartEntry** represents a number of products a customer has put into the cart and is interested in buying. In theory and technically, an **Order** can have no entries at all - but logically this would make no sense as the customer would not have ordered anything. In addition, both Orders and Carts have: **payment mode** and **delivery mode**.

There are two basic approaches to create an order:

- **By using a Cart or an Order as a template**

  When an order is placed, then the **Cart** or **Order** object is copied into an **Order** object; the **Order** is not the same object as the **Cart** or original **Order**, but a copy. In other words: Cart and Order are different objects in SAP Commerce, even if the Order is based on the Cart. The Cart's entries are copied as well.

- **By creating an Order outright**

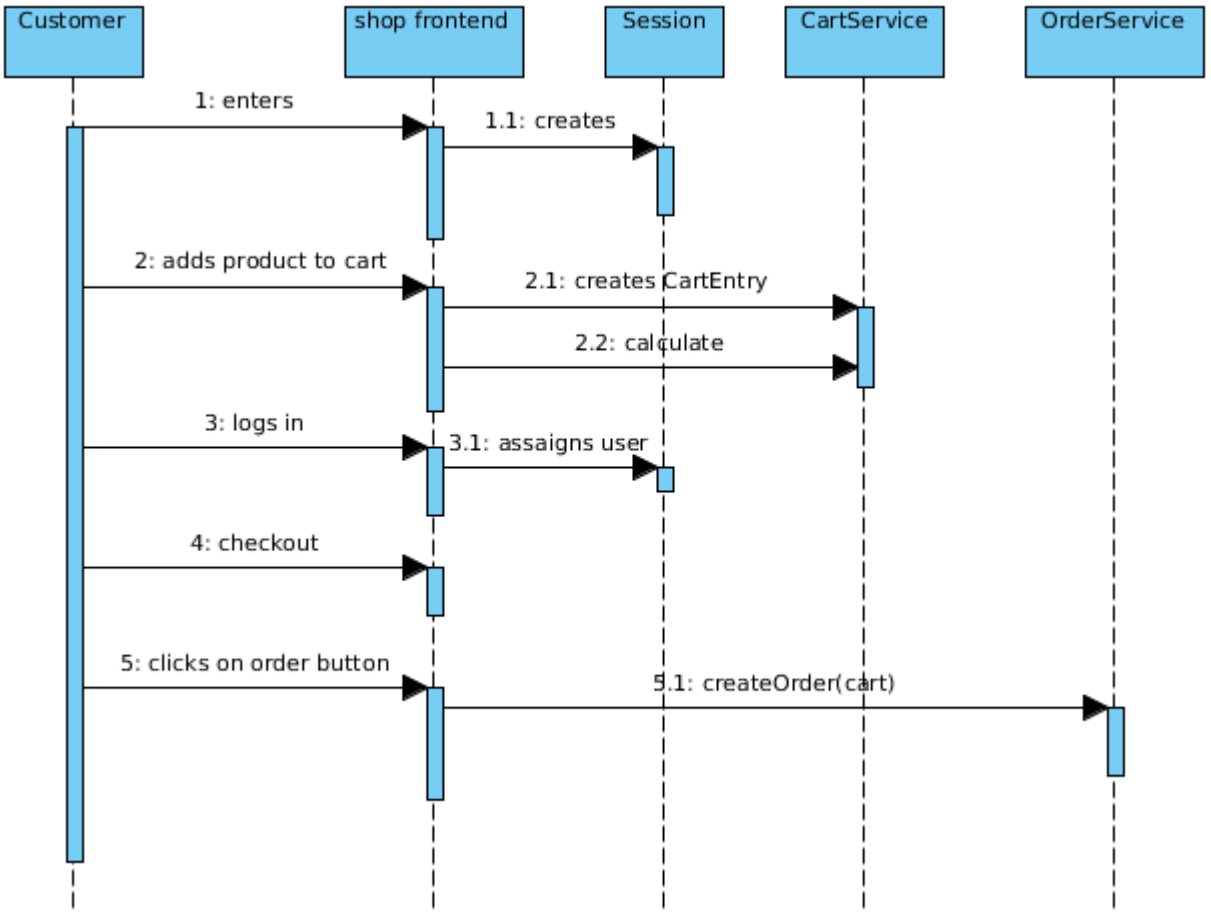
  This simply creates a new Order object with the given identifier and which is assigned to the customer. CartEntries are not copied.

Technical Discussion of the Ordering Process

The following diagram gives an overview on the customer actions during the ordering process and the technical reactions caused:

- **Phase 1: Customer enters the webshop front-end**

  When a customer enters the webshop, the customer's browser opens an HTTP session. SAP Commerce automatically assigns a Session to the HTTP session (**1.1**), and reserves a Cart for the Session (**1.1.1**). The Cart will only be actually created when it is used. The Session is set to the **anonymous** user by default.



- **Phase 2: Customer adds products to the cart/removes products from the cart**

The actual shopping phase contains two processes:

- **2.1: Modifying the Cart's content**

  The customer will modify the contents of the cart by adding products and removing products. SAP Commerce provides two methods to facilitate adding and removing entries from a Cart, respectively:

  - **create new entry and add it to entry list**

    Allows adding a given number of products to the cart. Adding entries causes the cart to be marked as not calculated. See Adding Entries to a Cart/Order section for details.

  - **remove entry from entry list**

    Allows removing an entry from the cart. Removing entries causes the cart to be marked as not calculated. See Removing Entries from a Cart/Order section for details.

- **2.2: Calculation of the cart by using calculation service**

  This must be triggered by the webshop application. Calculation is not run automatically. For more details, see Calculated Attribute and Calculation of Carts/Orders section.

- **Phase 3: Log in**

  The default user account for a Session is **anonymous**. Therefore, on a large system, several thousands of sessions might be open for the **anonymous** user. In order to individually identify a single customer, that customer will have to log in using their individual credentials (user id plus password, for example). During log in, the user account used by the session and the cart is set to the specified user account.

  The login is not only a security issue, but also an identification issue. Without an individual user account, the cart and later on, the order cannot be bound to the individual user, but remains related to **anonymous**.

- **Phase 4: Check-out** Even if a customer has successfully logged in using an individual account, some bits of information required for ordering are still missing, such as: delivery address, delivery mode, payment address, payment mode.

  Additionally, a customer may want to redeem a number of vouchers. Specifying the values for these parameters is referred to as check-out. You may need to implement a front-end page to request these bits of information from the customer.

- **Phase 5: Placing the order**

  Causes SAP Commerce to create an order from the Session's Cart. For details, see Creating Order section.

Technical Discussion on Carts and Orders

Cart and **Order** have a common super type, **AbstractOrder**. Therefore, they are similar in many respects. Both **Carts** and **Orders** have a **code** attribute that serves as unique identifier. Unless you explicitly set a value for this identifier, SAP Commerce will use a randomly generated number, such as **1219393457297**.

It is not technically necessary to set a **delivery mode** or a **payment mode** when creating an Order. However, as the Order would not include fees for delivery and payment, the total of the Order would probably not be correct. In addition, it would be uncertain how to get the money for the Order if no payment mode is selected or how to send the ordered products if no delivery mode is selected. Therefore, make sure that at least one of the following options is implemented:

- Delivery mode and a payment mode are preset for the **anonymous** user
- Customer is required to select a payment mode and a delivery mode prior to actually completing the order process

Calculated Attribute and Calculation of Carts or Orders

ℹ Note

Promotions Require Additional Handling

If you use promotions, you need to not only update the Cart, but also explicitly update promotions. For details, see Promotion Module

Both Carts and Orders have an attribute called **calculated** of type boolean that states whether they are calculated, or not. Being calculated for a Cart or an Order means that the total of the current set of data (entries, total sum, currency, etc.) is correct and matches customer expectations. Whenever a **price total** related action is performed on a Cart or an Order, the Cart or Order is marked to be not calculated, such as:

- Product is be added to the cart
- Product is be removed from the cart
- User is changed
- Currency is changed

Being marked as not calculated for Cart or Order means that the **as-is** total price for the Cart/Order does not match the nominal total price. To make sure that Cart/Order becomes calculated, call the **CalculationService.calculate** or **CalculationService.recalculate()** method:

- **CalculationService.calculate()**: Calculates only those Cart/Order elements that are marked to be not calculated
- **CalculationService.recalculate()**: Recalculates the entire Cart/Order regardless of the value of the **calculated** attribute

A **calculated** attribute also exists for the individual entries of Cart and Order and also indicates whether the cart entry or order entry is calculated, or not. This allows the **CalculateService.calculate()** method to calculate only those entries which are marked as not calculated and does not require calculating all entries.

When one of the entries of Cart or Order has its **calculated** attribute set to (for example, because the number of products referenced by the entry has changed), then Cart/Order is also set to be not calculated.

Running a calculation of the Cart or Order on every occasion can create heavy load on the SAP Commerce Server or the database, particularly with a larger number of concurrent customers. Therefore, SAP Commerce has been implemented to leave the Cart in an non-calculated state and to require your business logic to trigger the calculation according to your business processes.

The actual calculation of Cart/Order is mainly done by the Cart/Order itself, and not by the PriceFactory (by default, Europe1). The PriceFactory is only used to retrieve three bits of information for CartEntry and OrderEntry.

Relevant Bit of Information	PriceFactory Methods Used	Description
The base price for a single unit of the product	**getBasePrice()** method • PriceFactory API Doc • Europe1PriceFactory API Doc	The base price retrieved from the PriceFactory is stored in the **basePrice** attribute of the CartEntryModel/OrderEntryModel). By using a custom implementation of the **getBasePrice()** method, you can affect the ordering process' price calculation mechanism.
The tax values for CartEntry/OrderEntry	**getTaxValues()** method • PriceFactory API Doc • Europe1PriceFactory API Doc	The tax values retrieved from the PriceFactory are stored in the **taxValues** attribute of the CartEntryModel/OrderEntryMode

Relevant Bit of Information	PriceFactory Methods Used	Description
		By using a custom implementation of the **getTaxValues()** method, you can affect the ordering process' tax calculation mechanism.
The discount values for the CartEntry/OrderEntry	**getDiscountValues()** method • PriceFactory API Doc • Europe1PriceFactory API Doc	The discount values retrieved from the PriceFactory are stored in the **discountValues** attribute of the CartEntry/OrderEntry. By using a custom implementation of the **getDiscountValues()** method, you can affect the ordering process' discount calculation mechanism.

Adding Entries to Cart or Order

By using an implementation of the **CartService.addNewEntry()** method you can create new entries for Cart or Order. Adding entries causes the Cart/Order to be marked as not calculated.

Sample code:

```
// retrieve one product
ProductModel p = ...

CartModel cart = CartService.getSessionCart();
// 5 is the number of items to add
// the Boolean false makes sure that a new CartEntry is created, even if there already was
// a CartEntry for the combination of product and unit
cartService.addToCart(cart, p, 5, p.getUnit(), false);
// cart is uncalculated now
cart.calculate();
```

Calling the **CalculateService.calculate()** method on the Cart will calculate only those CartEntries which have their **calculated** status set to **false**. By passing the boolean value **false** to the **CartService.addNewEntry()** method, the Cart creates individual entries for each call. Thus, after calling the code three times, there will be three individual CartEntries, each containing five products.

Removing Entries from Cart or Order

By using an implementation of the **CartService.addNewEntry()** method, you can remove entries from Cart or Order. Removing entries causes Cart/Order to be marked as not calculated.

Removing entries causes the cart to be marked as not calculated.

- **CartModel.setEntries(Collections.EMPTY_LIST);**: Removes all entries from the Cart/Order
- **CartModel.setEntries();**: Sets new list of entries on cart.

Creating Cart

When a Session is created, a Cart is reserved for the Session. The Cart is not actually created until it is used, for example, by adding entries or by calling the **CartService.getSessionCart()** method. Until the first use, the Cart is reserved but not instantiated. The actual creation of a Cart is a rather complex process. Therefore, it is recommended to delay cart creation until the Cart is really needed.

Using a Different Cart Implementation

Carts are created by the Session on demand. The factory default class used for carts is **Cart**. You can set the implementation of the Cart class that is used by overriding the value of the **default.session.cart.type** property and specifiying the type code of the ComposedType you want to use, such as

```
default.session.cart.type=InMemoryCart
```

to use the InMemoryCart. The InMemory cart is a non-persistent implementation of a session cart. Its main purpose is to speed up session cart handling by avoiding frequent database lookups and updates.

For details on overriding system property values, see Configuring the Behavior of SAP Commerce.

Cart Lifetime

Every active session has a distinct Cart, valid only for that individual session. Being assigned to the session, the Cart's lifetime is limited by the session's lifetime. If the Session times out or is explicitly invalidated, the Cart and its contents are invalidated as well. Until then, the Cart and its content remain available. Invalidated Carts are automatically removed by the session, there is no need to remove Carts manually.

A Cart is not reset after an order has been placed. In other words, the Cart's entries are not removed during or after order creation.

Removing Cart

To reset the Cart, for example after ordering, you have to:

- Remove the entire Cart, such as:

  ```
  // remove entire cart
  CartService.removeSessionCart();
  ```

- Manually erase the Cart's entries, such as:

  ```
  // remove all the Cart's entries
  CartService.getSessionCart().setEntries(Collections.EMPTY_LIST);
  ```

To create a new Cart after a Cart has been removed, simply request the Cart again, such as:

```
// get a new Cart
CartService.getSessionCart();
```

Creating Order

By factory default, orders in SAP Commerce are created by the **OrderService**. The **OrderService** offers implementation of a **createOrder()** method with an **AbstractOrderModel** object as a parameter.

```
public Order createOrder(final ComposedTypeModel orderType, final ComposedTypeModel entryType,
                         final AbstractOrderModel original, final String code);
```

This implementation creates an exact copy of the **AbstractOrder** object passed as a parameter, usually the CartModel of the Session. That means the **AbstractOrderModel** object is used as a template for the order to be created.Since the Order created by this method call will be a exact copy of the Cart, there are some implications:

- Order's creation date is the same as the Cart's creation date. In other words: Order will have the same creation date as the Cart from which it was copied. To set the Order's creation date to another date, you need to set the creation date attribute of the Order manually:

```
Date myDate = ...
OrderModel o =...
o.setCreationTime(MyDate);
```

- The **code** of the Order - in other words, the order number will be the same as the Cart's code. If a customer creates several orders from one single Session, all the orders will have the same order number by factory default. To ensure unique order numbers for a Session, you either have to set an order number explicitly or to remove the cart after using it to create the order.

When creating an Order this way, the Order has all the entries of the **AbstractOrder**. You do not need to create the entries yourself.

```
createOrder(ComposedType type, ComposedType entryType, AbstractOrder)
```

For more details on the implementation, see API documentation.

Gross Versus Net

Net prices are given without taxes. Gross prices are given including taxes. By factory default, Cart/Order in SAP Commerce is always either net or gross. Mixing net prices and gross prices across Cart or Order is not supported in order to avoid rounding issues.

You can define prices to be either net or gross at the cart (or order) level only, not at the individual entry level.

ℹ Note

Automatic Conversion

If the Cart is specified to be net, gross prices are converted to net. If the Cart is specified to be gross, net prices are converted to gross. The conversion must be done by the PriceFactory. The Europe1 PriceFactory does conversion automatically if needed. If you use a custom PriceFactory, you may have to implement conversion manually. For details, see europe1 Pricing System Guide.

Related Information

Users in Platform
Payment Transaction and Delivery Mode Handling
Using Templates to Define Info Texts for OrderEntries

Using Templates to Define Info Texts for OrderEntries

SAP Commerce stores an OrderEntry for every single order position. An OrderEntry has an attribute named **info** that you can use to store bits of information about the OrderEntry, for example the product's name.

Conventional Usage

SAP Commerce stores an OrderEntry for every single order position. An OrderEntry has an attribute named **info** that you can use to store bits of information about the OrderEntry (the product's name, for example). Via the **OrderEntry** type's **public String getInfo()** and **public void setInfo(String info**) methods, you can retrieve and specify the **info** attribute's value.

Infofield Templates

In addition, SAP Commerce allows you to define Velocity-based templates for the **info** attribute for an OrderEntry instance. These templates follow a **key=value** pattern and will be interpreted when an **OrderEntry** instance is created. This article is going to refer to them as infofield templates.

You define those infofield templates in the **`project.properties`** or **`local.properties`** file.

You can define infofield templates for the **Product** type and any subtype of the **Product** type by referencing the type identifier:

```
orderentry.infofield.${typecode} = My info text.
orderentry.infofield.product=Infofield template for the Product type.
```

ℹ Note

Lower case for infofield template definitions key required

Due to the way of how the infofield template definitions are processed, you need to spell the the key for the infofield template definition is spelt in lowercase.

This is because the parser which processes properties from the **`project.properties`** and **`local.properties`** files is case-sensitive and the pattern to match is in lowercase. Therefore, you need to make sure that your infofield template definition **key** is spelt in lowercase too, otherwise the parser will skip your definitions. The **value** for the property is not handled in a case-sensitive way.

The **`project.properties`** contains a sample setting to illustrate how to use these templates:

```
######################## ORDER SETTINGS ############################
#
# settings for order handling
#
#####################################################################

# pattern of what should be placed into the info attribute of each
# order entry (get/setInfo())
# If no pattern is defined, the product code is used in info field.
#
# see API documentation of AbstractOrder.createEntryInformation(aoe)
# for more useful informations.

orderentry.infofield.product=product "${code}" with name "${name}"
```

Infofield Template References to Other Attributes

The infofield template definition allows plain text, but also references to attribute values.

When the **OrderEntry** is created, the value of the referenced attributes are parsed and written to the **OrderEntry** 's **info** attribute. For example:

Infofield template definition	OrderEntry info attribute value
`orderentry.infofield.product=${code}`	The **code** of the Product.
`orderentry.infofield.product=${name}`	The **name** attribute of the Product. actual value set depends on the lan OrderEntry's creation.
`orderentry.infofield.product = This product was created at ${creationtime}.`	`This product was created a`

However, the values of referenced attributes are in fact written directly to the **info** attribute of the **OrderEntry**. No parsing or interpretation of values takes place. This may not be relevant in the case of atomic types, for example, as AtomicTypes are usually correctly represented; but for collection types, for example, you might not receive meaningful results.

Infofield template definition	OrderEntry info attribute value
`orderentry.infofield.jeans=Jeans "${code}" with baseproduct "${baseproduct}"`	`Jeans "CL1100-aaa015x04-0`
`orderentry.infofield.product=product "${code}" with price "${europe1Prices}"`	`product "HW2200-0521" wit` The **europe1Prices** attribute holds result ends up being the represen

Inheritance and Subtypes

An infofield template definition will only apply to the type itself, not to any subtypes. To apply an infofield template to a subtype, you will have to explicitly define the infofield template for the subtype again. For example, in the following code sample, both **Product** and the **Product** subtype **MyProduct** have an individual infofield template definition. If no infofield template was defined for **MyProduct**, then **MyProduct** would not have an infofield template definition (not even if **Product** had an infofield template definition).

```
orderentry.infofield.product    =       Product "${code}" with catalog version "${catalogversion}"
orderentry.infofield.myproduct  =       MyProduct "${code}" with catalog version "${catalogversion}
```

Infofield template definition	Pr ha int te de
`orderentry.infofield.product = Product "${code}" with catalog version "${catalogversion}"`	ye
`orderentry.infofield.myproduct = MyProduct "${code}" with catalog version "${catalogversion}"`	no
`orderentry.infofield.product = Product "${code}" with catalog version "${catalogversion}"` `orderentry.infofield.myproduct = MyProduct "${code}" with catalog version "${catalogversion}"`	ye

Grouping Cart Entries

The cart entry grouping functionality enables you to group cart or order entries. You can use this functionality, for example, for bundles, where entries belonging to one bundle will be grouped in one cart entry group .

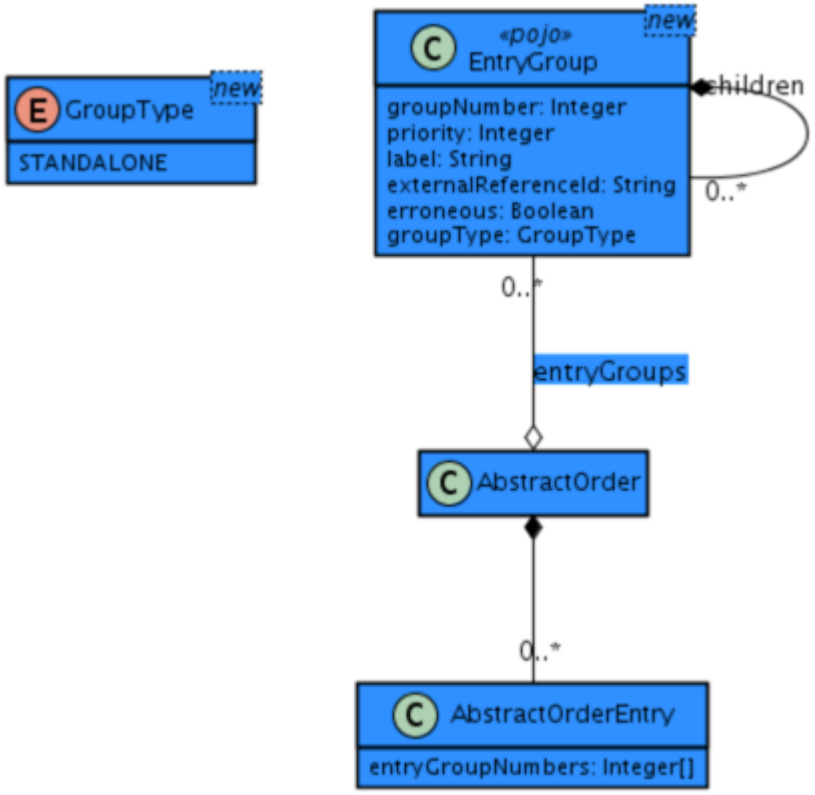
Overview

In the model, a cart entry group is represented by a POJO named `EntryGroup`, defined in the `core-beans.xml` file. Additionally, there is a dynamic enumeration named `GroupType`. For every different usage of cart grouping, define a new type.

Entry groups in one order are represented with the `entryGroups` attribute on `AbstractOrder`. Its type is a List of `EntryGroup`.

`AbstractOrderEntry` has the `entryGroupNumbers` attribute, which is a `groupNumber` of the `EntryGroup` this entry belongs to.

Here is the data model diagram for entry grouping:



`EntryGroupStructureValidateInterceptor` takes care that all `EntryGroups` have a unique `groupNumber` within one order, and that there is no cyclic dependency between them.

`EntryGroupService` provides basic methods for working with `EntryGroups`, such as getting an `EntryGroup` from an order for a specific `groupNumber`, or getting parent, or leaf groups. The `getGroupOfType` method gets a List of entry group numbers and finds `entryGroup` of the given type from the List.

Information about entry groups is added to Backoffice for **Cart and Order** (attribute `AbstractOrder.entryGroups`). It is visible on the **Positions and prices** tab.

For more information, see Cart Entry Grouping.

EntryGroup Properties

The `EntryGroup` properties include:

Property Name	Description
`groupNumber`	Uniquely identifies an entry group within the cart.
`label`	Describes the entry group. Could be used in the frontend, on the cart page, to show information about the group.
`groupType`	Type of grouping
`priority`	Priority of the group within the order

Property Name	Description
`externalReferenceId`	ID of the external item for which this group is created. For example, in case of bundles it would be an id of a specific bundle.
`erroneous`	A boolean value saying whether this group is in an erroneous state.
`children`	Each group can have 0 or more child `EntryGroups`.

Example

You can use cart grouping to group products in the cart or order, based on different types of business criteria. Grouping of products sold as bundles or packages is only one example.

For such an example case, you can introduce a new `GroupType`, called `BUNDLES`. Imagine an electronics store offers a Photography starter package containing two components - Camera and Lens with specific products. When the user adds the products belonging to this bundle to the cart, the cart will have 2 entries for the products belonging to the Lens and Camera component. Besides them, in the cart there could be other standalone products, or other bundles.

Cart grouping enables you to group products of one bundle in the cart, and to show them on the cart page together as a group. For the Photography starter package bundle, which is identified for example with `id=photoStarter`, one `EntryGroup` is created, with `groupNumber=0` and `groupType=BUNDLES`. This `EntryGroup` would have `externalRefernceId= photoStarter`, and this is a root entry group. The `label` field would have the `Photography starter package` value, which could be shown on the cart page for this bundle.

For each component, additional `EntryGroup` is created, one with `externalReferneceId = camera` and `entryGroup=1`, and another with `externalReferneceId = lens` and `entryGroup=2`. These two would be added to the `EntryGroup` with id 0 as children. Only the root entry group is assigned to the cart.

You could use the `erroneous` flag in the `EntryGroups` to indicate that something is wrong in the bundle. For example, for a complex bundle there could be a rule that the user has to select a specific number of one product. You can show it on the cart page as a piece of information for the user that this entry group (and its related bundle) needs to be fixed.

Payment Transaction and Delivery Mode Handling

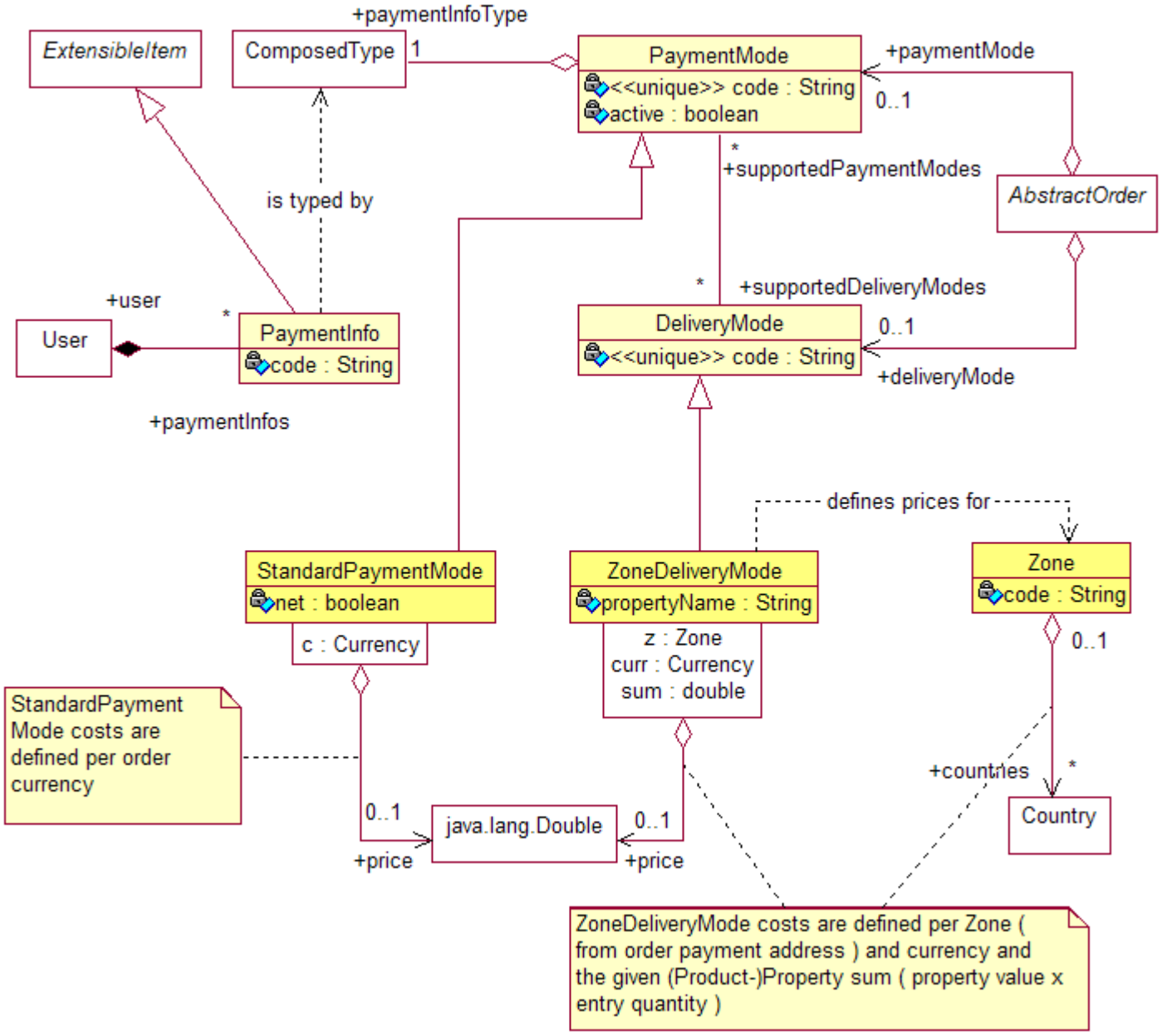
SAP Commerce offers built-in support for management of fees for delivery of orders and payment methods. When a set-up of payment and delivery costs is done, the costs are automatically calculated and added to an order. SAP Commerce uses the best-matching elements of the set-up in terms of currency, user location and so on.

For the end-user, costs for transactions and delivery are calculated automatically. The end-user does not have to do more than specifying the mode of delivery and the way of payment. SAP Commerce automatically selects the delivery mode cost and transaction cost and adds the cost to the order.

The SAP Commerce StoreFoundation comes with four individual payment modes as part of the sample data:

- Invoice
- Credit card
- Debit
- Advance

Payment and Delivery UML Diagram



Designing a Payment and Delivery Cost Model

A payment and delivery cost configuration consists of

- payment modes

  A payment mode: invoice, credit card, etc

- transaction cost

  A single potential transaction cost for using a payment mode. Can be specific per currency or converted from the base currency.

- delivery modes

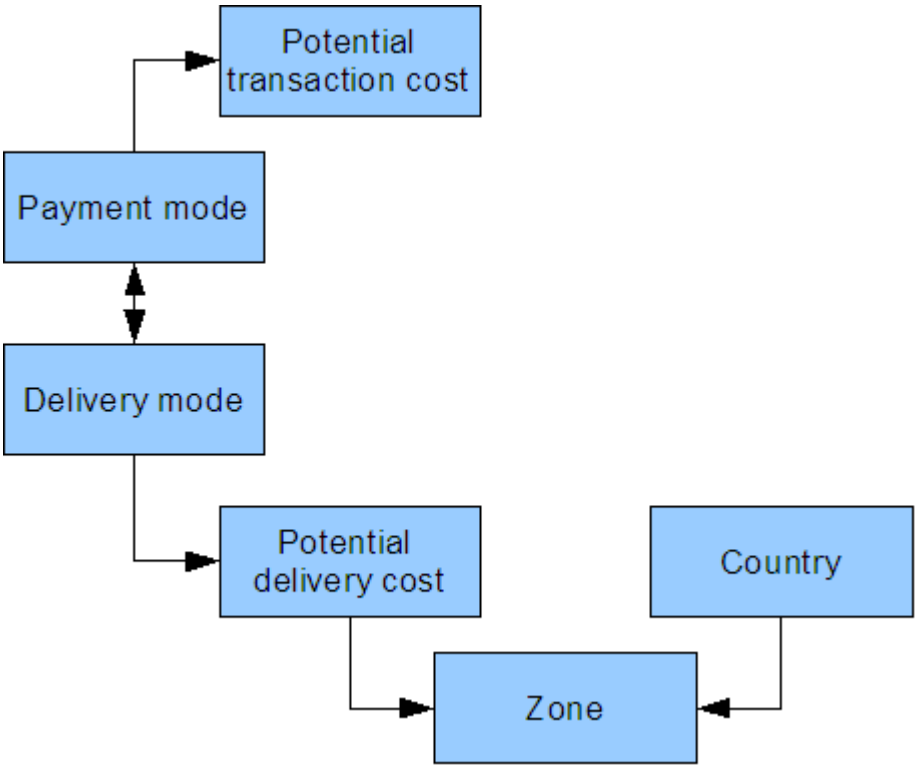
  In essence, summary data about a logistics company.

- delivery mode costs

  A potential price for a delivery mode: how much is the fee for the logistics company to deliver to this zone?

- zones

  A list of countries to which a logistics company delivers shipments at one single cost



Setting up a Transaction Cost Model

For each payment mode you allow, you set up all possible configurations of transaction fees per currency, such as:

Payment Mode	Potential Transaction Costs
Advance	2 EUR 3 USD 1.5 GBP
Debit	0 EUR 1.5 USD 0 GBP
Credit card	1 EUR 1 EUR 1 GBP

Be aware that by factory default these payment mode costs apply to any payment transaction and do not differ by country. In other words: if a credit card payment transaction is set at a cost of 1.5 USD, these 1.5 USD will apply worldwide and to any customer using credit card payment. If you need to set up country-specific transaction costs, you will need to use a custom payment mode implementation.

Setting up a Delivery Cost Model

Designing a delivery cost model is more complex than designing a transaction cost model as delivery modes are related to payment modes on the one hand and countries on the other.

A delivery mode is ordinarily connected to at least one payment mode. In addition, it can have any number of potential delivery costs. These potential delivery costs depend on the zone of delivery, which consists of any number of countries. In other words: a delivery mode applies to a combination of payment mode and countries. Technically it would be possible to set up a delivery mode

without a payment mode, but normally you want the customer to only be able to select payment modes matching a chosen delivery mode and vice versa.

To set up a delivery cost model, you therefore need to identify

- which logistics companies (delivery mode),
- the individual prices the logistics companies have (potential delivery cost), andIn addition to that, you need to set up order total thresholds. These thresholds define a minimum which an order must reach or exceed for the potential delivery cost to match. For example, at an order total of 20 EUR, the delivery cost could be 10 EUR; while starting with an order total of 50 EUR, the delivery cost could be 5 EUR - or vice versa. A delivery cost can only apply for an order if the order total is at least as high as the threshold (that is, equal or higher). A delivery cost is not applicable for an order if the order total is lower than the threshold. The lowest delivery cost threshold should start at 0 EUR.
- the countries they deliver to at one single individual price (zone and countries).

A zone must be distinct in its set-up for a delivery mode, it is not allowed for a zone to contain countries that are part of another zone of the same delivery mode. If a country was present in two zones for one single delivery mode, SAP Commerce could not determine the actual price of delivery because the zones would not be distinct for the country.

In other words, you need to find all possible unique constellations of prices for different countries for the logistics companies you want to employ. The following list shows three fictitious logistics companies, the prices they charge for delivery into different countries on the left hand side and the resulting price constellation in the table on the right hand side.

- Logistics company X
  - Germany, Austria, Switzerland: 3.50 EUR
  - Spain, Portugal, France: 4.5 EUR
  - USA, Canada: 5 EUR
  - Australia: 12 EUR
- Logistics company Y
  - Germany5: 5 EUR
  - Austria, Switzerland: 6.5 EUR
  - Spain, Portugal, France: 7 EUR
  - USA: 10 EUR
  - Australia: 15 EUR
  - Canada: ***(not delivered to)***
- Logistics company Z
  - Germany, Austria: 4.5 EUR
  - Switzerland: 6 EUR
  - Spain, Portugal, France: 5 EUR
  - USA, Australia: 8 EUR
  - Canada: 10 EUR

Delivery Mode (Logistics Company)	Potential Delivery Cost	Zone and Countries
X	3.5 EUR 4.5 EUR 5 EUR 12 EUR	**X-Central Europe** - Germany, Austria, Switzerland **Western Europe** - Spain, Portugal, France **X-Northern America** - USA, Canada **Australia** - Australia

Delivery Mode (Logistics Company)	Potential Delivery Cost	Zone and Countries
Y	5.0 EUR 6.5 EUR 7 EUR 10 EUR 15 EUR	**DE** - Germany **Y-ATCH** - Austria, Switzerland **Western Europe** - Spain, Portugal, France **Y-USA** - USA **Australia** - Australia
Z	4.5 EUR 6 EUR In addition to that, you need to set up order total thresholds. These thresholds5 EUR 8 EUR 10 EUR	**DEAT** - Germany, Austria **Z-CH** - Switzerland **Western Europe** - Spain, Portugal, France **Z-AUSUSA** - USA, Australia **Z-CND** - Canada

The **Western Europe** and **Australia** delivery zones contain the same countries for all delivery modes they are used in. Therefore, it is possible to use one single delivery zone each and assign different fees depending on the logistics company. Using single delivery zones is not possible for the other country combination because these are not distinct and parts of the zones overlap:

E.g. it would not be possible to use the **Z-AUSUSA** delivery zone for the other uses of Australia as well because **Z-AUSUSA** contains the USA as well, which would then be included in all other references of Australia.

Determination of Transaction Costs

The SAP Commerce calculates the actual cost for a certain payment mode (for an order calculation, for example) like this: Of all available potential transaction costs, SAP Commerce selects the one that matches the payment mode and the given currency.

What happens if no match is available in a given currency depends on whether a base currency is available or not. Non-base currencies have a conversion factor related to a base currency - convertible currencies, so to speak. For example, USD could be set as a non-base currency with a conversion factor of 1.5:1 with respect to the base currency.

- For non-base currencies, SAP Commerce does an internal conversion and uses the converted transaction cost.
- Base currencies are not converted.

  In other words: if EUR is the base currency for USD and no transaction cost is available for USD, then SAP Commerce will convert and use the transaction cost for EUR. If no transaction cost is available for the base currency EUR either, then no transaction cost will be available.

Determination of Delivery Mode Costs

Determining a delivery mode cost is more complex than determining a payment mode cost.

This is due to the fact that delivery costs depend on various factors:

- the country the customer is in (precisely: the delivery zone the customer is in, determined via the customer's country)
- the total of the order
- the currency

In addition, the list of available delivery modes is intended to be limited to the ones that support the currently active payment mode. If delivery mode **QuickExpress** supports invoice and debit payment and the user sets the payment mode to be credit card, then **QuickExpress** is not supposed to be available as a delivery mode option.

ℹ Note

Link between payment mode and delivery mode not pre-implemented outside of SAP Commerce StoreFoundation

The payment mode-related and delivery mode-related basic functionality are technically independent of one another. SAP Commerce does not ensure automatically that only delivery modes related to the currently set payment mode are available. You can, for example, you can set any combination of payment mode and delivery mode to an order.

You will have to manually ensure that only supported combinations of payment mode an delivery mode are selected. The SAP StoreFoundation, for example, does not allow selecting a delivery mode that is not supported by the currently active payment mode by deactivating unsupported combinations.

SAP Commerce calculates delivery costs in two steps:

1. Pre-selecting eligible delivery costs from all potential delivery costs

   A delivery cost is relevant for pre-selection if all of the three conditions are met:

   - the delivery cost applies to the zone the customer is in, and
   - the total of the cart reaches or exceeds the minimum value (the threshold) of the delivery cost, and
   - the currency of the delivery cost matches the currency of the cart.
2. Selecting the delivery cost whose minimum is closest to the order total

What happens if no match is available in a given currency depends on whether a base currency is available or not. Non-base currencies have a conversion factor related to a base currency - convertible currencies, so to speak. For example, USD could be set as a non-base currency with a conversion factor of 1.5:1 with respect to the base currency.

- For a non-base currency, SAP Commerce does an internal conversion and uses the converted delivery cost.
- Base currencies are not converted.

  In other words: if EUR is the base currency for USD and no delivery cost is available for USD, then SAP Commerce will convert and use the delivery cost for EUR. If no delivery cost is available for the base currency EUR either, then no delivery cost will be available.

Delivery Mode and Payment Mode When Ordering

ℹ Note

Only part of order process discussed

This section only discusses the payment- and delivery-related aspects of ordering.

When a cart is converted into an order, SAP Commerce

- determines the cost for the payment method specified by the customer
- determines the cost for the delivery method specified by the customer
- creates a persistent copy of the payment mode data to document the payment mode selected
- creates a persistent copy of the delivery address
- creates a persistent copy of the payment address

This makes sure that the order always holds a valid combination of payment mode data and delivery mode data and that the addresses kept with the order remain stable. That way, even when the current data in SAP Commerce is changed (for example, because a user changes home), the existing orders will remain the same.

The delivery mode and the payment mode are stored with the order for documentation purposes even if no costs for payment transaction and / or delivery actually arise. In an actual business case the customer will either have to select or be assigned a payment mode and a delivery mode in any case - no matter whether it is going to cost them money or not.

Technical Discussion

The SAP Commerce delivery cost and payment management functionality is implemented in two core extensions: **DeliveryZone** and **PaymentStandard**.

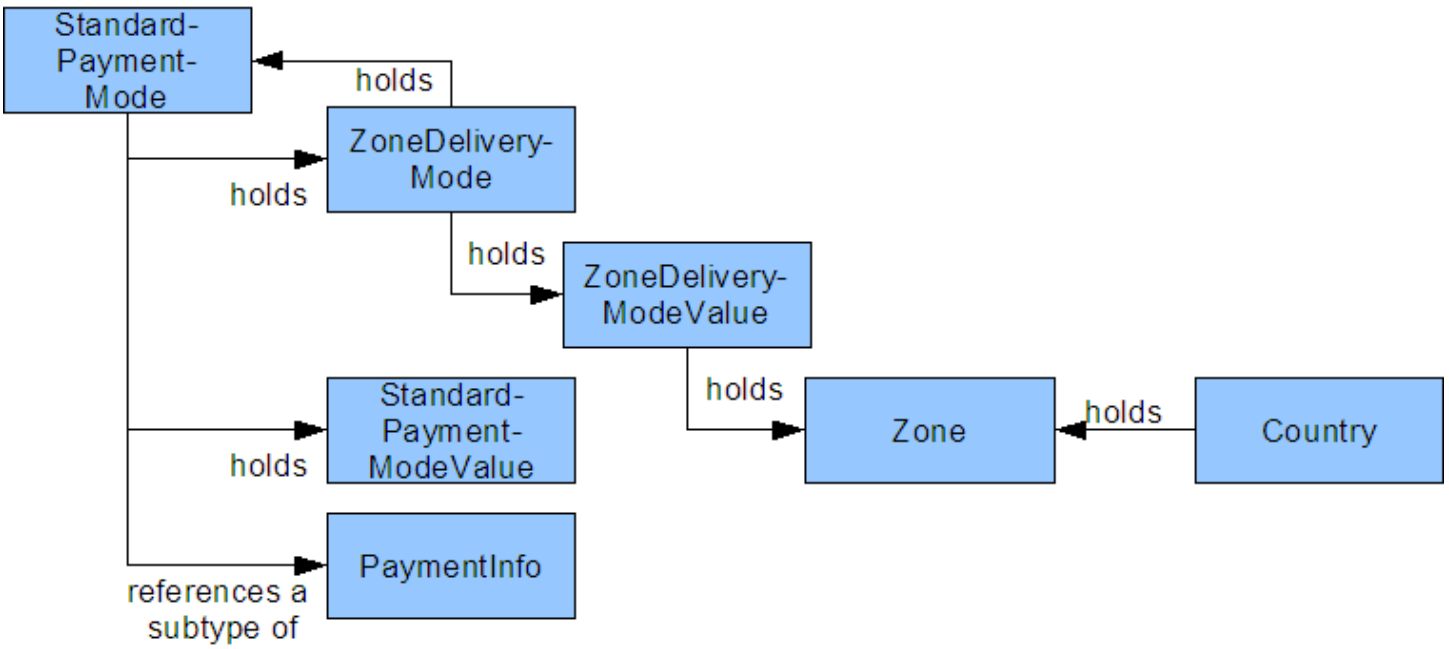
The **PaymentStandard** extension and the **DeliveryZone** extension are designed to enable interdependencies between delivery modes and payment modes.

Please also refer to the DeliveryZone Extension JavaDoc and the StandardPayment Extension JavaDoc.

The types listed here are part of the sample data created by the SAP Commerce StoreFoundation; Platform itself does not come with any pre-implemented payment or delivery modes by factory default.

Type System Model

Type System Model



For more information, see paymentstandard Extension and deliveryzone Extension.

Commerce Quotes Items

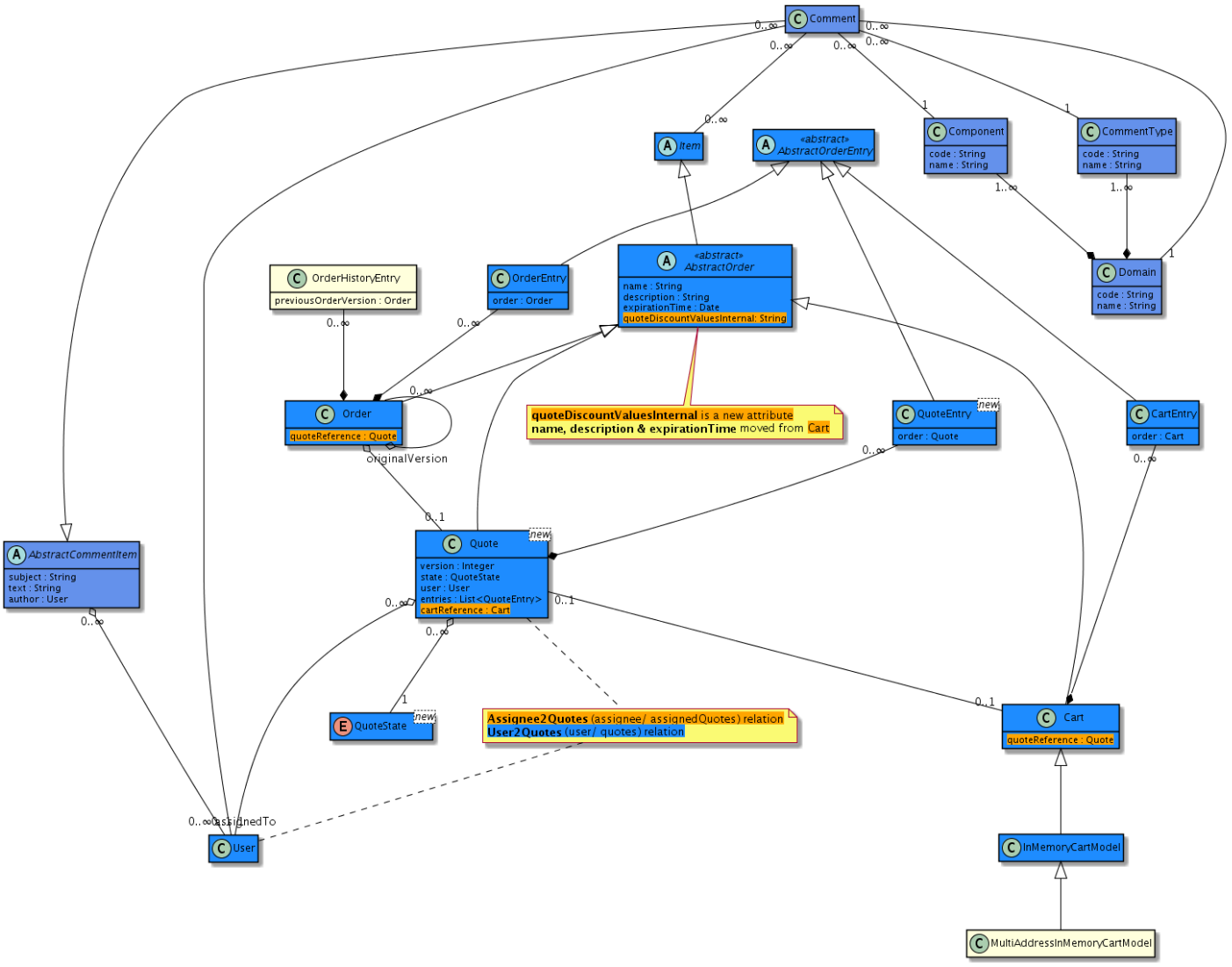
Commerce Quotes enables buyers to create quotes and negotiate the final price of an order using the storefront. Learn about the items included in platform core and platform services that enable Commerce Quotes.

Platform Core

Three items in platform core enable Commerce Quotes: `Quote`, `QuoteEntry`, and `QuoteState`.

- `Quote`: A quote is a container (like a cart) that holds quote entries instead of cart entries and some additional information such as the quote version and status. A quote can be created from a cart using `CreateQuoteFromCartStrategy` (default implementation is `DefaultCreateQuoteFromCartStrategy`). A cart can be created from a Quote object using the `CreateCartFromQuoteStrategy` (default implementation is `DefaultCreateCartFromQuoteStrategy`).
- `QuoteEntry`: Same as CartEntry that has its own deployment table just like any subtype of `AbstractOrderEntry`.
- `QuoteState`: An enum for holding Quote-related statuses.

This diagram illustrates how these items are related to Cart and Order classes:



Platform Services

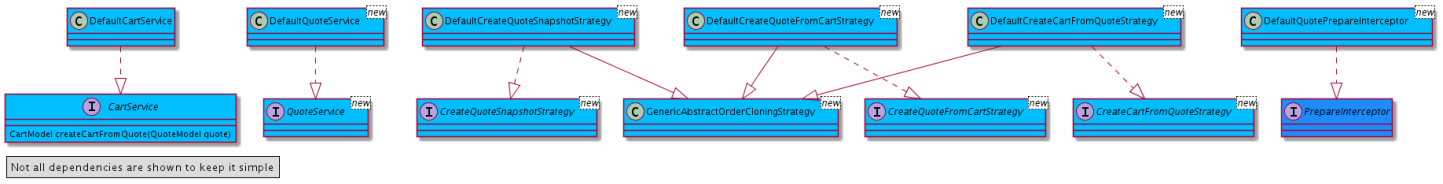
These platform services are used for Commerce Quotes:

- `QuoteService`: An interface that provides basic Quote retrieval services, services for creating a quote from cart, and a quote snapshot. For example:
    - Quote retrieval: `getCurrentQuoteForCode(final String code);`, `getQuoteForCodeAndVersion(final String code, final Integer version);`
    - Creating a quote from a cart: `createQuoteFromCart(final CartModel cart);`

    - Creating a quote snapshot: `createQuoteSnapshot(QuoteModel quote, QuoteState quoteState);`

- `DefaultQuoteService`: This class provides the default implementation for the services mentioned above. The service methods that deal with creating a quote from cart and creating a quote snapshot have individual strategies plugged in which can be overridden. For extension points, override the existing service methods by extending the super class.
- `GenericAbstractOrderCloningStrategy`<T extends `AbstractOrderModel`, E extends `AbstractOrderEntryModel`, O extends `AbstractOrderModel`>: A generic strategy for abstract order cloning, taking the target order and order entry classes as generic parameters. It acts as a wrapper around `CloneAbstractOrderStrategy`, which provides common functionality for the `AbstractOrder` cloning related strategies.
- `CreateQuoteSnapshotStrategy`: An interface that offers a method to create a new quote snapshot based of the supplied Quote and the Quote status.
- `DefaultCreateQuoteSnapshotStrategy` extends `GenericAbstractOrderCloningStrategy<QuoteModel, QuoteEntryModel, QuoteModel>`: This class provides the default implementation of the `CreateQuoteSnapshotStrategy` interface. A newly created snapshot has an incremented version number. For extension points, the `createQuoteSnapshot(final QuoteModel quote, final QuoteState quoteState)` strategy method provides a `postProcess` hook. To make use of this hook, you need to subclass this strategy.
- `CreateCartFromQuoteStrategy`: An interface that offers a method to create a new `Cart` based on the supplied `Quote`.
- `DefaultCreateCartFromQuoteStrategy` extends `GenericAbstractOrderCloningStrategy<CartModel, CartEntryModel, QuoteModel>`: This class provides the default implementation of the `CreateCartFromQuoteStrategy` interface. The newly created `Cart` is a clone of the supplied `Quote`. For extension points, the `createCartFromQuote(final QuoteModel quote)` strategy method provides a `postProcess` hook. To make use of this hook, you need to subclass this strategy.
- `CreateQuoteFromCartStrategy`: An interface that offers a method to create a new `Quote` based on the supplied `Cart`.
- `DefaultCreateQuoteFromCartStrategy` extends `GenericAbstractOrderCloningStrategy<QuoteModel, QuoteEntryModel, CartModel>`: This class provides the default implementation of the `CreateQuoteFromCartStrategy` interface. The newly created `Quote` is a clone of the supplied `Cart`. For extension points, the `createQuoteFromCart(final CartModel cart)` strategy method provides a `postProcess` hook. In order to make use of this hook, you need to subclass this strategy.
- `DefaultQuotePrepareInterceptor`: This default prepare interceptor sets the code, version, statues, and name if they are not set by the delegating service of the new `QuoteModel`.

This diagram shows the relationship of the different items:



Related Information

Commerce Quotes

Technical Overview

Enabling Commerce Quotes in Other Storefronts

Quote Statuses

User Types

User Type Identification and State Selection Strategies