Commerce

Generated on: 2024-12-03 13:54:41 GMT+0000

SAP Commerce | 2205

Public

Original content: https://help.sap.com/docs/SAP_COMMERCE/9d346683b0084da2938be8a285c0c27a?locale=en-US&state=PRODUCTION&version=2205

Warning

This document has been generated from the SAP Help Portal and is an incomplete version of the official SAP product documentation. The information included in custom documentation may not reflect the arrangement of topics in the SAP Help Portal, and may be missing important aspects and/or correlations to other topics. For this reason, it is not for productive use.

For more information, please visit the https://help.sap.com/docs/disclaimer.

Commerce Services Module

The Commerce Services module allows you to configure your products, multiple carts, and manage client consent and future stock.

Features

Product Configurator Support
Example Implementation of a Product Configurator
Consent Management
Multiple Saved Carts
Future Stock Availability

Architecture

commerceservices Extension
commercefacades Extension
commercewebservicescommons Extension
ycommercewebservicestest Extension
ycommercewebservices Extension
yoccaddon Extension
OCC AddOns Architecture
OCC Extension Architecture
commercewebservices Extension
commercewebservicestests Extension
yocc Extension
yocctests Extension
commerceservicesbackoffice Extension

Implementation

Cart Merging
Extending CommerceCartService
Extending CommerceCheckoutService
Converters and Populators
Value Resolvers
Populating the In-Store Customers List with IoT Device Data
Commerce Quotes
Cart Entry Grouping Functionality
Voucher Redemption, Validation, and Brute Force Attack Detection
Omni Commerce Connect

Commerce Services Features

The Commerce Services module allows you to configure your products, multiple carts, and manage client consent.

Product Configurator Support

SAP Commerce supports various configurable products.

Consent Management

B2C Accelerator provides functionality for data subjects (a natural person such as a customer, contact, or account) to give consent to collect or transfer their personal data.

Multiple Saved Carts

SAP Commerce Accelerator provides a saved cart feature that allows users to save one or more carts for later use.

Future Stock Availability

The Future Stock Availability feature indicates when inventory is going to be replenished with new stock.

Verification Token

This feature provides API for verification token code management. Client can request verification token code for specific purpose, and upon validation, the system delivers it to a designated channel. This code can further be utilized for verifying identities or transactions, thereby enhancing system security.

Product Configurator Support

SAP Commerce supports various configurable products.

SAP Commerce supports a mix of products configurable by different configurators:

- in the same catalog
- in the same cart
- in the same order

In the checkout page, the configuration of a configurable product is validated to check if the product has been modified correctly and the customer can proceed to the checkout.

To implement a product configurator, see an example implementation.

For information about specific configurators, see:

- Product Configuration with SAP Variant Configuration and Pricing
- SAP CPQ Integration for Configurable Products

Example Implementation of a Product Configurator

With the configurable products functionality, you can define different variants for your products. Learn how to recognize a configurable product and possible configuration options of this kind of product.

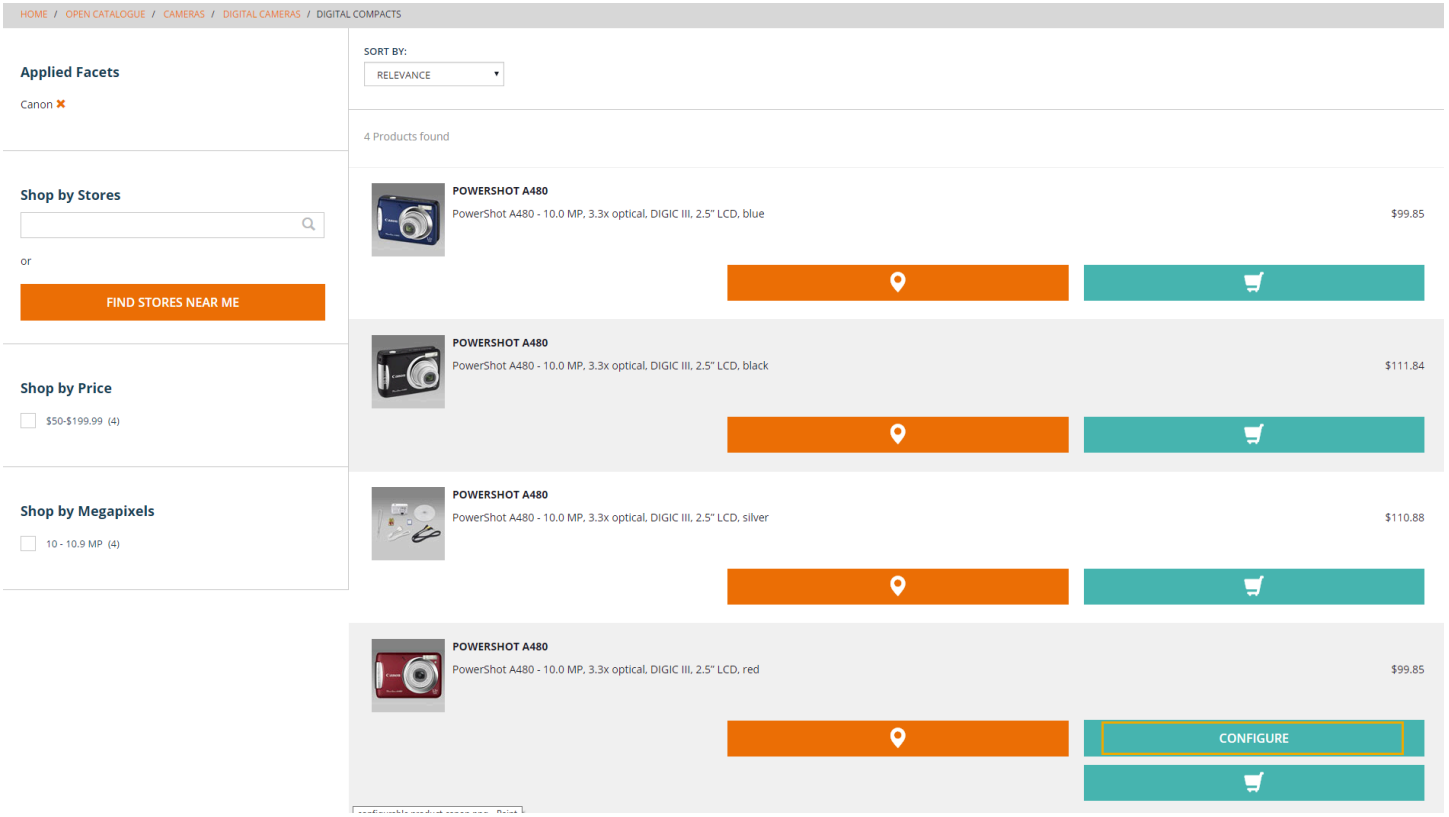
All configurable products have Configure button that navigate a customer to **Configure Product Options** page. On the **Configure Product Options** page, you can configure the following:

- **Engraved Text**
- **Font Size**
- **Font Type**

Example of configurable product: **POWERSHOT A480** camera, red.

You can change a configuration of a product that is already in the cart, just click Change Configuration button next to the configurable product.

If a product is configurable, you can navigate to **Configure Product Options** page directly from the **Product Details** page. Click the Configure button on the **Product Details** page.



Steps

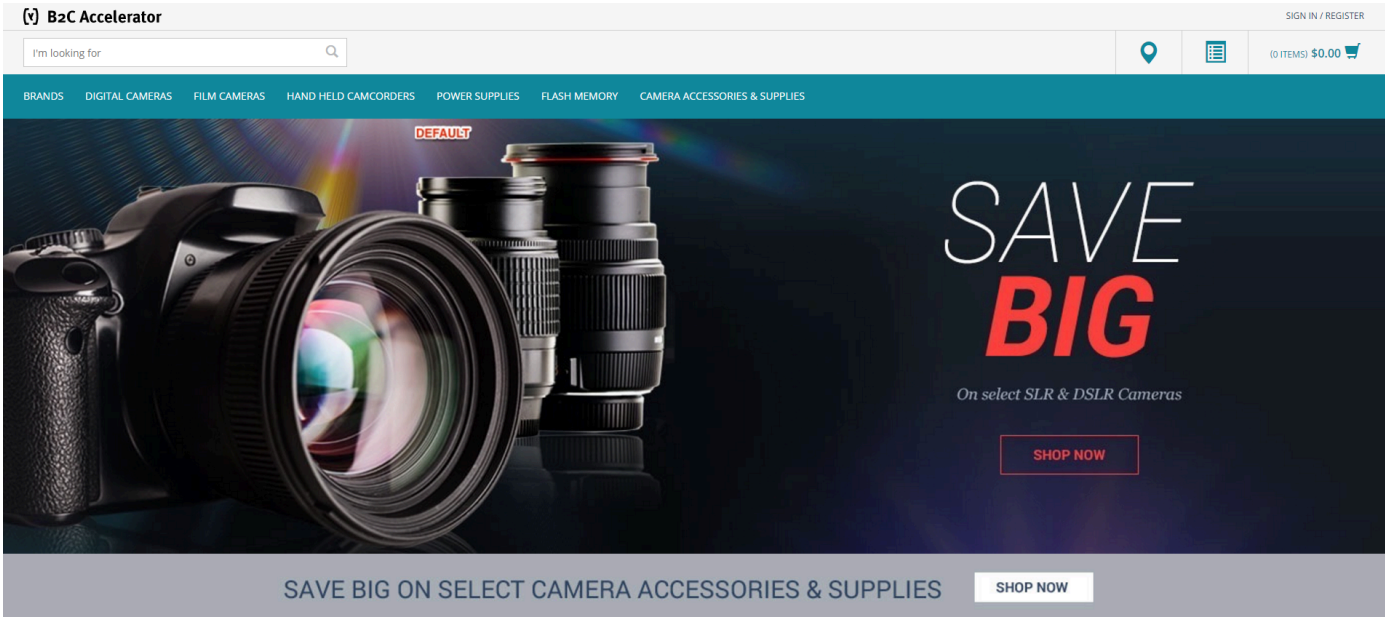
Follow the steps to learn how to recognize a configurable product and how to configure this kind of product.

Prerequisites

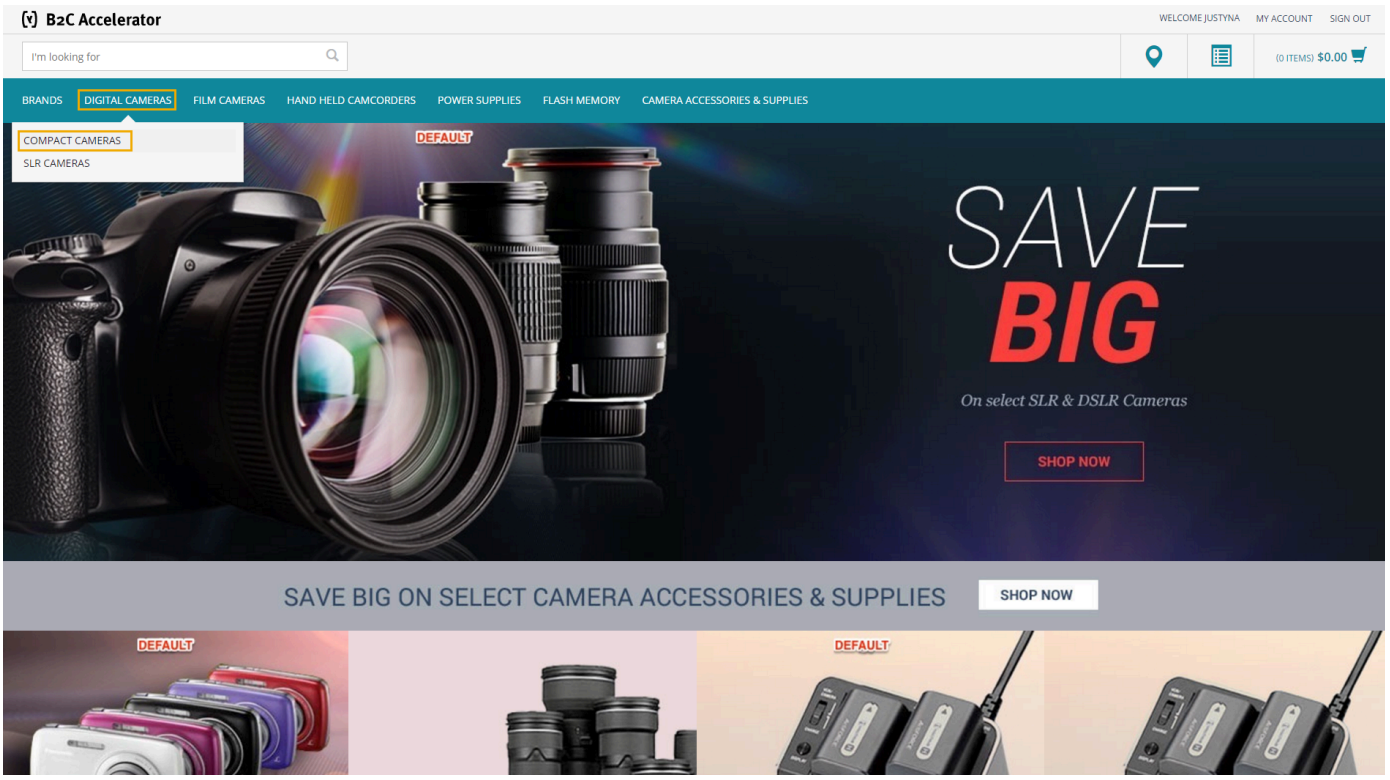
Install B2C Accelerator. For information about installation, see the Installing B2C and B2B Accelerator Locally.

Procedure

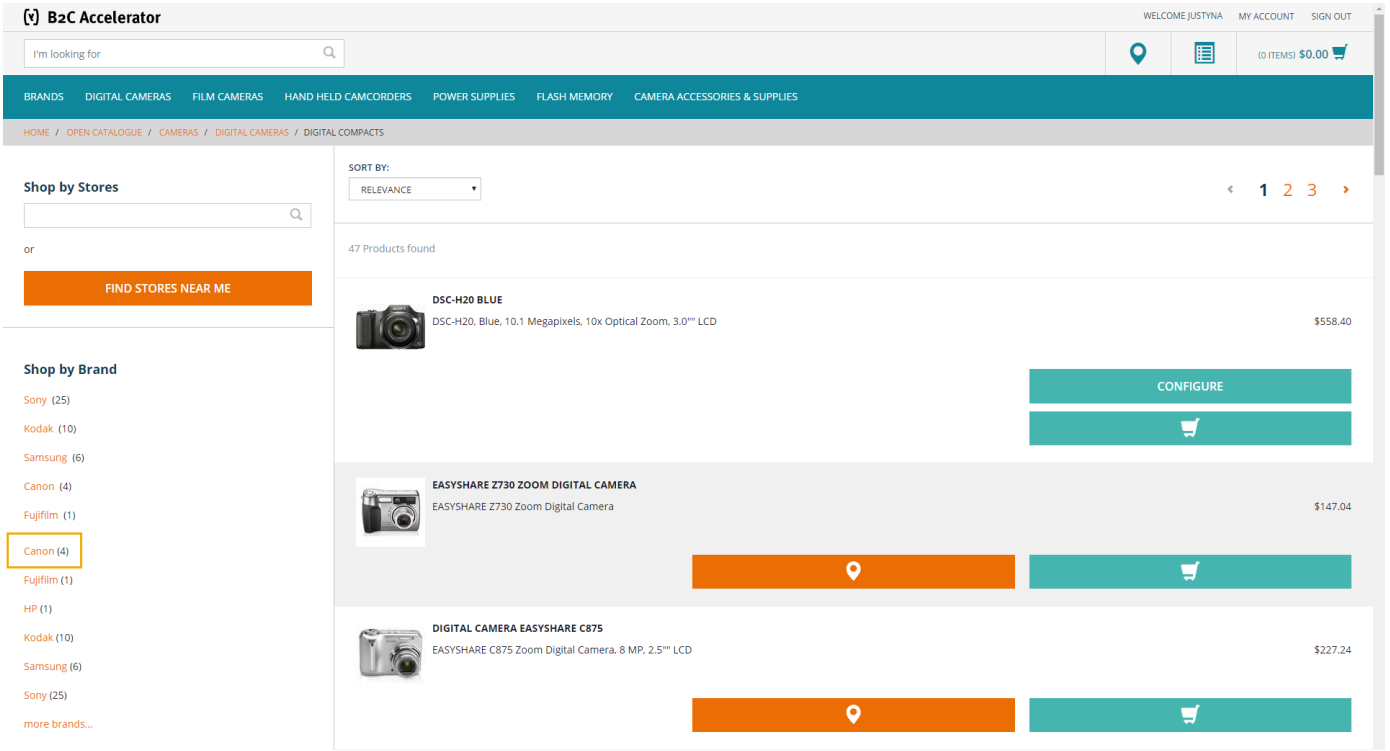
1. Navigate to `https://electronics.local:9002/yacceleratorstorefront/electronics/en/`.



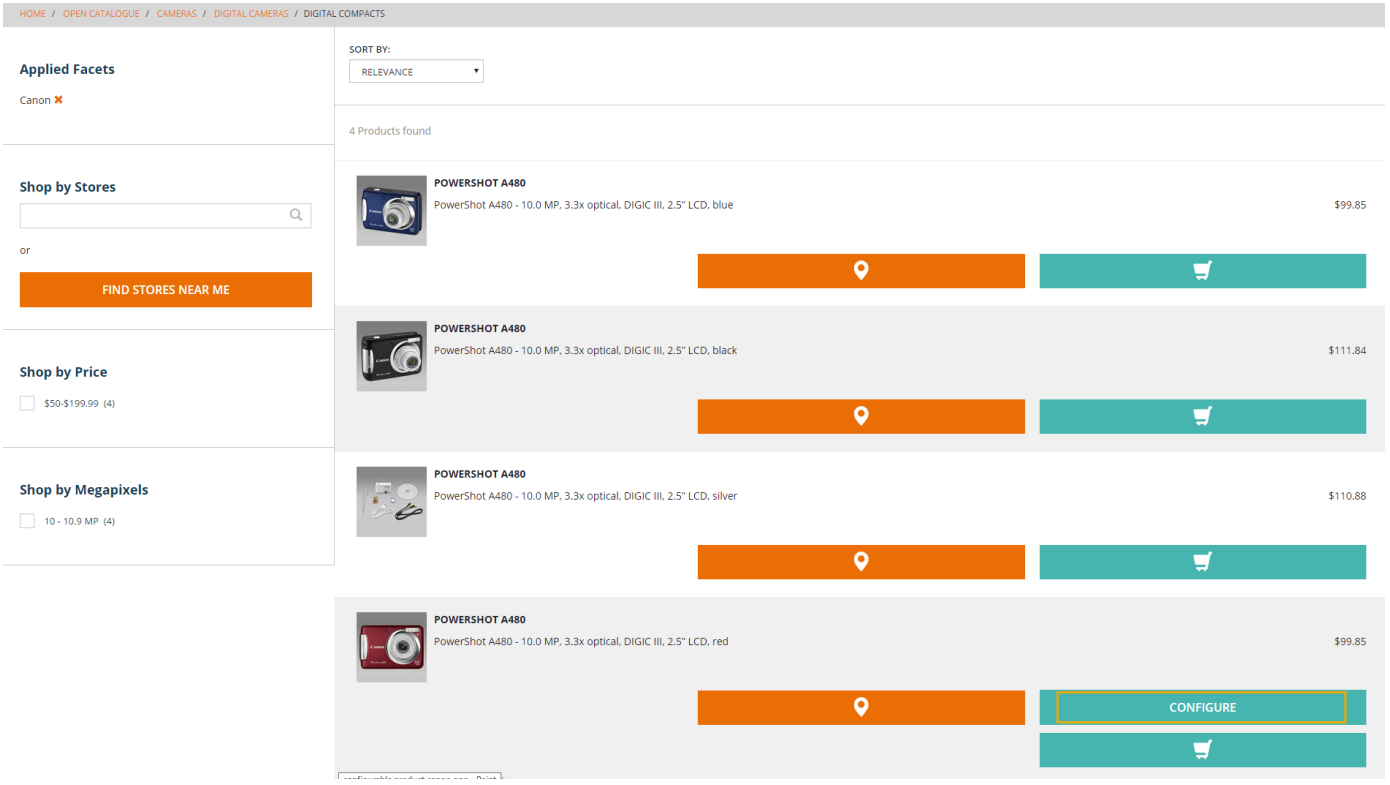
2. From the Digital Cameras drop-down list, select Compact Cameras.



3. From the **Shop by Brand** menu select Canon.

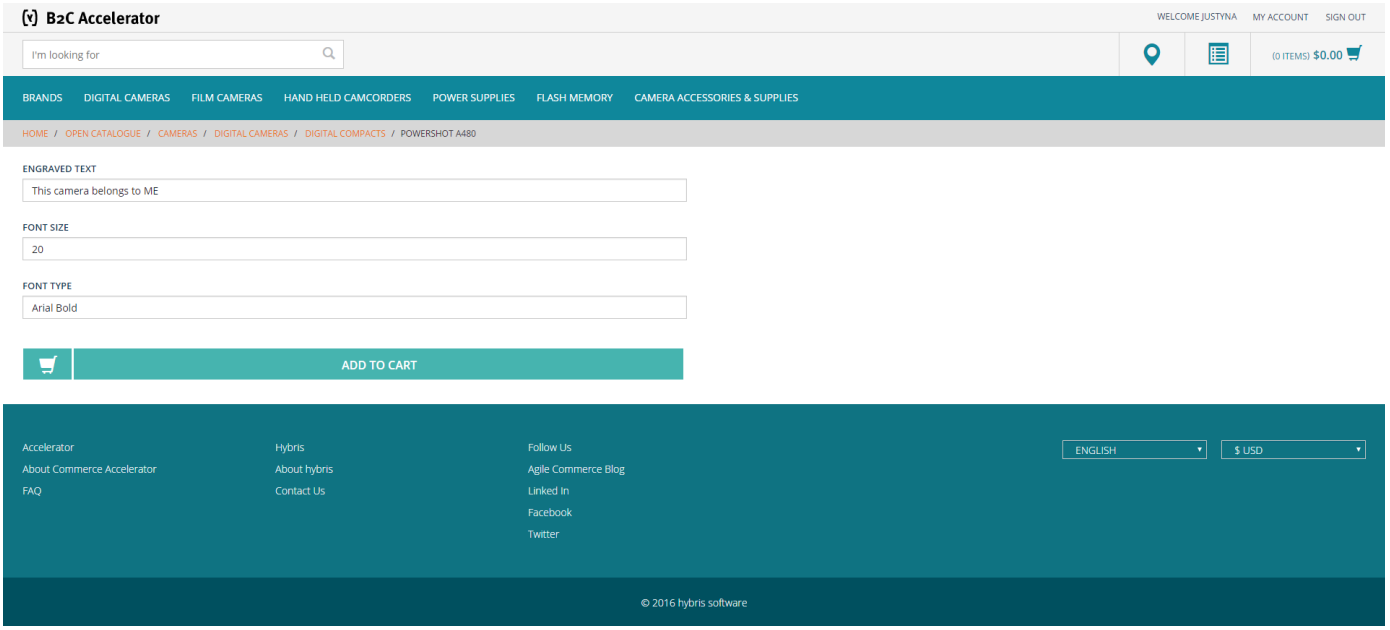


4. Navigate to the **POWERSHOT A480** camera and click the Configure button.



The **configure product options** page appears.

5. Provide the product configuration:
   - Engraved Text
   - Font Size
   - Font Type



6. Click the Add to Cart button.

   Your product is added to cart. And you are directed to cart.

7. You can continue shopping or navigate to check out. For more information about checkout, see the Managing Multi-Step Checkout Strategies document.

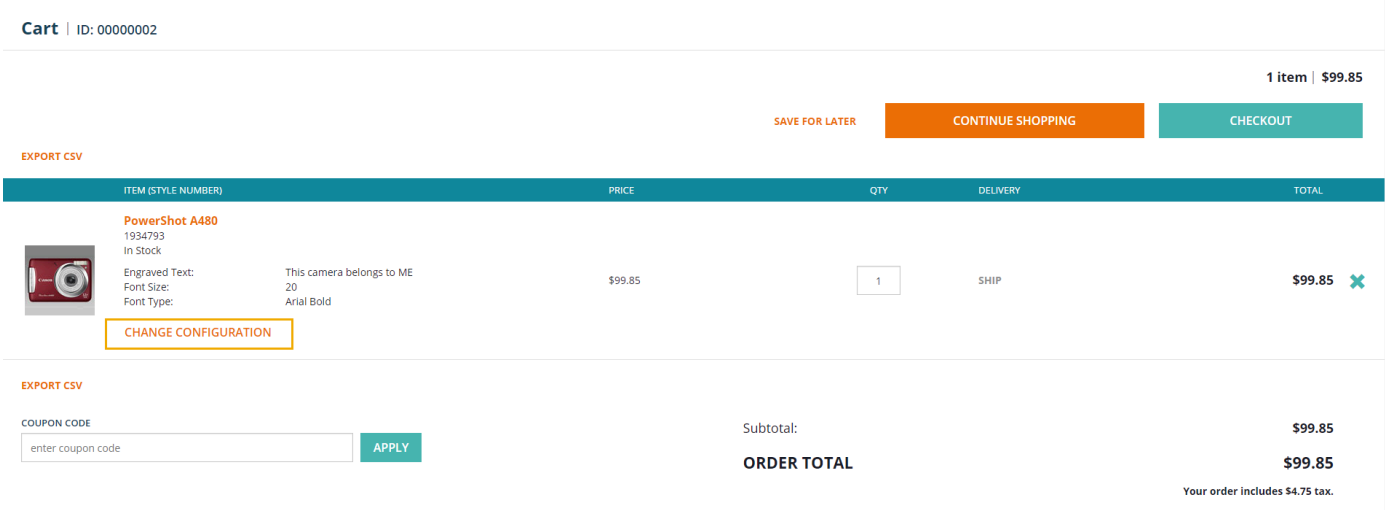
Configurable Product in Cart

Context

You can change a configuration of a product that is already in the cart.

Procedure

1. Navigate to your cart.
2. Click the Change Configuration button.

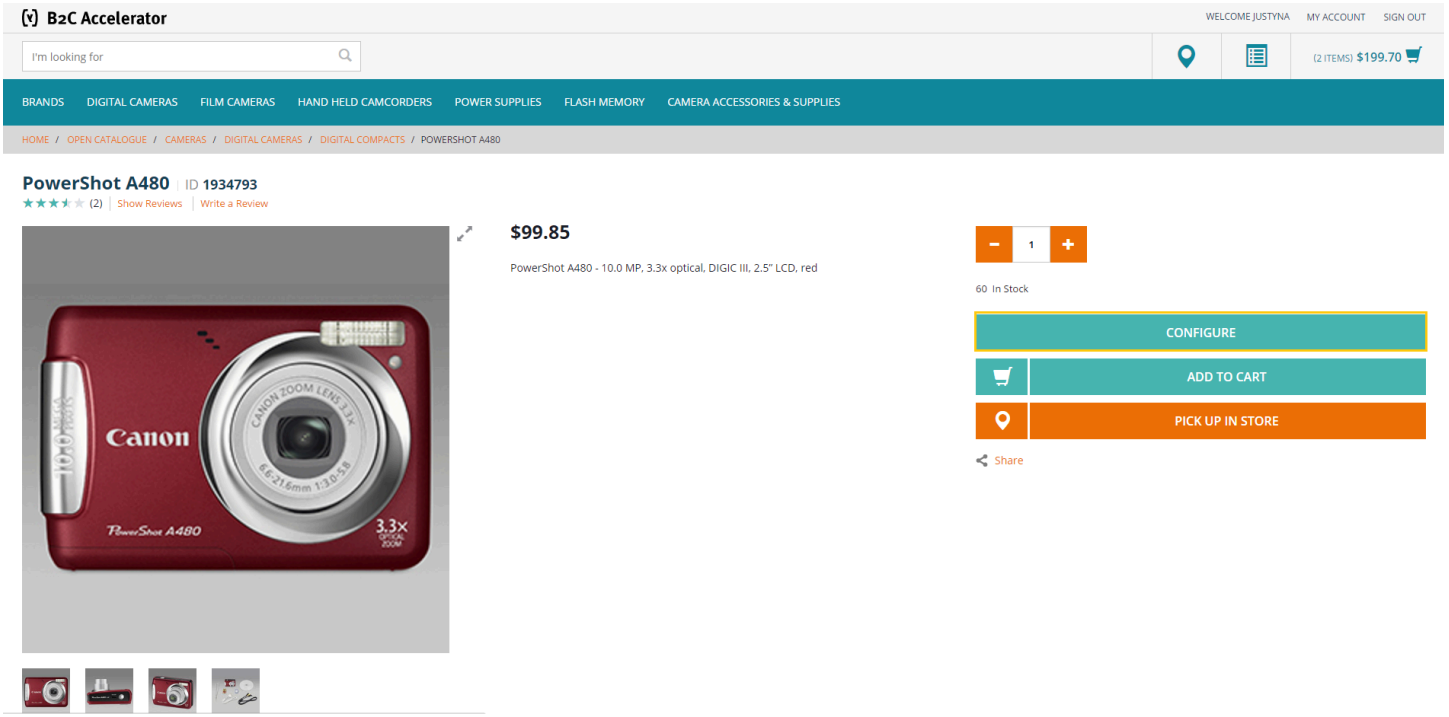


   You are redirected to the **configure product options** page.

3. Update your product and click the Update button to update a product.
4. You can continue shopping or navigate to check out. For more information about checkout, see the Managing Multi-Step Checkout Strategies document.

Product Details Page

You can navigate to **configure product options** directly from the **Product Details** page, if a product is configurable. To do this, click the Configure button.



Related Information

textfieldconfiguratortemplateservices Extension
textfieldconfiguratortemplatebackoffice Extension
textfieldconfiguratortemplatefacades Extension
textfieldconfiguratortemplateaddon Extension
Product Configurator Support

Consent Management

B2C Accelerator provides functionality for data subjects (a natural person such as a customer, contact, or account) to give consent to collect or transfer their personal data.

Introduction

Users give consent to collect and use their data in particular entry points of the storefront. By default, pages where users create an account serve as entry points, but other entry points can be created. Consent templates are used to display the consent text in the entry points, as well as the Consent Management page, where users can change their consent settings. Any changes made to consent settings are recorded in the back end and can be viewed in the Backoffice Administration Cockpit.

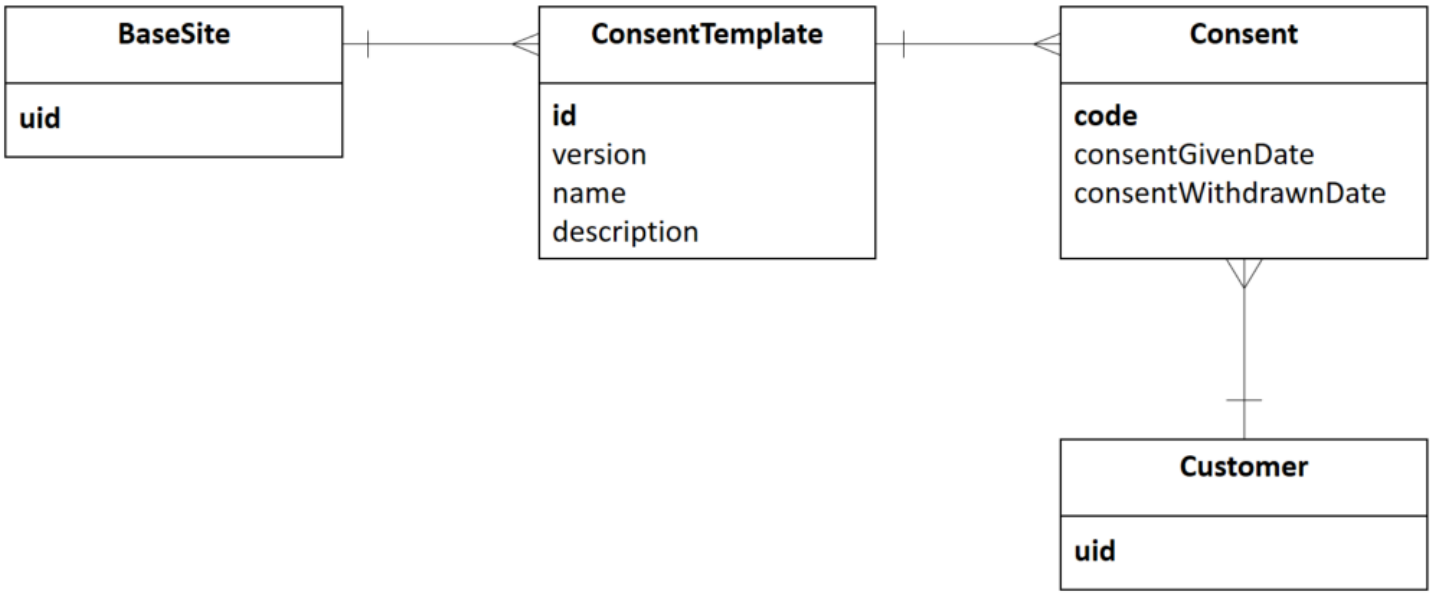
ℹ Note

Although Commerce B2C Accelerator includes templates you can use for Consent management, you can only manually implement Consent templates for Commerce B2B Accelerator. For more information on how to add the Consent feature to a B2C or a B2B storefront, see one of the following sections:

- **B2C storefront**: Consent Templates
- **B2B storefront**: Adding Consent Entry Points

Data Model

This diagram illustrates the data model for consent management, which is in accordance with the SAP Central Consent Management Solution:



Each `BaseSite` (for example, electronics) can have multiple `ConsentTemplates`. `ConsentTemplate` has the following properties:

- `id`
- `version`
- `name`
- `description`
- `baseSite`

By default, one template is available in the sample data: `MARKETING_NEWSLETTER`. Here is an example of the properties for the default template in the electronics storefront (note that the Java properties of the model class are used as column headers, and not the actual database column headers):

PK	id	version	name	description	baseSite
PK1	MARKETING_NEWSLETTER	0	Newsletter Subscription Consent	I approve to use my personal data for receiving e-mail newsletters for marketing campaigns	electronics

When a user gives consent, a consent record is created in the back end. The record captures both the business event and corresponding time information. Here is an example of two consent records. The first record shows that a user gave consent and then withdrew it at a later date; the second record shows that the same user gave consent and has not withdrawn it.

code	Customer uid	ConsentTemplate Primary Key	consentGivenDate	consentWithdrawnDate
00000	customer1	PK1	2017-06-01 9:00:23.656	2017-06-20 23:12:56.412
00001	customer1	PK1	2017-07-03 11:45:33.552	null

Because consent records in the database store both the business event and time, they can be used as an audit trail record.

Giving Consent

Storefront users can give consent when they register an account. At any time, they can change their consent settings in the Consent Management page.

Consent Logs

When a user gives or withdraws consent, a consent record is created in the back end and can be viewed in the Backoffice Administration Cockpit.

Consent Templates

Consent templates define the content displayed to users in the storefront. A storefront can have multiple consent templates and one template can be used for multiple storefronts.

Consent Template Versions

When the terms and conditions of a consent template changes, a new consent template version can be created to reflect those changes. Users of the storefront must then give explicit consent to the new version of the template.

Adding Consent Entry Points

Users give consent at entry points, such as the customer registration page. Learn how to add entry points to other areas of the storefront, and to other storefronts such as powertools.

Anonymous Consent Management

Anonymous Consent Management gives anonymous users control over the tracking of their data. Anonymous users can grant or decline their consent for applications that collect and process personal data.

Personal Data Erasure

Customers can choose to close their account in the storefront and have their personal data deleted. Cron jobs included in the `yacceleratorcore` extension clean up data related to deactivated customers.

Giving Consent

Storefront users can give consent when they register an account. At any time, they can change their consent settings in the Consent Management page.

By default, the B2C storefronts include a consent template that asks users for consent to use their personal data for receiving marketing newsletters.

Giving Consent Upon Account Registration

When users create an account in the storefront, they have the option to give consent for the default consent template. There are two areas where users can create an account:

- In the Sign In/Register page
- In the order confirmation page after placing an order as a guest

In both cases, a check box appears where users can give their consent to receiving marketing newsletters.

Note

By default, the account creation pages are the only ones where the consent check box appears. To learn how to add consent entry points in other areas of the storefront, see Adding Consent Entry Points.

Consent Management Page

At any time, registered users can change their consent settings by navigating to **My Account** > **Consent Management**. The Consent Management page displays all consent templates for the storefront, along with a switch that users can turn on or off for each template. Any change to consent settings on this page are recorded and displayed in the Backoffice Administration Cockpit. Note that any consent templates created using Backoffice are also displayed on this page.

Parent topic: Consent Management

Related Information

Consent Logs
Consent Templates
Consent Template Versions
Adding Consent Entry Points
Anonymous Consent Management
Personal Data Erasure

Consent Logs

When a user gives or withdraws consent, a consent record is created in the back end and can be viewed in the Backoffice Administration Cockpit.

Consent records capture both the business event and corresponding time stamp. Because consent records in the database store both the business event and time, you can them as an audit trail record if necessary.

Viewing Consent Records

To view consent records in Backoffice, navigate to **User** > **Consent**. If no consent records appear, click **Search** to display all consent records. To refine your search, click to switch to Advanced Search mode, and enter the required search criteria. Use the advanced search to view all consent records for a particular user or consent template.

Consent records display the following information:

- Consent record code
- Customer
- Date consent was given
- Date consent was withdrawn (displays **Null** if consent has not been withdrawn)
- Consent template

Exporting Consent Records

If required, you can export consent records to a CSV file. To do so, search for the records you want to export, click the Export List to CSV icon ( ), and then complete the wizard.

Parent topic: Consent Management

Related Information

Giving Consent
Consent Templates
Consent Template Versions
Adding Consent Entry Points
Anonymous Consent Management
Personal Data Erasure

Consent Templates

Consent templates define the content displayed to users in the storefront. A storefront can have multiple consent templates and one template can be used for multiple storefronts.

Default Consent Templates

B2C Accelerator includes one consent template by default (MARKETING_NEWSLETTER) that asks users to consent to receiving marketing newsletters. This template is available in all B2C storefronts (electronics, apparel-uk, and apparel-de).

You can view and edit consent templates in Backoffice by navigating to **User** > **Consent Template**.

Adding Consent Templates

You can add new templates to the storefront using Backoffice:

1. In Backoffice, navigate to **User** > **Consent Template**.
2. Click the plus (+) icon to open the Create New Consent form.
3. Enter the template **ID**, select the **Site** where the template will be used, and then enter the **Version** of the template.
4. Click **Next**.
5. Enter the template **Name**. The name appears in the Consent Management page.
6. Enter the template **Description**, then click **Done**.

The Consent Management page displays the name of the newly created template.

Note

Creating a consent template in Backoffice does not create an entry point for the user to give consent; the template only appears in the Consent Management page. To create a new entry point for the consent template, see Adding Consent Entry Points.

Parent topic: Consent Management

Related Information

Giving Consent
Consent Logs
Consent Template Versions
Adding Consent Entry Points
Anonymous Consent Management
Personal Data Erasure

Consent Template Versions

When the terms and conditions of a consent template changes, a new consent template version can be created to reflect those changes. Users of the storefront must then give explicit consent to the new version of the template.

For example, if there is a consent template for a marketing newsletter that advises customers about popular products, you may want to add promotions to that newsletter that are specific to the customer based on their purchasing history. In this case, you can create a new template with an updated version number that the customer must give consent to in the storefront.

Note

To create a new template version, you must create a new template so that the customers can give their explicit consent for the changed terms and conditions. Changing the version number for an existing template does not withdraw a customer's consent for the old version.

When a new consent template version is created:

- the new template version replaces the old version in the storefront.
- the customer's consent is not withdrawn from the old version.
- the customer's consent is not given in the new version; customers must give their consent to the new version in the Consent Management page.

New consent template versions are created in Backoffice:

1. Log into Backoffice and navigate to **User** > **Consent Template**.
2. Click the plus (+) icon to open the Create New Consent form.
3. Enter the template **ID**. This ID must be identical to the version you are updating.
4. Enter the **Site** where this template appears. The site must be the same as the version you are updating.
5. Enter the template **Version** number. This number should increment by one from the old version.
6. Click **Next**.
7. Enter the template **Name** and **Description**. These should be different from the previous version to reflect the changes in terms and conditions.
8. Click **Done**.

The new template version replaces the old version in the Consent Management page of the storefront.

Note

Consent templates should not be deleted. The consent versions are kept in the system to have a proof of the changes to the consent declarations. If a consent statement is not valid anymore, the withdrawal of the consent must be in place. The rights to delete the consent templates should be revoked from the users.

Parent topic: Consent Management

Related Information

Giving Consent
Consent Logs
Consent Templates
Adding Consent Entry Points
Anonymous Consent Management
Personal Data Erasure

Adding Consent Entry Points

Users give consent at entry points, such as the customer registration page. Learn how to add entry points to other areas of the storefront, and to other storefronts such as powertools.

The following example shows how to add a consent entry point to the Edit Units page, which you can access by navigating to **My Company** > **Units** in the powertools storefront. We'll create a marketing campaign consent template for this entry point.

Creating your own consent entry point involves the following main steps:

- Setting up the consent template
- Updating the controllers
- Adapting the front-end form

Setting Up the Consent Template

1. In the `powertools` extension, add a new consent impex file along with build-time generated impex localizations. For more information on build-time generated localizations, see Localization of ImpEx Using Build Time Generation.
   - `powertoolsstore/resources/powertoolsstore/import/sampledata/stores/powertools/consents.impex`

```
# -----------------------------------------------------------------------
# [y] hybris Platform
#
# Copyright (c) 2017 SAP SE or an SAP affiliate company.  All rights reserved.
#
# This software is the confidential and proprietary information of SAP
# ("Confidential Information"). You shall not disclose such Confidential
# Information and shall use it only in accordance with the terms of the
# license agreement you entered into with SAP.
# -----------------------------------------------------------------------
# ImpEx for Consents for Electronics Store
#% impex.setLocale( Locale.GERMAN );
```

```
$siteUid=powertools

INSERT_UPDATE ConsentTemplate;id[unique=true];name;description;version[unique=true];baseSite(uid)[unique=true,default=$site
;BUSINESSUNIT_MARKETING;"Receive e-mails for Business Unit Marketing campaigns";"I approve to use my personal data for rece
```

- powertoolsstore/resources-lang/powertoolsstore/import/sampledata/stores/powertools/consents.vt

```
# -----------------------------------------------------------------------
# [y] hybris Platform
#
# Copyright (c) 2017 SAP SE or an SAP affiliate company.  All rights reserved.
#
# This software is the confidential and proprietary information of SAP
# ("Confidential Information"). You shall not disclose such Confidential
# Information and shall use it only in accordance with the terms of the
# license agreement you entered into with SAP.
# -----------------------------------------------------------------------
#
# ImpEx for Consents for Electronics Store
#

# Language
\$lang=$lang.toLowerCase()

\$siteUid=powertools

#set( $consents = $query.load('consents') )
UPDATE ConsentTemplate;id[unique=true];name[lang=\$lang];description[lang=\$lang];version[unique=true];baseSite(uid)[unique
#foreach( $consent in $consents )
;$consent.key;"$consent.values.name";"$consent.values.description";0;;
#end
```

- powertoolsstore/resources-lang/powertoolsstore/import/sampledata/stores/powertools/consents_en.properties

```
# -----------------------------------------------------------------------
# [y] hybris Platform
#
# Copyright (c) 2017 SAP SE or an SAP affiliate company.  All rights reserved.
#
# This software is the confidential and proprietary information of SAP
# ("Confidential Information"). You shall not disclose such Confidential
# Information and shall use it only in accordance with the terms of the
# license agreement you entered into with SAP.
# -----------------------------------------------------------------------
Consent.BUSINESSUNIT_MARKETING.name=Receive e-mails for Business Unit Marketing campaigns
Consent.BUSINESSUNIT_MARKETING.description=I approve to use my personal data for receiving e-mail newsletters for Business
```

2. Add a new property denoting the ID of the consent template to use for the B2B unit page in the `project.properties` file in the `powertools` extension:

```
businessunit.consent.id.powertools=BUSINESSUNIT_MARKETING
```

Updating the Controllers

1. In the `commerceorgaddon` extension, modify `MyCompanyPageController.java` to add a constant for the consent template ID property name. This Java file is located in `commerceorgaddon/acceleratoraddon/web/src/de/hybris/platform/commerceorgaddon/controllers/pages`.

```
protected static final String BUSINESSUNIT_CONSENT_ID = "businessunit.consent.id";
```

2. Add a new method that sets up the consent template data in the page model:

```
protected void addBusinessUnitConsentDataToModel(final Model model)
{
    final String consentId = getSiteConfigService().getProperty(BUSINESSUNIT_CONSENT_ID);
    if (StringUtils.isNotBlank(consentId))
    {
        final ConsentTemplateData consentData = getConsentFacade().getLatestConsentTemplate(consentId);
        model.addAttribute("consentTemplateData", consentData);
    }
}
```

3. In `BusinessUnitManagementPageController.java`, add a call to `addBusinessUnitConsentDataToModel` to the `editUnit` (GET) method just before the return statement. This Java file is located in `commerceorgaddon/acceleratoraddon/web/src/de/hybris/platform/commerceorgaddon/controllers/pages`.

```
@RequestMapping(value = "/edit", method = RequestMethod.GET)
@RequireHardLogIn
public String editUnit(@RequestParam("unit") final String unit, final Model model) throws CMSItemNotFoundException
{
    ...
    addBusinessUnitConsentDataToModel(model);
    return ControllerConstants.Views.Pages.MyCompany.MyCompanyManageUnitEditPage;
}
```

4. In the `editUnit` (POST) method, add a code block to handle the `giveConsent` and `withdrawConsent` logic, just after the call to `b2bUnitFacade.updateOrCreateBusinessUnit(...)`. Also add a call to `addBusinessUnitConsentDataToModel(...)` in the catch block. This sets up the page model in the event of returning to the edit page after an error:

```
@RequestMapping(value = "/edit", method = RequestMethod.POST)
@RequireHardLogIn
public String editUnit(@RequestParam("unit") final String unit, @Valid final B2BUnitForm unitForm,
        final BindingResult bindingResult, final Model model, final RedirectAttributes redirectModel)
                throws CMSItemNotFoundException
{
```

```
        ...
        ...


        try
        {
            b2bUnitFacade.updateOrCreateBusinessUnit(unit, unitData);
            final ConsentForm consentForm = unitForm.getConsentForm();
            if (consentForm != null)
            {
                if (consentForm.getConsentGiven())
                {
                    getConsentFacade().giveConsent(consentForm.getConsentTemplateId(), consentForm.getConsentTemplateVersion());
                }
                else if (consentForm.getConsentCode() != null)
                {
                    getConsentFacade().withdrawConsent(consentForm.getConsentCode());
                }
            }
        }
        catch (final ModelSavingException e)
        {
            LOG.error(String.format("Failed to save unit. Possibly non-unique id (original id: [%s] new id: [%s]).",
                    unit, unitData.getUid()), e);
            GlobalMessages.addErrorMessage(model, "form.global.error");
            bindingResult.rejectValue("uid", "form.b2bunit.notunique");
            addBusinessUnitConsentDataToModel(model);
            return ControllerConstants.Views.Pages.MyCompany.MyCompanyManageUnitEditPage;
        }
        GlobalMessages.addFlashMessage(redirectModel, GlobalMessages.CONF_MESSAGES_HOLDER, "form.b2bunit.success");

        return String.format(REDIRECT_TO_UNIT_DETAILS, urlEncode(unitForm.getUid()));
    }
```

5. Add the necessary import statements to each file.

Adapting the Front-End Form

1. Add a `consentForm` property to `B2BUnitForm.java` (located in `commerceorgaddon/acceleratoraddon/web/src/de/hybris/platform/commerceorgaddon/forms`):

```
public class B2BUnitForm
{
    ...
    private ConsentForm consentForm;


    ...

    public ConsentForm getConsentForm()
    {
        return consentForm;
    }

    public void setConsentForm(final ConsentForm consentForm)
    {
        this.consentForm = consentForm;
    }
}
```

2. In `b2bUnitForm.tag`, add a block just above the buttons to handle the display of the consent checkbox. This file is located in `commerceorgaddon/acceleratoraddon/web/webroot/WEB-INF/tags/responsive/company`.

```
...
            <c:if test="${ not empty consentTemplateData }">
                <form:hidden path="consentForm.consentTemplateId" value="${consentTemplateData.id}" />
                <form:hidden path="consentForm.consentTemplateVersion" value="${consentTemplateData.version}" />
                <form:hidden path="consentForm.consentCode" value="${consentTemplateData.consentData.code}" />
                <div class="checkbox">
                    <label class="control-label uncased">
                        <c:choose>
                            <c:when test="${not empty consentTemplateData.consentData && empty consentTemplateData.consentData.c
                                <form:checkbox path="consentForm.consentGiven" checked="checked" />
                            </c:when>
                            <c:otherwise>
                                <form:checkbox path="consentForm.consentGiven" checked="" />
                            </c:otherwise>
                        </c:choose>
                        <c:out value="${consentTemplateData.description}" />
                    </label>
                </div>
            </c:if>
...
```

3. Add a `tablib` statement for the JSTL core `taglib`.

Users can now navigate to the Edit Units page of the powertools storefront and select the checkbox to give consent to receiving marketing newsletters.

Parent topic: Consent Management

Related Information

Giving Consent

Anonymous Consent Management

Anonymous Consent Management gives anonymous users control over the tracking of their data. Anonymous users can grant or decline their consent for applications that collect and process personal data.

This functionality allows the management of consent for both anonymous and registered users. The anonymous users can manage the consent of their anonymous user data, which can eventually be associated to their registered user data. The registered users can view and edit their consents within a new **Consent Management** section of the **My Account** area.

When users visit the site for the first time, a banner appears at the top of the page and prompts them to agree or deny to consent. When users register, they can give consent. However, if registered users do not enable the consent checkbox, the system considers this to be a refusal to give consent and the message banner no longer appears.

Anonymous Consent Popups

Anonymous users can decide whether they want their data to be tracked by accepting or declining consent. The consent popups appear at the top of the storefront and display the name of a particular user tracking service. They also display a description of the particular tracking service when the chevron is clicked.

When an anonymous user accepts or declines to give consent, the popup disappears, and the decision is processed with the following code from `acc.consent.js`:

```
...
    changeConsentState:function(element, consentState){
        var consentCode = ($(element).closest('.consentmanagement-bar').data('code'));
        $.ajax({
            url: ACC.config.encodedContextPath+"/consents/"+consentCode+"?state="+consentState,
            type: 'POST',
            success: function () {
                $(element).closest('.consentmanagement-bar').hide();
            }
        });
    },
...
```

Anonymous Consent Templates

Consent templates reflect what anonymous user consent is needed for and are located in Backoffice Administration Cockpit **User > Consent Template**. Each user tracking service should have at least one consent template.

The same consent template can have multiple versions and the latest version is displayed in the **Version** column. When a particular consent template is modified and a new version is created, then a new consent popup gets displayed for the user to accept or decline in the storefront.

Exposed Consent Template

The anonymous template can be displayed or hidden on the storefront by toggling the Exposed flag to **True** or **False**:

You can also configure the **Exposed** attribute item types in `commerceservices-items.xml` as shown following:

```
<itemtype code="ConsentTemplate" jaloclass="de.hybris.platform.commerceservices.jalo.consent.ConsentTemplate" autocreate="true" genera
    <description>A type of consent associated with a particular store.</description>
    <deployment table="ConsentTemplates" typecode="6233" propertytable="ConsentTemplateProps" />
    <attributes>
        ...
        <attribute qualifier="exposed" type="boolean">
            <description>Indicates whether the consent template should be exposed to integrators in a storefront implementation as par
            <persistence type="property" />
            <defaultvalue>false</defaultvalue>
        </attribute>
        ...
</itemtype>
```

Anonymous Consent Data Storage

There are two cases for storing consent data:

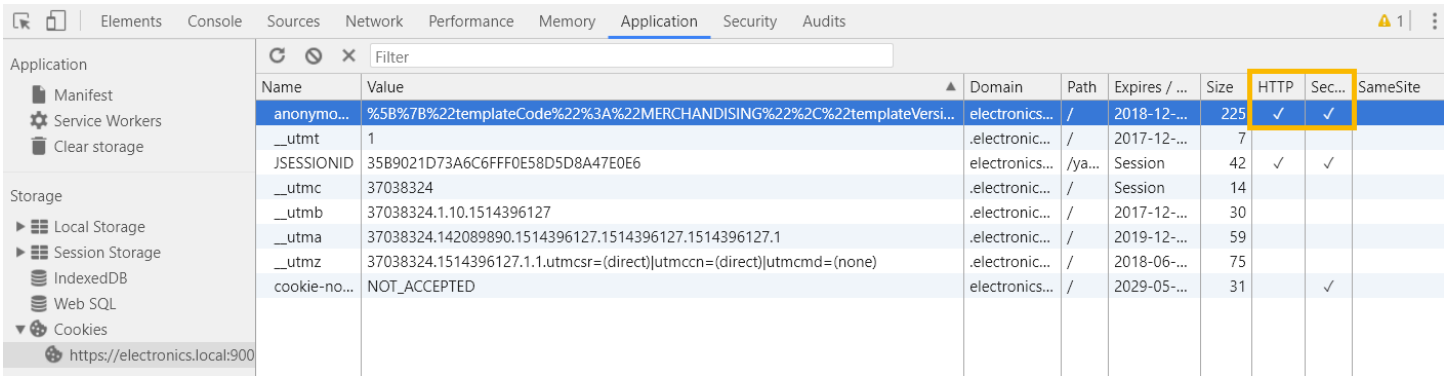
- **For anonymous users**: The consent data is stored in an `anonymous-consents` cookie in the storefront and in an `X-Anonymous-Consents` HTTP header in OCC.
- **For registered or logged in users**: The consent data is stored in a session and the cookie data is ignored.

ℹ Note

The consent data for the anonymous users is also replicated in the session. When the user logs in, the session attribute for the anonymous user is overridden by the consent data of logged in user.

Cookie

`anonymous-consents` cookie reflects different consents along with the user decision and the information that the users can interact with.



Anonymous consent cookies are tagged both with **Secure** and **HttpOnly** flag.

HTTP Header

`X-Anonymous-Consents` HTTP header is used in OCC REST APIs as the equivalent of `anonymous-consents` cookie. Cookie value and HTTP header value have the same format.

Session

After the user logs in or registers, the consent data is read only once from the back end and gets transferred to the session.

Consent Management Section

The **Consent Management** section displays user decisions on different user tracking services. It is located in **My Account** area and enables the logged in users to manage their consent choices.

For more information on how consent is managed in SAP Commerce Accelerator, see Consent Management.

Localization

The current language is stored in the session. If the user changes the language, the template is automatically removed from the session and a new one is loaded in the new language. See the following code snippet from `DefaultAnonymousConsentFacade.java`:

```
/**
 * Erase CONSENT_TEMPLATES from session on language change
 */
protected void checkLanguageChange()
{
    final String currentLanguage = storeSessionFacade.getCurrentLanguage().getIsocode();
    final String previousLanguage = sessionService.getAttribute(PREVIOUS_CONSENT_LANGUAGE);
    if (StringUtils.isEmpty(previousLanguage) || !currentLanguage.equals(previousLanguage))
    {
        sessionService.removeAttribute(CONSENT_TEMPLATES);
        sessionService.setAttribute(PREVIOUS_CONSENT_LANGUAGE, currentLanguage);
    }
}
```

Parent topic: Consent Management

Related Information

Giving Consent
Consent Logs
Consent Templates
Consent Template Versions
Adding Consent Entry Points
Personal Data Erasure
Main Use Cases

Main Use Cases

Explore the various use cases for anonymous consent management.

Anonymous User

Anonymous users can make choices in the form of consents.

Accelerator and OCC API

Each anonymous user consent template is exposed for every successive anonymous user request, until it has been responded to.

The described logic is implemented by the `synchronizeAnonymousConsents` method in the `DefaultAnonymousConsentFacade` class.

Moving from Anonymous User to Registered Customer

Anonymous users can make choices in the form of consents and then register to create a new account. In such scenario, their choices are carried over to the created account accordingly.

Accelerator

Choices selected by the user persist in the backend consent table and get displayed in the session. This is done only once during the registration process.

The described logic is implemented by the `processRegisterUserRequest` method in the `AbstractRegisterPageController` class.

OCC API

During the registration process, consents from the `X-Anonymous-Consents` HTTP header are carried over to the newly created user account. This also works when a guest user creates an account after checkout.

When an anonymous or guest user signs up, the register method from `DefaultCustomerAccountService` carries over the consents if the `POPULATING_CONSENTS_ENABLED` session flag is set to true.

Moving from Anonymous User to Logged In Customer

When an anonymous user logs into an existing account, any consent choices made before logging in are ignored.

Accelerator

When a registered user logs in, any anonymous user cookie data is ignored, and consent data from the registered customer session is used instead.

The `CustomerConsentDataStrategy` of the `onAuthenticationSuccess` method in the `StorefrontAuthenticationSuccessHandler` is used to populate consent data in session on login.

OCC API

The value of the `X-Anonymous-Consents` HTTP header is ignored when an anonymous user logs in as an existing user.

Moving from Logged In Customer to Anonymous User When Logging Out

When a customer logs out, any choices made anonymously take over.

Accelerator

After logging out, any previously used registered customer session information disappears, and the existing anonymous user cookie information takes over.

When logging out, consent data is removed from the session by the `onLogoutSuccess` method of the `StorefrontLogoutSuccessHandler`.

OCC API

This action is not applicable for OCC v2.

Personal Data Erasure

Customers can choose to close their account in the storefront and have their personal data deleted. Cron jobs included in the `yacceleratorcore` extension clean up data related to deactivated customers.

When customers decide to close their storefront account, they navigate to **My Account** > **Close Account** and click **Close Account**. The date when the account was closed is set as the `deactivationDate` for the customer. Once the account is closed, the customer cannot use that account to access the storefront, but can contact Customer Support to cancel the deactivation before the retention period expires (two days by default). When the retention period expires, cron jobs delete customer-related data, such as:

- email address
- contact information
- shipping details

- delivery preferences
- consent settings
- payment details

Order history is not deleted at the same time as customer-related data, as there is a separate retention period for order-related data (10 years by default). Customers can contact Customer Support within the order retention period if they need any information about their order history.

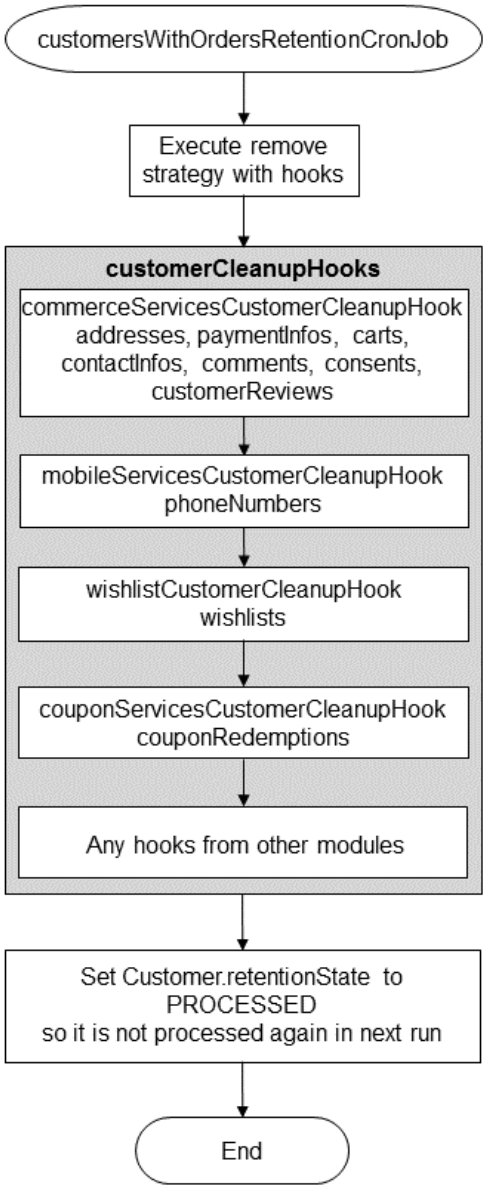
Cleanup Cron Jobs

Three erasure cron jobs are used to erase customers, orders, and the corresponding related data object:

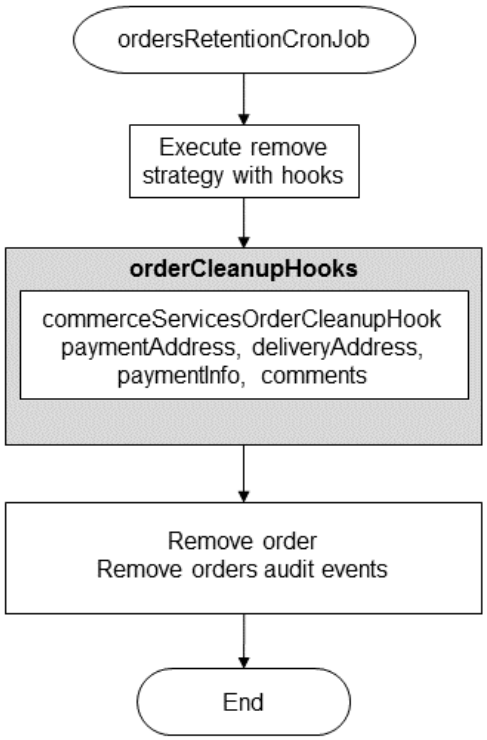
Cron Job	Description	retentionTime
`customersWithoutOrdersRetentionCronJob`	Cleans up customer-related data and deletes deactivated customers who don't own any orders. Finds all deactivated customers based on the following conditions: • Customer does not currently have orders • Current date is past the customer `deactivationDate` + `retentionTime`. Customers can cancel their deactivation before the retention time is reached.	2 days
`customersWithOrdersRetentionCronJob`	Cleans up customer-related data for deactivated customers who own orders. The `Customer.retentionState` is set to PROCESSED so that subsequent executions of this cron job do not process the same customer. Finds all deactivated customers based on the following conditions: • Customer currently has orders • Current date is past the customer `deactivationDate` + `retentionTime`. • Customer is not already processed by this cron job.	2 days
`ordersRetentionCronJob`	Cleans up orders and order-related data for deactivated customers. It only picks up orders for customers already processed by `customersWithOrdersRetentionCronJob`. Finds orders based on the following conditions: • Order expiration is more than 10 years • Customer is deactivated and `Customer.retentionState` is PROCESSED **ⓘ Note** The `ordersRetentionCronJob` does not take the customer's `retentionState` variable into consideration when deleting orders.	10 years

These cron jobs are defined in the `yacceleratorcore/resources/yacceleratorcore/import/common/cronjobs.impex` file of the `yacceleratorcore` extension. The definitions automatically trigger the cron jobs once a day. The following diagrams illustrate the workflow of the cron jobs.

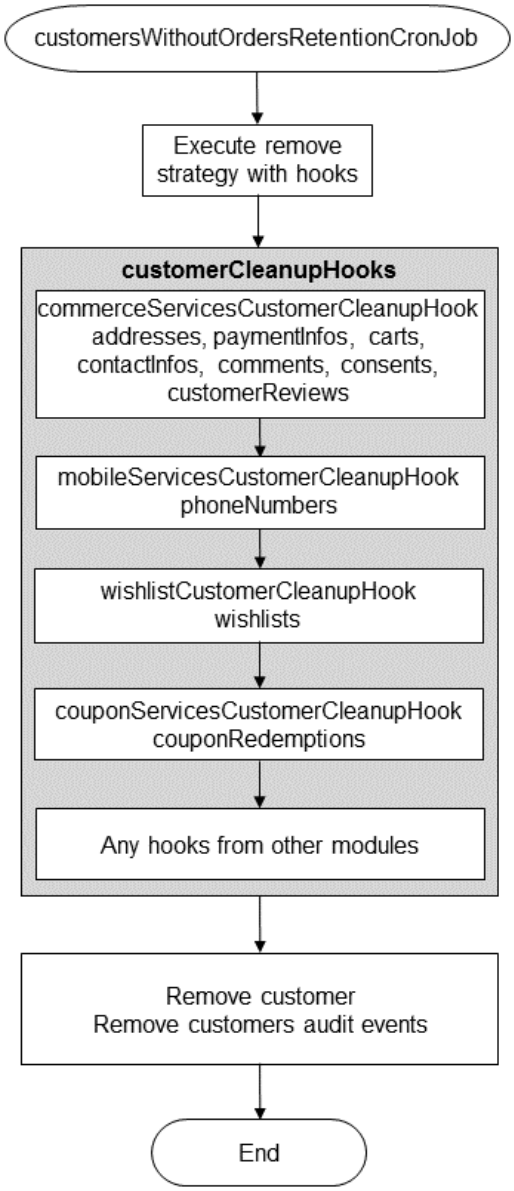
The first cron job to run is `customersWithOrdersRetentionCronJob`, which finds all deactivated customers past the customer retention period that have orders in their account and deletes customer-related data:



The second cron job to run is `ordersRetentionCronJob`, which finds all orders past the retention period and deletes order-related data:



The last cron job to run is `customersWithoutOrdersRetentionCronJob`, which finds all deactivated customers past the customer retention period without orders in their account and deletes customer-related data:



You can configure each of the cron jobs to reference `RetentionRule`. In addition, you can configure the `RetentionRule` to reference a Spring bean, which can be used to plug in custom logic to perform a specific task. The Platform data retention framework provides this mechanism. The Platform module provides `AbstractExtensibleRemoveCleanupAction` and `DefaultExtensibleRemoveCleanupAction`, which can be used to plug in to the `RetentionRule` to execute the logic that cleans up an item and its related data. The default implementation of `DefaultExtensibleRemoveCleanupAction` implements the `cleanup(final AfterRetentionCleanupJobPerformable retentionJob, final AbstractRetentionRuleModel rule, final ItemToCleanup item)` method. This method has a hooking mechanism that calls a list of hooks to clean up any data related to the item being cleaned up by the cron job. After all the hooks are executed, the item is removed and any audit records pertaining to the item being cleaned up are also removed.

Cleanup Actions

The erasure cron jobs reference three actions in the `yacceleratorcore` extension:

- `customerCleanupRelatedObjectsAction` is referenced by `customersWithOrdersRetentionCronJob`
- `customerNotOwningOrdersRemoveCleanupAction` is referenced by `customersWithoutOrdersRetentionCronJob`
- `orderRemoveCleanupAction` is referenced by `ordersRetentionJob`

`customerCleanupHooks` and `orderCleanupHooks` are injected to these actions accordingly.

Customer and Order Cleanup Hooking Mechanism

A hooking mechanism in the Platform module is available so that you can add hooks to clean up any customer or order-related data in your modules. The Platform module includes `orderCleanupHooks` and `customerCleanupHooks` as Spring util lists. To clean up any customer or order-related data, add your hooks to one of these lists. Note the following:

- Hooks added to `customerCleanupHooks` are invoked before the customer is removed or cleaned up.

- Hooks added to `orderCleanupHooks` are invoked before orders are removed or cleaned up.

If any audit records persist for the customer and order-related objects, the concrete hooks must remove the audit records. The Platform module provides an API to remove audit records. The API method to remove audit records should be called right after the related object is removed.

Here is a code snippet demonstrating removal of audit records:

```
WriteAuditRecordsDAO writeAuditRecordsDAO;


// remove audit records
final ComposedTypeModel type = getModelService().get(PK.parse(item.getItemType()));
writeAuditRecordsDAO().removeAuditRecordsForType(type.getCode(), item.getPk());
```

Adding a New Hook

To create a hook to clean up customer and order-related data, start by creating a Java class. We recommend that you put this class in the following package of your extension, `<your_base_package>.<extensionName>.retention.impl` package, and that you use the naming convention `<extensionName><itemTypeName>CleanupHook`. An example is this hook:

```
<your_base_package>.commerceservices.retention.impl.CommerceServicesCustomerCleanupHook
```

The hook should implement the `<your_base_package>.retention.hook package.ItemCleanupHook` interface. The concrete implementation of the `ItemCleanupHook` should implement the `cleanupRelatedObjects()` method, which has the logic to remove data objects related to the item supplied by the cron job. `DefaultExtensibleRemoveCleanupAction` invokes the `cleanupRelatedObjects()` method before the item is removed or cleaned up. If a custom action is implemented by extending `AbstractExtensibleRemoveCleanupAction`, that action should call the `cleanupRelatedObjects()` method so that it invokes the hooking mechanism. For example, this code shows how a hook cleans up customer-related objects in the `commerceservices` extension:

```
/*
 * [y] hybris Platform
 *
 * Copyright (c) 2000-2017 SAP SE
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of SAP
 * Hybris ("Confidential Information"). You shall not disclose such
 * Confidential Information and shall use it only in accordance with the
 * terms of the license agreement you entered into with SAP Hybris.
 */
package de.hybris.platform.commerceservices.retention.impl;

import static de.hybris.platform.servicelayer.util.ServicesUtil.validateParameterNotNullStandardMessage;

import de.hybris.platform.comments.model.CommentModel;
import de.hybris.platform.commerceservices.consent.dao.ConsentDao;
import de.hybris.platform.commerceservices.model.consent.ConsentModel;
import de.hybris.platform.core.model.order.CartModel;
import de.hybris.platform.core.model.order.payment.PaymentInfoModel;
import de.hybris.platform.core.model.user.AbstractContactInfoModel;
import de.hybris.platform.core.model.user.AddressModel;
import de.hybris.platform.core.model.user.CustomerModel;
import de.hybris.platform.customerreview.model.CustomerReviewModel;
import de.hybris.platform.retention.hook.ItemCleanupHook;
import de.hybris.platform.servicelayer.model.ModelService;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Required;


/**
 * This Hook removes customer related objects such as addresses, payment methods, carts and contact infos.
 *
 */
public class CommerceServicesCustomerCleanupHook implements ItemCleanupHook<CustomerModel>
{
    private static final Logger LOG = LoggerFactory.getLogger(CommerceServicesCustomerCleanupHook.class);

    private ModelService modelService;
    private ConsentDao consentDao;

    @Override
    public void cleanupRelatedObjects(final CustomerModel customerModel)
    {
        validateParameterNotNullStandardMessage("customerModel", customerModel);

        if (LOG.isDebugEnabled())
        {
            LOG.debug("Cleaning up customer related objects for: {}", customerModel);
        }

        // remove addresses
        for (final AddressModel address : customerModel.getAddresses())
        {
            getModelService().remove(address);
        }
        ......
    }
```

```
    protected ModelService getModelService()
    {
        return modelService;
    }

    @Required
    public void setModelService(final ModelService modelService)
    {
        this.modelService = modelService;
    }

    protected ConsentDao getConsentDao()
    {
        return consentDao;
    }

    @Required
    public void setConsentDao(final ConsentDao consentDao)
    {
        this.consentDao = consentDao;
    }

}
```

Finally, create a Spring bean for the hook class and add the bean to the `customerCleanupHooks` or `orderCleanupHooks` list. For example, this Spring configuration creates a Spring bean, `commerceServicesCustomerCleanupHook` and adds it to the `customerCleanupHooks` list.

```
<!-- Customer cleanup -->
<bean id="commerceServicesCustomerCleanupHook" class="de.hybris.platform.commerceservices.retention.impl.CommerceServicesCustomerClear
    <property name="modelService" ref="modelService"/>
    <property name="consentDao" ref="consentDao"/>
</bean>

<bean id="commerceServicesCustomerCleanupHookMergeDirective" depends-on="customerCleanupHooks" parent="listMergeDirective" >
    <property name="add" ref="commerceServicesCustomerCleanupHook" />
</bean>
```

Parent topic: Consent Management

Related Information

Giving Consent
Consent Logs
Consent Templates
Consent Template Versions
Adding Consent Entry Points
Anonymous Consent Management
Implementing Cleanup Logic
Data Retention Framework

Multiple Saved Carts

SAP Commerce Accelerator provides a saved cart feature that allows users to save one or more carts for later use.

Users can benefit from the saved cart feature to enhance their online shopping experience. For example, they can create a saved cart for items that they purchase on a regular basis, or create different saved carts for specific types of purchases.

Technical Overview

The saved cart feature is part of the Core Accelerator, so it is available in other responsive Accelerators, such as B2B. Two methods extend the saved cart feature included in the `commerceservices` extension to allow saving multiple carts:

Method	Description	Returns
`getSavedCartsCountForSiteAndUser(BaseSiteModel, UserModel)`	New method added to get the number of saved carts for a user.	Integer
`getSavedCartsForSiteAndUser(PageableData, BaseSiteModel, UserModel, List<OrderStatus>)`	Modified method used for pagination, and also includes option to sort the saved carts.	SearchPageData<CartModel>

For more details on the other methods used for saved carts, refer to the Save Cart section of commerceservices Extension.

By default, the saved cart feature is enabled in these storefronts:

- Apparel-UK (B2C)
- Apparel-DE (B2C)
- Electronics (B2C)

- Powertools (B2B)

To enable the saved cart in other storefronts, add the following code to your `local.properties` file:

```
# enabling accelerator save cart hook to remove the session cart & fetch a new cart for the user in electronics store
acceleratorservices.commercesavecart.sessioncart.hook.enabled.<storefront>=true

# enabling accelerator save cart restoration hook to set the saveTime as null before performing a restoration in electronics store
acceleratorservices.commercesavecart.restoration.savetime.hook.enabled.<storefront>=true
```

If no *<storefront>* is specified, then the saved cart is enabled for all storefronts.

Saved Cart Behavior

Users must be logged into the storefront to create saved carts. If they are not logged in, they are redirected to the login page when they want to save their cart.

Saved carts are created from the current active session cart (there can only be one active cart at a time). When a saved cart is created, the active cart is flagged as a saved cart, and a new active session cart is created.

When a saved cart is restored, it becomes the active cart. If there any items in the active cart before a saved cart is restored, the user has the option to create a separate saved cart with those items. Currently, there is no option to merge the saved and active carts.

User Experience

This section provides details on how users create, view, and restore saved carts in the storefront.

Creating Saved Carts

Users create saved carts on the checkout page. Clicking the **New Cart** link opens a dialog box where users enter a cart name and description, and then save the cart. Users must be logged in to save carts; if they are not logged in and click the **New Cart** link, they are redirected to the login page, and then returned to the checkout page once logged in.

If the cart includes any coupons or promotions applied to line items or the entire order, they are saved with the cart. When the saved cart is restored, all coupons and promotions are applied provided they have not expired when the cart was restored.

Viewing Saved Carts

Users can view all their saved carts in one of two ways:

- Clicking **My Account** > **Saved Carts** in the header menu
- Clicking the **You have *<X>* Saved Carts** link in the checkout page

Either method displays the saved cart list. The **Sort by** drop-down list allows users to sort the entries by:

- Date modified
- Date saved
- Cart ID
- Cart name
- Cart total

In the saved cart list, users can perform the following actions:

- Restore the saved cart by clicking **Restore**
- Delete the cart by clicking **X**
- View the cart details by clicking the cart name

The saved cart details page displays the all items and their respective quantities. Any promotions applied to line items are also displayed. In the details page, users can perform the following actions:

- **Edit** the saved cart name and description
- **Delete** the cart
- **Restore** the cart
- Go **Back to Saved Carts** to see the saved cart list

Restoring Saved Carts

When users click **Restore** in the saved cart list or details page, a pop-up window appears where users have the option of keeping a copy of the saved cart for future use. This is useful for recurring orders. Once a saved cart is restored, it becomes the active cart.

If there already were items in the active cart when a saved cart is restored, users also have the option to create a new, separate saved cart for those items as they are not merged with the restored cart.

If the saved cart has any coupons or promotions applied to line items, they are restored provided they have not expired at the time the cart is restored.

If an item in a saved cart has a quantity that is higher than the available stock at the time when the cart is restored, the quantity is adjusted to the available stock before completing the checkout.

Related Information

commerceservices Extension

Future Stock Availability

The Future Stock Availability feature indicates when inventory is going to be replenished with new stock.

For more information, see B2B Future Stock Availability.

Verification Token

This feature provides API for verification token code management. Client can request verification token code for specific purpose, and upon validation, the system delivers it to a designated channel. This code can further be utilized for verifying identities or transactions, thereby enhancing system security.

With this feature, for example, a login scenario could look like:

1. A customer enters the user name and password, then requests a verification token code.
2. The system validates the request and dispatches the verification token code to a designated channel, for example, the customer's email.
3. The customer receives the verification token code from the designated channel and enters it into the system along with their login request.

The entire process protects the customer's login against unauthorized users.

Introduction

Retrieve Verification Token

For a valid request to retrieve a verification token, the system will generate a record that contains the token id, token code, timestamp, and user information.

The token id serves to identify the client and is sent back to the client.

The token code, used for subsequent identification verification, is delivered by the system to a predetermined channel on an asynchronous basis.

Should the request to create a verification token not meet security standards, the system still assigns a token id to the client. However, the system will not generate or dispatch a token code.

Token Validation

If the supplied token id and token code match and have not yet expired, the system validates it successfully and the verification token is instantly removed. Otherwise, the validation is failed.

When login with verification token feature (`otp.customer.login.enabled`) is switched on, the spring security framework has been optimized to accept the token id and token code as user credentials.

- For OAuth2: Customers can obtain an OAuth2 token by applying the token id as the username and the token code as the password. If token validation is successful, the OAuth2 token is issued.
- For Accelerator, both the token id and the token code are necessary for authentication. Successful token validation leads to the establishment of an authenticated session.

Configuration

This table lists all the configurable properties related to verification token. Restart the server to apply the changes.

Property Name	Sample Value	Description
`otp.customer.login.enabled`	false	Specifies if the One-Time Password (OTP)
`otp.customer.login.token.code.encoder`	pbkdf2	Specifies the token encoder to be used for See the available options in `passwordEn` `de.hybris.platform.persistence`

Property Name	Sample Value	Description
`otp.customer.login.token.code.length`	8	When OTP is enabled, it allows to specify t functionality. The minimum of allowed characters is 6 ar
`otp.customer.login.token.code.alphabet`	1234567890ABCDEFGHKLMNPRSTUWXYZabcdefghijklmnoprstuwxyz	When OTP is enabled, it allows to set a spe OTP login functionality will be generated. The minimal charset length should be 10 c default charset value 1234567890ABCDEF is used.
`otp.customer.login.token.ttlseconds`	300	When OTP is enabled, it allows to specify t functionality. The minimum allowed value is 60 seconds (15 minutes).
`otp.customer.login.token.max.verification.attempts`	3	Specifies the maximum number of failed v the token is removed from the system. Vali
`otp.customer.login.token.id.generator.bits`	256	Number of bits used to generate the randc functionality. The minimum allowed value is 128 bits anc recommended to use at least 256 bits.
`otp.customer.login.token.purpose.short.name`	LGN	This short name is used to quickly identify functionality. The token id format is *<{shortName}[{ran* The short name should be unique for initia characters) as it does not add any security If the short name exceeds 10 characters, it name.

Compatibility

Learn which versions of SAP Commerce Cloud accelerators, integrations, and other components are supported with SAP Commerce Cloud in the public cloud.

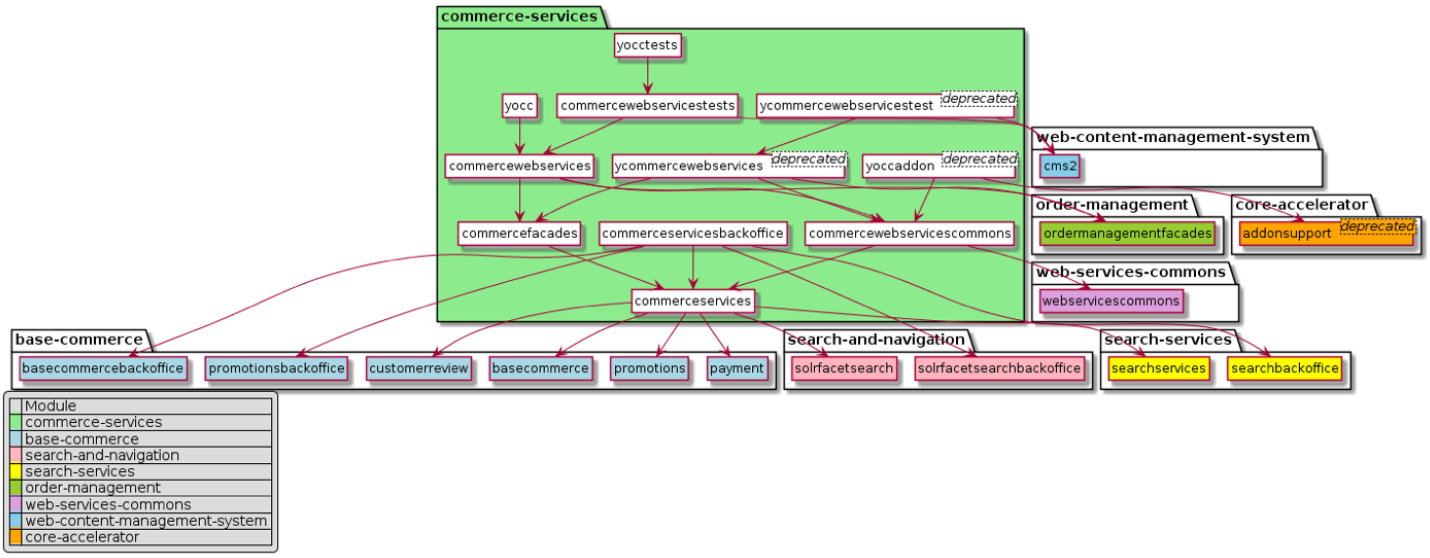
The following modules are currently supported for use of Login with Verification Token feature with SAP Commerce Cloud in the public cloud:

- B2C Accelerator AddOns Module. For more information, see Login with Verification Token in B2C Accelerator.
- B2B Accelerator AddOns Module. For more information, see Login with Verification Token in B2B Accelerator.
- China Accelerator People Profile Module. For more information, see Login with Verification Token in Accelerator for China.

Commerce Services Architecture

Commerce Services includes a set of extensions that affect your product configuration, multiple carts, and client consent management.

Dependencies



Recipes

For a complete list of SAP Commerce recipes that may include this module, see Installer Recipes.

For a complete list of the SAP Commerce Cloud, integration extension pack recipes that may include this module, see Installer Recipe Reference.

Extensions

The Commerce Services module consists of the following extensions:

- commerceservices Extension
- commercefacades Extension
- commercewebservicescommons Extension
- ycommercewebservicestest Extension
- ycommercewebservices Extension
- yoccaddon Extension
- commercewebservices Extension
- commercewebservicestests Extension
- yocc Extension
- yocctests Extension
- commerceservicesbackoffice Extension

commerceservices Extension

The `commerceservices` extension organizes functionality from one or more Platform services. These services fulfill certain tasks on their own, but often need to be combined to provide a complete B2C commerce use case. This often involves combining functionality from separate extensions that are licensed separately. The `commerceservices` extension also extends more generic functionality from certain SAP Commerce extensions to add more B2C commerce features.

ⓘ Note

An SAP Commerce extension may provide functionality that is licensed through different SAP Commerce modules. Make sure to limit your implementation to the features defined in your contract license. In case of doubt, please contact your sales representative.

The following sections are a guide to all the services and service enhancements provided with the `commerceservices` extension. Dependencies on extensions outside of the core platform are highlighted for your reference.

Catalog Services

The following sections describe the catalog services of the `commerceservices` extension.

Commerce Category Service

The **`CommerceCategoryService`** adds category lookup functionality within the scope of only the current session product catalogs. Other types of session catalogs are excluded, such as content catalogs and classification catalogs.

Commerce Price Service

The **`CommercePriceService`** is intended to replace the use of the existing **`PriceService`**. This provides a reduced interface to only expose typical B2C product price requests, such as getting the current web price or a from price, in the event of variants. All the additional **`PriceService`** complexities, such as price banding, and deciding which price is appropriate, is either hidden away or simply not supported by this service.

For more information, see basecommerce Extension.

ⓘ Note

The default implementation adds a dependency to the **basecommerce** extension.

Net Gross Strategy

The `commerceservices` extension extends **basecommerce** extension **`BaseStore`** to include a net flag attribute that identifies whether prices on a storefront should be net or gross of sales tax. A **`NetGrossStrategy`** is used by all services wanting to determine if net or gross prices should be returned. The **`NetPriceService`**,which is the **`commerceservices`** extension extended version of **`PriceService`** uses this strategy. There are two implementations out of the box: the **`DefaultNetGrossStrategy,`** which uses the default **`PriceFactory`** rules, and the **`CommerceNetGrossStrategy,`** which uses the value from the **`BaseStore`**.

Sales Tax

The `commerceservices` extension extends the **basecommerce** extension **`BaseStore`** to allow a **`UserTaxGroup`** to be specified. This can be used to set the tax group to ensure correct sales tax is picked up by the **`PriceService`** when sites are deployed across multiple countries.

External Tax

The `commerceservices` extension integrates the **`External Tax`** functionality. The **`ExternalTaxesService`** is used during checkout by the **`CommerceCheckoutService`**. External tax calculation consists of the overarching responsible for deciding whether a tax calculation call is necessary and, if so, for calling the calculation. The out-of-the box **`DefaultExternalTaxesService`** uses the **`RecalculateExternalTaxesStrategy`** to determine whether an external tax calculation is required. If it is deemed that a new calculation is required, the **`CalculateExternalTaxesStrategy`** is called to calculate tax. In order to determine the tax code of a product, the **`TaxCodeStrategy`** uses the **`TaxAreaLookupStrategy`** to call the **`ProductTaxCodeService`** and get the tax code.

Commerce Product Service

The **`CommerceProductService`** adds classification class filter functionality. This service previously provided stock support, which is now deprecated in favor of the Commerce Stock Service (described below).

For more information, see basecommerce Extension.

> **Note**
> The default implementation adds an implicit dependency to the **basecommerce** extension.

Facet Search Enhancements

A number of B2C commerce functions have been added to extend the functionality provided in the **solrfacetsearch** extension in the **commerceservices** extension.

For more information, see solrfacetsearch Extension and Search and Navigation.

> **Note**
> The `commerceservices` extension has a dependency on the **solrfacetsearch** extension.

Commerce Payment Services

The following sections describe the commerce payment services of the `commerceservices` extension.

Payment Merchant Transaction Code

To authorize payments or create subscriptions, PSPs require a merchant transaction code. The commerce layer provides the **`GenerateMerchantTransactionCodeStrategy`**. The default implementation uses the `userId` of the cart suffixed with a random UUID.

Fraud Handling

To create a payment transaction entry for a review decision of the submit-order process, you need a proper payment transaction type. This is provided by the **`REVIEW_DECISION`** value contained in the **`PaymentTransactionType`**.

Order Services

The following sections describe the order services of the **commerceservices** extension.

Commerce Cart Service

The **`CommerceCartService`** replaces the **`CartService`** by adding promotions calculation, stock checks, and stale cart cleanup to the cart services.

For more information, see basecommerce Extension and Extending CommerceCartService.

> **Note**
> The default implementation adds an implicit dependency to the **basecommerce** and **promotions** extensions.

Commerce Stock Service

The **`CommerceStockService`** is intended to be used in place of the **`StockService`** and provides methods for determining stock levels for a **`BaseStore`** or a **`PointOfService`**. This is essential to the "buy online pickup in store" funtionality. The **`CommerceServices`** add additional relationships from `Warehouse` to `PointOfService,` in addition to the existing relationships from `Warehouse` to `BaseStore,` and `PointOfService` to `BaseStore`.

For more information, see Stock Service - Technical Description and basecommerce Extension.

> **Note**
> The default implementation adds an implicit dependency to the **basecommerce** extension.

Commerce Checkout Service

The **`CommerceCheckoutService`** adds services to orchestrate typical operations during checkout, such as registering a new credit card, adding and setting a delivery address, choosing a delivery method, and placing the order. It is intended to replace direct use of the **`CartService`** and **`OrderService`**.

For more information, see the following:

- basecommerce Extension
- cms2 Extension - Technical Guide
- cms2 Extension Tutorial
- payment Extension
- Extending CommerceCheckoutService

> **Note**
> The default implementation adds an implicit dependency to the **basecommerce**, **promotions** and **cms2** extensions. The interface also has an explicit dependency on the **payment** extension.

Delivery Service

The **`DeliveryService`** provides functionality around available and supported delivery rules that are scoped to a single store front, rather than an entire platform deployment, which may include multiple store fronts. This includes identifying supported delivery countries and delivery options.

Customer Account Services

The following sections describe the customer account services of the `commerceservices` extension.

Customer Account Service

The customer account service handles typical customer account management capabilities. It provides methods for registering, verifying user credentials, updating profile settings, forgotten password services, managing address books, viewing a user's orders for the current store front, and managing payment information using a PCI compliance-friendly subscription interface.

For more information, see the following:

- basecommerce Extension
- cms2 Extension - Technical Guide
- cms2 Extension Tutorial
- payment Extension
- Authentication with Credit Card by Saved Token

> **Note**
> The default implementation adds an implicit dependency to the **cms2** extension. Furthermore, the interface adds an explicit dependency to the **basecommerce** and **payment** extensions.

Customer Email Resolution Service

The customer email resolution service adds a layer of abstraction on top of the **`CustomerModel`** to resolve the customer's email address.

Secure Token Service

The secure token service is used to encrypt **`SecureToken`** data into a short encrypted token and to decrypt a token.

The **`SecureToken`** data consists of a data string and a timestamp. The resultant encrypted string of the token is in Base64 format.

The **`SecureTokenService`** is used in forgotten password request process. When a customer requests for a forgotten password, a token is generated, using the service, by passing **`SecureToken`** data that consists of customer uid and a timestamp. The generated token is maintained in the system against the customer and an email is sent to the customer with a password reset URL incorporating the token. When the customer tries to reset the password, by accessing the password reset page using the password reset URL, the token is decrypted and validated. It is also verified with the token maintained in the system against the customer before resetting the customer's password.

DefaultSecureTokenService

The **`DefaultSecureTokenService`** implements the **`SecureTokenService`**.

The data to be encrypted is signed and encrypted using two separate keys. The keys are specified as hex character strings. Unlimited strength cryptography is required.

The data assembled to encrypt includes the **`Signature`** key (not included in the data block), and the data block, which itself consists of the data string in the `SecureToken`, the generated checksum for the data string in the **`SecureToken,`** and the timestamp in the **`SecureToken`**. The data block is also prefixed and postfixed with padding lengths. All of the above, except the signature key, are AES encrypted using the encryption key. The result is in Base64 format.

Store Locator Services

The following sections describe the store locator services of the `commerceservices` extension.

The Store Finder Service

The **`StoreFinderService`** replaces the now deprecated **`StoreLocatorService`**. The **`StoreFinderService`** orchestrates the various services provided by the hybris Store Locator in the **basecommerce** extension to deliver typical store locator page functionality. This includes finding the nearest store, searching by town or postal code, maps for locations in proximity to the current location, and content required for a store information page.

Additional Brick and Mortar store attributes are added to the **`PointOfService`** to capture content such as the store image, store features, and some marketing content for a store details page. For more information, see Store Locator Configuration.

The `StoreFinderService` includes the following methods:

- **`locationSearch`** methods, which find nearby Points of Service (POS) for the given `BaseStore` and location text.
- **`positionSearch`** methods, wich find nearby POS for the given `BaseStore` and **`GeoPoint`**, which is an object containing latitude and longitude.
- **`getPointOfService*ForName`** methods, which return the named POS for the given **`BaseStore`**, with or without a distance calculation.
- `getAllPoS`, which is a method to get all the POS for the given **`BaseStore`**.

i Note

The interface has an explicit dependency on the **basecommerce** extension.

The **`DefaultStoreFinderService`** uses the **`GeoWebServiceWrapper`** to geocode and determine the direction. For more information, see GeoWebServiceWrapper.

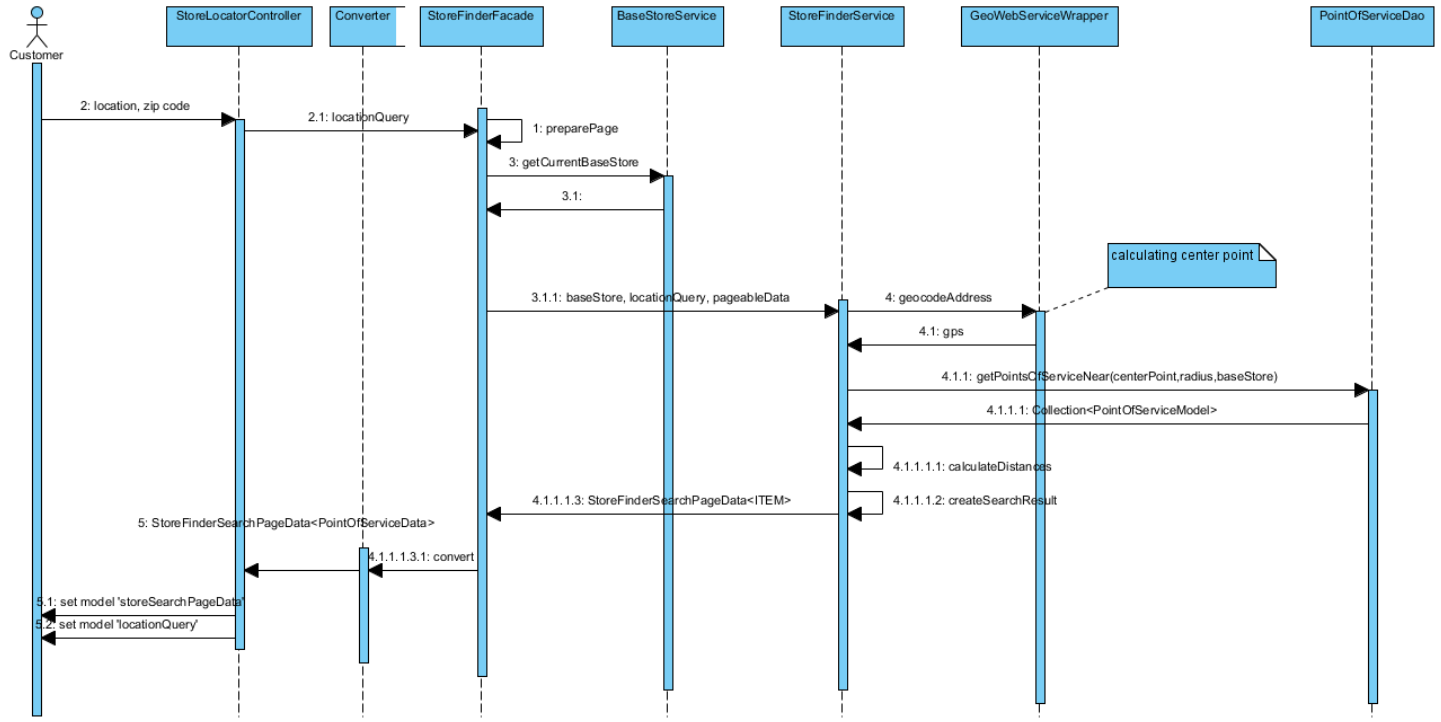
⚠ Caution

Mock Geocoding

The default **`DefaultStoreFinderService`** uses a of **`GeoWebServiceWrapper`** for the location information. The **`CommerceMockGeoWebServiceWrapper`** returns fixed geocoding points based on the country.

Specific Store Locator Use Cases

For most situations, the hybris `commerceservices` extension uses the default implementation of the Store Locator feature from the **basecommerce** extension. However, due to migration to Google Maps API v.3, the use cases diagram looks as follows:



I18n Services

The following section describes the I18n services of the **commerceservices** extension.

Commerce Common I18n Service

The **`CommerceCommonI18nService`** complements the hybris Internationalization and Localization capabilities by providing storefront-scoped services for obtaining default or supported languages, currencies, supported territories and operating locale to the current executing storefront.

For more information, see cms2 Extension - Technical Guide.

ℹ Note

The interface has an explicit dependency on the **cms2** extension.

Promotion Services

The following section describes the commerce promotion service of the `commerceservices` extension.

Commerce Promotion Service

The **`CommercePromotionService`** provides methods for getting information about defined promotions.

Merge Directive Services

The `ListMergeDirective` and the `MapMergeDirective` allow for greater control over Spring bean lists and maps.

ℹ Note

When the `ListMergeDirective` and `MapMergeDirective` were moved from `commerceservices` to platform core, this change included the following alterations:

- The package for classes chaged from `de.hybris.platform.commerceservices.spring.config` to `de.hybris.platform.spring.config`.
- The classes in `de.hybris.platform.commerceservices.spring.config` are now deprecated, so use the `de.hybris.platform.spring.config` package instead.
- All code which used `listMergeDirective` or `mapMergeDirective` will work normally. If `listMergeBeanPostProcessor` or `mapMergeBeanPostProcessor` was declared in your context, update the defintion to use classes from the platform core.

ListMerge

The `ListMergeDirective` allows an AddOn to merge additional elements into Spring bean lists and list properties on Spring beans. The minimal property to set is the add. The directive also supports the ability to insert the bean before or after a specified list element bean definition or bean class. `ListMergeDirective` beandefinitions must also include a `depends - on` qualifier, which should be the listbean or the bean enclosing the list property, as follows:

```
<bean id="listMergeBeanPostProcessor" class="de.hybris.platform.spring.config.ListMergeDirectiveBeanPostProcessor" />
<bean id="listMergeDirective"  class="de.hybris.platform.spring.config.ListMergeDirective" abstract="true" />

<!-- This snippet adds a bean to end of the target list -->
<bean id="addToListMergeDirective" depends-on="targetListBean" parent="listMergeDirective">
        <property name="add" ref="MyBeanToInsertAtEndOfList" />
</bean>

<!-- This snippet inserts a bean after all beans of a given type and before all beans of a different type -->
<bean id="insertBeforeAfterListMergeDirective" depends-on="targetListBean" parent="listMergeDirective">
        <property name="add" ref="myBeanToInsert" />
        <property name="afterClasses">
                <list value-type="java.lang.Class">
                        <value>full.package.of.class.ToInsertAfter</value>
                </list>
        </property>
        <property name="beforeClasses">
                <list value-type="java.lang.Class">
                        <value>full.package.of.class.ToInsertBefore</value>
                </list>
        </property>
</bean>
```

Beans can also be inserted before or after other beans by referencing them by name, as follows:

```
<bean id="insertAfterBeanListMergeDirective" depends-on="targetListBean" parent="listMergeDirective">
        <property name="add" ref="myBeanToInsert" />
        <property name="afterBeanNames">
                <list value-type="java.lang.String">
                        <value>beanToInsertAfter</value>
                </list>
        </property>
</bean>
```

A specific list can be targeted for insertion, either by property access or field access, as follows:

```
<bean id="myBeanWithListProperty" class="BeanWithList">
        <property name="myListOfBeans">
                <list>
                        <ref bean="oneBean">
                        <ref bean="twoBean">
```

```
                        <ref bean="redBean">
                        <ref bean="blueBean">
                </list>
        </property>
</bean>

<bean id="addBeanToListPropertyListMergeDirective" depends-on="myBeanWithListProperty" parent="listMergeDirective">
        <property name="add" ref="myBeanToAdd" />
        <property name="listPropertyDescriptor" value="myListOfBeans" />
</bean>
```

Field access is best used when the list being modified does not have a public getter, as follows:

```
<bean id="addBeanToListFieldListMergeDirective" depends-on="myBeanWithNoPublicGetterList" parent="listMergeDirective">
        <property name="add" ref="myBeanToAdd" />
        <property name="fieldName" value="myNoPublicGetterListOfBeans" />
</bean>
```

MapMerge

The `MapMergeDirective` is very similar to the `ListMergeDirective`, as can be seen in the following example:

```
<bean id="mapMergeBeanPostProcessor" class="de.hybris.platform.spring.config.MapMergeDirectiveBeanPostProcessor" />
<bean id="mapMergeDirective"  class="de.hybris.platform.spring.config.MapMergeDirective" abstract="true" />

<bean id="myMapMergeDirective" depends-on="myBeanWithMap" parent="mapMergeDirective">
        <property name="key" value="myMapKey" />
        <property name="value" value="myMapValue" />
</bean>
```

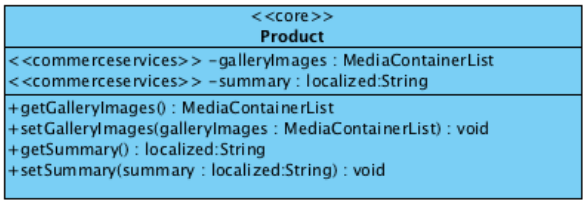
Similarly, properties for accessing maps can be used with either `fieldName` or `mapPropertyDescriptor`.

Data Model

The `commerceservices` extension adds a number of data model modifications to add further B2C commerce support to existing, more generic extensions. This extension can also combine data models from multiple extensions to enable additional B2C commerce functions.

Product

The following sections describe the `Product` class.



A marketing purpose **`summary`** attribute has been added to the product to complement the existing name and large description attribute. This can be used when a more concise product description is required.

The **`galleryImages`** attribute is used to store multiple images, each resized to a number of standard formats expected by the storefront. A list of **`MediaContainer`** objects are used to model the gallery.

Using Media Containers for a Gallery

Media containers are a useful way of grouping related images. The `commerceservices` extension updates the **`Product`** data model to support a gallery, which enables you to store multiple groups of images. Each image group uses the same image, which is resized up-front to dimensions that are expected to fit on the storefront, such as thumbnail, detail, or zoom. This enables the storefront to serve a link to an image that is correctly size-optimized, and does not require any browser or client-side resizing. This also ensures that bandwidth is not unnecessarily wasted when downloading images to the client browser. For galleries to work well and look good, each image should be resized to a set of standard dimensions and formats.

hybris uses Medias to store image information such as its location on the web server, and a Media has an attribute of type Media Format that can be used for the purpose of logically defining the dimensions of an image.

For example, a "thumbnail" format could be 60x40px and a "zoom" format could be 1000x700px.

A Media Container's primary function is to group related Media with some extra context information about the related Media.

The Gallery Images attribute is modeled as a List of Media Containers. If we need to group all the related media that differ only by format (for example, by different dimensions) you can use Media Containers. Media Containers are catalog version-aware as well, which means changes can be uploaded to a Staged Product Catalog and then published online.

For more information, see Handle Medias Using the MediaContainer and Managing Media in the hybris Product Cockpit.

Customer Reviews

The **customerreview** extension data model has been extended to support an approval process and additional functionality.

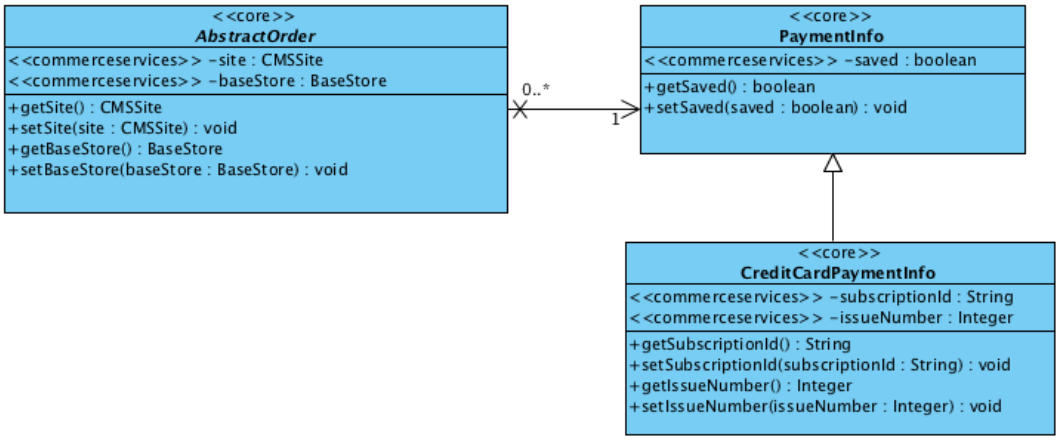
For more information, see Customer Reviews.

Order

The following sections describe the `Order` class.

Tracking the Site in Which the Order Was Placed

The `AbstractOrder` data model is modified to add **`Site(BaseSite)`** and **`Store(BaseStore)`**. The **`CommerceCartFactory`** deals with setting up the cart with the current `BaseSite` and `BaseStore`. The normal clone order process moves these settings over to the created `Order`.



In a multiple storefront scenario, this can determine in which storefront the order was placed, for reporting as well as filtering purposes. In the CsCockpit, the agent makes a selection regarding which `BaseSite` to work within, restricting products and prices accordingly.

Buy Online Pick Up In Store Support

A `PointofService` is now associated with the `AbstractOrderEntry`, and can be used to record orders for collection for a point of service. This is also added onto consignment to support order splitting.

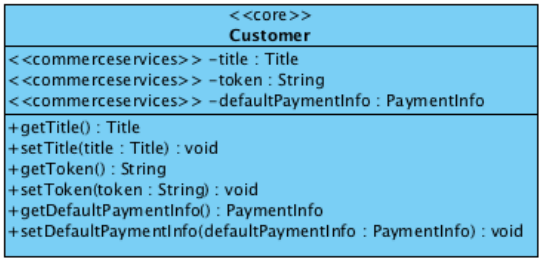
Payment

It is possible to flag whether **`PaymentInfo`** should be used as a saved payment for a customer. An issue number also exists to support a subset of Switch and Maestro credit cards.

For more information, see payment Extension.

Customer

This section describes the **`Customer`** class.

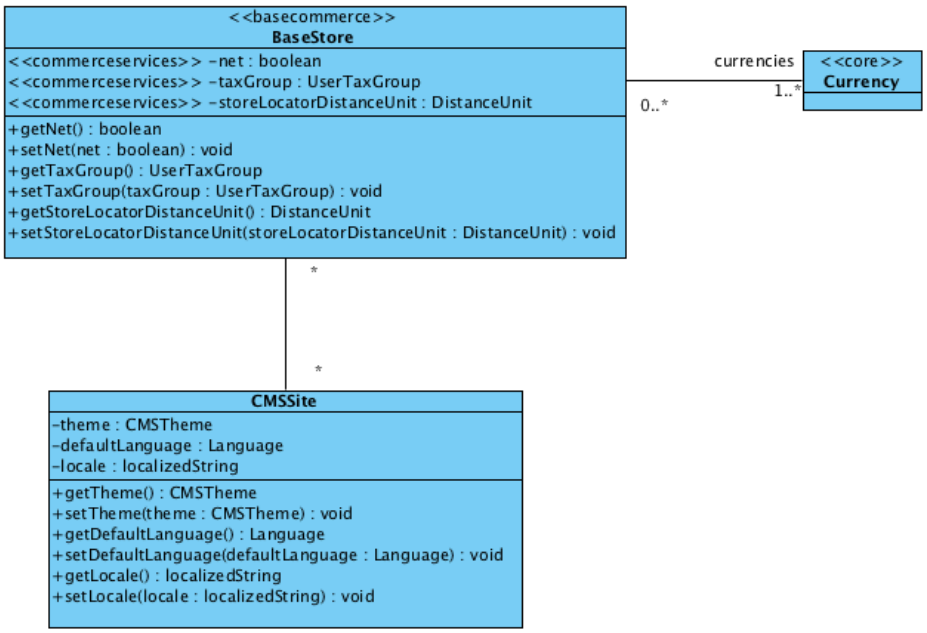


The `Customer` class has the following features:

- The class includes a title attribute.
- A customer's default payment information can be stored against the customer.
- A special token attribute supports the ability to authenticate certain Customer Account requests that would need the same token to be passed back from the client as a request parameter. The forgotten password functionality, described in the Customer Account Service section above, uses this to store a secure token that can only be validated once.

Storefront Setup

A number of additional attributes have been added to **`BaseStore`** and **`CMSSite`** to capture configuration settings for each storefront.



BaseStore Financial Settings

The `BaseStore` financial settings include the following:

- A net flag exists to identify if the store prices are calculated net or gross of sales tax. This is used by the price calculation logic.
- The list of currencies supported for the store are captured against the store.
- The user tax group from the **europe1** extension is stored to enable the price calculation logic to pick appropriate store-specific sales tax rates in a multi-storefront setup.

CMSSite Localization Settings

The `CMSSite` localization settings include the following:

- The default language of the storefront is configurable. This is especially relevant if content on a specific storefront comes in multiple languages. The total list of languages available comes from the active content catalog version.
- The Java Locale to use for each language is configured to ensure numbers and currency information are formatted appropriately on the storefront.

CMSSite Presentation Settings

A theme attribute exists that allows the look and feel of the Storefront to be switched.

For more information, see Storefront and Catalog Modeling.

Solr Facet Search

A number of additions to the data model have been made to support a more enriched B2C SOLR configuration out-of-the box.

For more information, see Search and Navigation.

Store Locator

The Store Locator data model has been extended to include more attributes specific to Brick And Mortar stores.

For more information, see Store Locator Configuration.

Configuration of SetupSyncJobService

It is possible to configure **`de.hybris.platform.commerceservices.setup.impl.DefaultSetupSyncJobService`** using Spring configuration. You can configure `rootTypes` for both `ProductCatalogSyncJob` and `ContentCatalogSyncJob`, as well as attributes of **`SyncAttributeDescriptorConfig`** for given items.

The following attributes are supported using **`EditSyncAttributeDescriptorData`**:

- **`includeInSync`**: if set to `false`, the attribute is excluded from the sync
- **`copyByValue`**: if set to `true`, the attribute is copied by value
- **`untranslatable`**: if set to `true`, the attribute is marked as not translatable for sync

The following example shows the use of these attributes:

```
<entry key="de.hybris.platform.category.jalo.Category">
                        <list>
                                <bean class="de.hybris.platform.commerceservices.setup.data.EditSyncAttributeDescriptorData">
                                        <property name="qualifier" value="linkcomponents"/>
                                        <property name="includeInSync" value="false"/>
                                        <property name="copyByValue" value="true"/>
                                        <property name="untranslatable" value="false"/>
                                </bean>
                        </list>
                </entry>
```

The following is a snippet of configuration of the `defaultSetupSyncJobService` bean from the `commerceservices -spring.xml` file:

```
<util:list id="contentCatalogSyncRootTypeCodes">
           <value>CMSItem</value>
           <value>CMSRelation</value>
           <value>Media</value>
           <value>MediaContainer</value>
           <value>BTGItem</value>
        </util:list>
        <util:list id="productCatalogSyncRootTypeCodes"/>
        <util:map id="contentCatalogEditSyncDescriptors" key-type="java.lang.Class" value-type="java.util.List"/>
        <util:map id="productCatalogEditSyncDescriptors" key-type="java.lang.Class" value-type="java.util.List">
                <entry key="de.hybris.platform.jalo.Item">
                        <list>
                                <bean class="de.hybris.platform.commerceservices.setup.data.EditSyncAttributeDescriptorData">
                                        <property name="includeInSync" value="false"/>
                                        <property name="qualifier" value="comments"/>
                                </bean>
                                ...
                        </list>
                </entry>
                <entry key="de.hybris.platform.jalo.product.Product">
                        <list>
                                <bean class="de.hybris.platform.commerceservices.setup.data.EditSyncAttributeDescriptorData">
                                        <property name="includeInSync" value="false"/>
                                        <property name="qualifier" value="sequenceId"/>
                                </bean>
                                <bean class="de.hybris.platform.commerceservices.setup.data.EditSyncAttributeDescriptorData">
                                        <property name="includeInSync" value="false"/>
                                        <property name="qualifier" value="productReviews"/>
                                </bean>
                                ...
                        </list>
                </entry>
                <entry key="de.hybris.platform.europe1.jalo.PriceRow">
                        <list>
                                <bean class="de.hybris.platform.commerceservices.setup.data.EditSyncAttributeDescriptorData">
                                        <property name="includeInSync" value="false"/>
                                        <property name="qualifier" value="sequenceId"/>
                                </bean>
                        </list>
                </entry>
                <entry key="de.hybris.platform.category.jalo.Category">
                        <list>
                                <bean class="de.hybris.platform.commerceservices.setup.data.EditSyncAttributeDescriptorData">
                                        <property name="includeInSync" value="false"/>
                                        <property name="qualifier" value="linkcomponents"/>
                                </bean>
                                ...
                                <bean class="de.hybris.platform.commerceservices.setup.data.EditSyncAttributeDescriptorData">
                                        <property name="includeInSync" value="false"/>
                                        <property name="qualifier" value="categoryFeatureComponents"/>
                                </bean>
                        </list>
                </entry>
        </util:map>

        <alias name="defaultSetupSyncJobService" alias="setupSyncJobService" />
        <bean id="defaultSetupSyncJobService"
              class="de.hybris.platform.commerceservices.setup.impl.DefaultSetupSyncJobService">
                <property name="modelService" ref="modelService"/>
                <property name="contentCatalogRootTypeCodes" ref="contentCatalogSyncRootTypeCodes"/>
                <property name="productCatalogRootTypeCodes" ref="productCatalogSyncRootTypeCodes"/>
                <property name="contentCatalogEditSyncDescriptors" ref="contentCatalogEditSyncDescriptors"/>
                <property name="productCatalogEditSyncDescriptors" ref="productCatalogEditSyncDescriptors"/>
        </bean>
```

Save Cart

The save cart functionality primarily allows a customer to save and restore their saved carts at a later date. This functionality is provided as a set of methods embedded within independent strategies that can be wired-up, relative to the different business requirements and the front-end implementations they are used for. These strategies can be easily extended with pre/post-hooks.

The following save cart operations are currently supported:

- Save a session cart as a saved cart
- Save specific cart IDs, for a back-end operations, as saved carts
- Display a list of saved carts
- Display the details of a saved cart
- Restore a saved cart to an active session cart

- Delete saved carts
- Clone saved carts
- Edit the name and description of a saved cart

In order to support this functionality, a new **CommerceSaveCartService** interfacehas been implemented. This interface contains the following methods:

Method Signature	Returns
saveCart(CommerceSaveCartParameterData)	**CommerceSaveCartResultData**
flagForDeletion(CommerceSaveCartParameter)	**CommerceSaveCartResultData**
getCartForCodeAndCurrentUser(CommerceSaveCartParameterData)	**CartData**
restoreSavedCart(CommerceSaveCartParameterData)	**CartRestorationData**
getSavedCartsForCurrentUser(PageableData , List<OrderStatus>)	**SearchPageData<CartData>**
cloneSavedCart(CommerceSaveCartParameterData)	**CommerceSaveCartResultData**

Also, a new **SaveCartDao** interface has been introduced. This interface contains the following methods:

Method Signature	Returns
getSavedCartsForRemovalForSite(BaseSiteModel)	**List<CartModel>**
getSavedCartsForSiteAndUser(PageableData , BaseSiteModel , UserModel , List<OrderStatus>)	**SearchPageData<CartModel>**

Apart from implementing the **SaveCartDao**, your custom implementation of this interface has to extend an implementation of the **CommerceCartDao** interface in order to support operations on **Cart** objects. In other words, if your implementations of the **CommerceCartDao** and **SaveCartDao** interfaces are named **DefaultCommerceCartDao** and **DefaultSaveCartDao** respectively, the class header looks as follows:

DefaultSaveCartDao

```
public class DefaultSaveCartDao extends DefaultCommerceCartDao implements SaveCartDao
```

SaveCartStrategy

The default class extending the **AbstractCommerceSaveCartStrategy** class, the **DefaultCommerceSaveCartStrategy** class, handles the save cart functionality for bundle products. This class provides a **calculateExpirationTime()** method which calculates the time in which a saved cart expires based on the save date and the **commerceservices.saveCart.expiryTime.days** property. You can change the default expiry time in the **`project.properties`** file:

project.properties

```
#Defines after how many days a saved cart expires
commerceservices.saveCart.expiryTime.days=30
```

FlagForDeletionStrategy

The default class extending the **AbstractFlagForDeletionStrategy** class, the **DefaultCommerceFlagForDeletionStrategy** class, provides the **flagForDeletion()** method which sets all the saved cart's attributes to null:

DefaultCommerceFlagForDeletionStrategy

```
final CartModel cartToBeFlagged = parameters.getCart();

                cartToBeFlagged.setExpirationTime(null);
                cartToBeFlagged.setSaveTime(null);
                cartToBeFlagged.setSavedBy(null);
                cartToBeFlagged.setName(null);
                cartToBeFlagged.setDescription(null);
```

The actual deletion of carts is done by the **CartRemovalJob** CronJob. For more information, see the **Save Cart Support** section of the acceleratorservices Extension - Technical Guide document.

ℹ Note

For more information on Cron Jobs, see the cronjob - Technical Guide document.

SaveCartRestorationStrategy

The default class extending the **DefaultCommerceCartRestorationStrategy** class, the **DefaultCommerceSaveCartRestorationStrategy** class, handles the restoration of saved carts. By calling the **restoreCart()** method, the saved cart passed as the parameter is set as an active session's cart, allowing the customer to checkout the products from the cart.

The default implementation of the **SaveCartDao** interface uses **FlexibleSearch** queries in order to retrieve a list of saved carts.

- retrieving a list of saved user carts with a specific status

  DefaultSaveCartDao

```
protected final static String FIND_SAVED_CARTS_FOR_SITE_AND_USER_WITH_STATUS = "SELECT {" + CartModel.PK + "} FROM {"
                        + CartModel._TYPECODE + "}, {" + OrderStatus._TYPECODE + "} " + "WHERE {" + CartModel._TYPECODE + "." +
                        + "} = {" + OrderStatus._TYPECODE + ".pk} AND {" + CartModel.USER + "} = ?user AND {" + CartModel.SITE
                        + "} = ?site AND {" + CartModel.SAVETIME + "} IS NOT NULL AND {OrderStatus.CODE} in (?orderStatus) " + O
```

- retrieving a list of expired saved carts

  DefaultSaveCartDao

```
protected final static String FIND_EXPIRED_SAVED_CARTS_FOR_SITE = SELECTCLAUSE + "WHERE {" + CartModel.SITE
                        + "} = ?site AND {" + CartModel.SAVETIME + "} IS NOT NULL AND {" + CartModel.EXPIRATIONTIME + "} <= ?cur
                        + ORDERBYCLAUSE;
```

- retrieving a list of saved user carts

```
protected final static String FIND_SAVED_CARTS_FOR_USER_AND_SITE = SELECTCLAUSE + "WHERE {" + CartModel.USER
                        + "} = ?user AND {" + CartModel.SITE + "} = ?site AND {" + CartModel.SAVETIME + "} IS NOT NULL " + ORDER
```

Bean Configuration

In the **`commerceservices/resources/commerceservices-beans.xml`** file, you can find the definition of the **CommerceSaveCartParameter** and **CommerceSaveCartResult** beans:

commerceservices-beans.xml

```
<beans>
..
<bean class="de.hybris.platform.commerceservices.service.data.CommerceSaveCartParameter">
        <property name="cart" type="de.hybris.platform.core.model.order.CartModel">
            <description>The CartModel to be saved</description>
        </property>
        <property name="name" type="String">
            <description>The name of the saved cart provided by user or auto-generated</description>
        </property>
        <property name="description" type="String">
            <description>The description of the saved cart provided by user or auto-generated</description>
        </property>
        <property name="enableHooks" type="boolean">
            <description>Should the method hooks be executed</description>
        </property>
    </bean>

<bean class="de.hybris.platform.commerceservices.service.data.CommerceSaveCartResult">
        <property name="savedCart" type="de.hybris.platform.core.model.order.CartModel">
            <description>The CartModel that was saved</description>
        </property>
    </bean>

</beans>
```

Exceptions

In the `commerceservices` extension, the **CommerceSaveCartException** class is defined. This simple exception is thrown if a cart cannot be retrieved, saved, or unsaved.

CommerceSaveCartException.java

```
public class CommerceSaveCartException extends BusinessException
{
        public CommerceSaveCartException(final String message)
        {
                super(message);
        }
        public CommerceSaveCartException(final String message, final Throwable cause)
        {
                super(message, cause);
        }
}
```

Customer Lists Framework

The *Customer List Framework* is a generic framework that can display different customer lists to Customer Support Agents relative to different business applications. The customer records within the customer lists contain ASM deep links that when clicked on by a CSA, take them to their respective pages such as the customer's profile, cart or orders page for example. This allows a CSA to prepare themselves with customer specific information before providing personalized support to the customer. Currently, the following implementations are available:

Customer List Name	Description	Documentation Link
Current In-Store Customers	Displays a list of customers that are currently in-store or in different places within a store as detected by one or multiple hardware devices so that a Customer Support Agent can review a customers profile and approach the customer prepared. By default the customer data is mocked, however in a real life implementation this data will come from a single, or multiple, physical in-store IoT devices.	assistedserviceservices Extension document, section **Current In-Store Customers**
My Recent Customer Sessions	Displays a list of customers whose sessions were recently started by the Customer Support agent so that a Customer Support Agent can quickly return to a previous customer support user journey.	assistedservicefacades Extension document, section **My Recent Customer Sessions**
Pick-Up In-Store Customers	Displays a list of customers who bought online and are picking their orders up in-store so a Customer Support Agent can prepare their order.	assistedserviceservices Extension document, section **Pick-Up In-Store Customers**

Customer Lists Framework Overview

To introduce the relation between the strategy implementation and the customer list, a map is introduced holding the relation between the `implementationType` String and the implementation strategy:

```
<util:map id="customerListSearchStrategyMap" key-type="java.lang.String" value-type="de.hybris.platform.commerceservices.customer.stra
```

Using the Recent Sessions customer list as an example, see how the strategy is mapped to the actual list in the `assistedserviceservices-spring.xml` file:

```
    <bean id="defaultRecentlyStartedSessionCustomerListSearchStrategy"
              class="de.hybris.platform.assistedserviceservices.strategy.DefaultRecentlyStartedSessionCustomerListSearchStrategy">
              <property name="customerSupportEventService" ref="customerSupportEventService"/>
              <property name="userService" ref="userService"/>
        </bean>

        <bean id="defaultRecentlyStartedSessionCustomerListMergeDirective" depends-on="customerListSearchStrategyMap" parent="mapMerge
                <property name="key" value="ASM_RECENT_SESSIONS"/>
                <property name="value" ref="defaultRecentlyStartedSessionCustomerListSearchStrategy" />
        </bean>
```

The `CustomerListService` interface is responsible for returning all valid customer lists available for specific employee, but it is the `CustomerListSearchStrategy` class where the actual business logic is implemented.

ℹ Note

If you want to introduce new customer lists for different business applications, you have to implement new strategies for each module.

Below, you can find a list of the example implementations of the `CustomerListStrategy` interface:

Implementation	Description
`de.hybris.platform.assistedserviceservices.strategy.DefaultRecentlyStartedSessionCustomerListSearchStrategy`	Lists the customers who have recently been assisted by the current agent with maximum of 20 customers displayed. For more information, see the **My Recent Customer Sessions** section of the assistedservicefacades Extension document.
`de.hybris.platform.assistedserviceservices.strategy.DefaultInStoreCustomerListSearchStrategy`	Lists all customers checked-in to a specific point of service (POS). For more information, see the **Current In-Store Customers** section of the assistedserviceservices Extension document and the **points-of-service.impex** section of the assistedservicestorefront AddOn document.
`de.hybris.platform.assistedserviceservices.strategy.DefaultBopisCustomerListSearchStrategy`	Lists all customers who pick up what they bought online in a specific point of service. For more

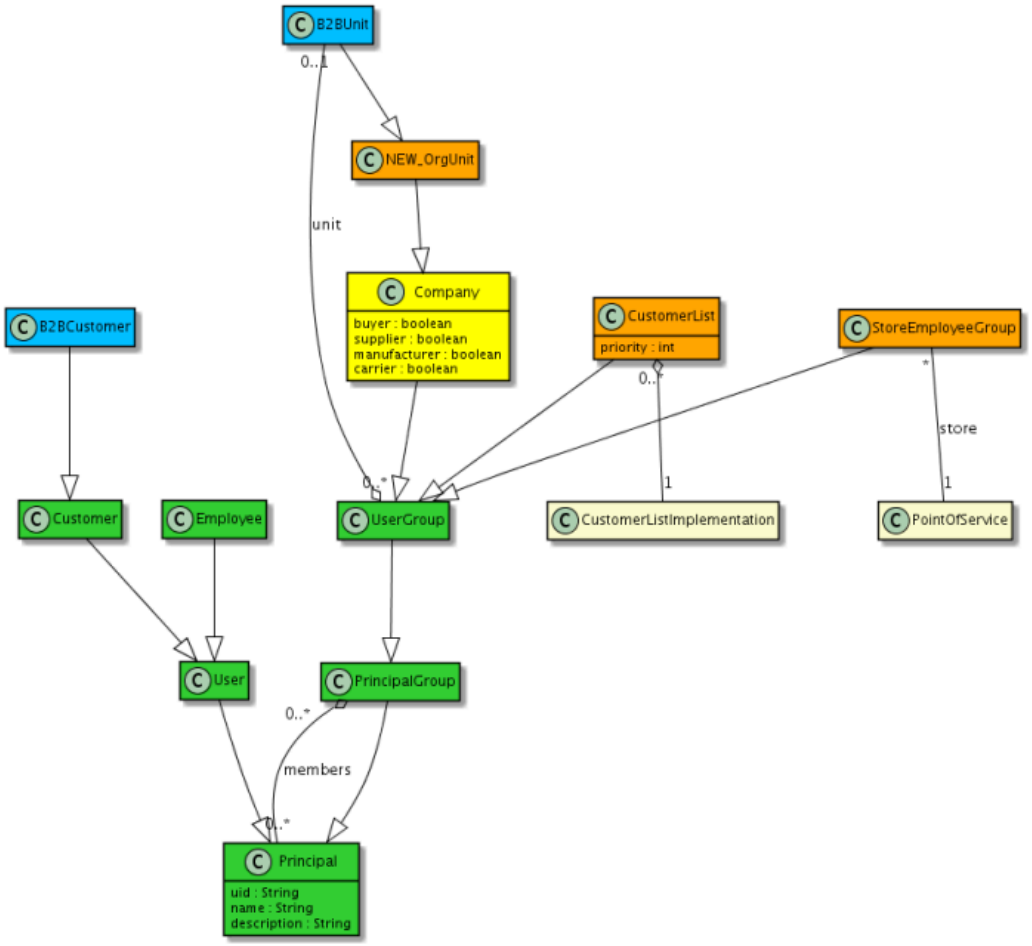
Implementation	Description
	information, see the **Pick-Up In-Store Customers** section of the assistedserviceservices Extension and the **points-of-service.impex** section of the assistedservicestorefront AddOn document.
`de.hybris.platform.b2bcommerceservices.DefaultB2BCustomerListSearchStrategy`	List all B2B customers assigned to a common B2B unit.

Data Model

Below, you can find the UML diagram of the data model of the Customer Lists Framework relative to the `commerceservices` extension:

i Note

core (#LimeGreen)
catalog (#Yellow)
basecommerce (#LightYellow)
commerceservices (#Orange)
acceleratorservices (#Purple)
yacceleratorcore (#Plum)
assistedserviceservices (#Wheat)
b2bcommerce (#DeepSkyBlue)



CustomerList type

The `CustomerList` type is generated, based on the definition found in the `commerceservices-items.xml` file:

```
<typegroup name="CustomerList">
            <itemtype code="CustomerList" autocreate="true" generate="true" extends="UserGroup">
                <description>A CustomerList are visible to certain Employees and represents an implementation specific search query to
                <attributes>
```

```
                    <attribute qualifier="implementationType" type="java.lang.String">
                        <description>The implementation type for this customer list</description>
                        <persistence type="property" />
                    </attribute>
                    <attribute qualifier="priority" type="java.lang.Integer">
                                    <description>priority for the customer list and zero by default, this will affect the pos
                                    <persistence type="property" />
                                    <defaultvalue>Integer.valueOf(0)</defaultvalue>
                                </attribute>
                </attributes>
            </itemtype>
        </typegroup>
```

customerListSearchStrategyMap

The `customerListSearchStrategyMap` bean is a map holding different implementations of the Customer Lists framework. This map is defined in the `commerceservices-spring.xml` file:

```
<util:map id="customerListSearchStrategyMap" key-type="java.lang.String"
                                value-type="de.hybris.platform.commerceservices.customer.strategies.CustomerListSearchStrategy" scop
```

The `customerListSearchStrategyMap` bean takes String values as the keys and the actual implementation beans as values.

ℹ Note

If you create a new customer list, make sure to reference it in this map.

CustomerListService

The `CustomerListService` interface is responsible for returning a list of customer lists for a given employee and for returning a specific list for a specific employee. This default implementation of this interface is referenced in Spring

CustomerListSearchService

The `CustomerListSearchService` interface is responsible for returning paginated customers for a given customer list.

CustomerListSearchStrategy

The `CustomerListSearchStrategy` class is an interface you need to implement in your custom customer list. The `getPagedCustomers()` method is where you need to place your business logic which returns the actual list of customers. If you want to look up how such an implementation may look like, the following classes implement the `CustomerListSearchStrategy` interface:

Class	Available Documentation
`DefaultInStoreCustomerListSearchStrategy`	**In-Store Customers** section of the assistedserviceservices Extension
`DefaultRecentlyStartedSessionCustomerListSearchStrategy`	**Recent Sessions** section of the assistedserviceservices Extension

For more information about the *Customer Lists Framework*, see the commercefacades Extension document.

Pagination and Sorting

The following objects related to pagination and sorting in the `commerceservices` extension have been deprecated:

- SortData
- SearchPageData
- PaginationData
- PagableData

The replacement objects can be found in the `platform` extension.

ℹ Note

Not all of the attributes in the deprecated objects have been migrated to the new objects, and some new data objects do not have any supporting services.

For more information, see Pagination and Sorting.

Future Stock Service

The future stock service provides the functionality of getting the future product availability, such as the stock level and available date, for the specified products.

- **Parameter:** `List<ProductModel> products`.This parameter is a list of product model.
- **Returns:** `Map<String, Map<Date, Integer>>`. Returns a map of product codes with corresponding map of future stock availability.
- If there is no future availability for these products, an empty map is returned.

Related Information

SAP Commerce Accelerator Documentation

commercefacades Extension

The `commercefacades` extension provides a suite of facades that make up a unified multichannel storefront API which can be used by multiple front-ends. The facade's responsibility is to integrate existing business services from the full range of the SAP Commerce extensions and expose a Data object (POJO) response adjusted to meet the storefront requirements.

> **Note**
> An SAP Commerce extension may provide functionality that is licensed through different SAP Commerce modules. Make sure to limit your implementation to the features defined in your contract license. In case of doubt, please contact your sales representative.

Breakdown of a Facade

Facade Bean

A facade in the `commercefacades` extension is a Spring managed bean that is an implementation of a software design pattern of abstracting the underlying implementation responsible for servicing a storefront **action** by exposing a simplified interface optimized for the storefront.

In general facades typically integrate one or more SAP Commerce ServiceLayer method calls that, when combined, fulfill some form of **business action** requested by a storefront user. The facade interface is however totally independent of the ServiceLayer model, so it is possible to completely substitute the underlying implementation of a facade.

Examples of such business actions include:

- Viewing product details.
- Adding a product to a cart.
- Adding a delivery address during checkout.
- Posting a review.
- Searching for products with a free text search.
- Refining or sorting a search result.

Data Objects

The hybris **commercefacades** extension facades mostly return **Data Objects**. These are designed to be used on a frontend like the Accelerator storefronts or mapped into some XML or JSON representation. They are populated using a subset of the data contained in the hybris ServiceLayer Models and are not backed by a persistence layer like Model objects. A Data object may also be constructed from multiple models as well as from data that is derived after executing business logic on services. In an Accelerator storefront these Data objects form part of the **Model** delivered to construct the **View** in the default **MVC** pattern. The hybris ServiceLayer Models should not be used as part of a facade interface, maintaining a clean abstraction of the business layer and the presentation layer.

Source Code Example

We can refer to the **ProductFacade getProductByCode** method for a suitably complex example. The facade interface takes a product article number together with a set of data options as instruction from the front end as to the level of data that needs to be returned in the resulting **ProductData** object. The example uses a Converter, Populators, Configurable Populators and Services.

```
ProductData getProductForCode(String code, Set <ProductOption> options)
        {
                getProductConfiguredPopulator().populate(productModel, productData, options);
        }

        return productData;
}
```

The implementation in **DefaultProductFacade** looks like this:

```
@Override
public ProductData getProductForCode(final String code, final Set<ProductOption> options)
{
    final ProductModel productModel = getProductService().getProductForCode(code);
    final ProductData productData = getProductConverter().convert(productModel);
```

```
    if (options != null)
    {
        getProductConfiguredPopulator().populate(productModel, productData, options);
    }

    return productData;
}
```

It first calls the **ProductService** to get the product model for the provided article number. The configured **productConverter** bean is then used to create the prototype **ProductData** and the ProductPopulator is used to populate the object with basic data.

```
@Override// From ProductPopulator
@Override
protected ProductData createTarget(final ProductModel source)
{
  return new ProductData();
}

// From ProductConverter
@Override
public void populate(final ProductModel source, final ProductData target)
{
     getProductBasicPopulator().populate(source, target);
     getVariantSelectedPopulator().populate(source, target);
     getProductPrimaryImagePopulator().populate(source, target);

      super.populate(source, target);
}
```

→ Tip

Extensibility Tip

Use the **`beans.xml`** file to generate new attributes on your **ProductData** then add new Populators for these attributes.

The **ProductBasicPopulator** looks like this:

ProductBasicPopulator

```
@Override
public void populate(final ProductModel productModel, final ProductData productData) throws ConversionException
{
        productData.setName((String) getProductAttribute(productModel, ProductModel.NAME));
        productData.setManufacturer((String) getProductAttribute(productModel, ProductModel.MANUFACTURERNAME));

        productData.setAverageRating(productModel.getAverageRating());
        if (productModel.getVariantType() != null)
        {
                productData.setVariantType(productModel.getVariantType().getCode());
        }

        productData.setPurchasable(Boolean.valueOf(productModel.getVariantType() == null && isApproved(productModel)));
}

protected boolean isApproved(final ProductModel productModel)
{
        final ArticleApprovalStatus approvalStatus = productModel.getApprovalStatus();
        return approvalStatus != null && ArticleApprovalStatus.APPROVED.equals(approvalStatus);
}
```

ⓘ Note

Implementation Detail

Product Populators are a little unique in that they typically extend a **AbstractProductPopulator** class that is variant-aware and supports the ability of falling back to a variants parent product for attribute values in the event of the source product value being **null**.

The **ProductFacade** uses then its configured Populator to populate the prototype Data Object based on the requested data options. The configured Populator is simply a spring configured pipeline of populators that are invoked in a configured order and only invokes if they are assigned to an option that has been specified. Here's a sample spring configuration of the **ConfigurablePopulator** used by the **ProductFacade**.

```
<alias name="defaultProductConfiguredPopulator" alias="productConfiguredPopulator" />
        <bean id="defaultProductConfiguredPopulator" class="de.hybris.platform.commercefacades.convert.impl.DefaultConfigurablePopulat
                <property name="populators">
                        <map key-type="de.hybris.platform.commercefacades.product.data.ProductData$ProductOption">
                                <entry key="GALLERY" value-ref="productGalleryImagesPopulator" />
                                <entry key="SUMMARY" value-ref="productSummaryPopulator" />
                                <entry key="DESCRIPTION" value-ref="productDescriptionPopulator" />
                                <entry key="CATEGORIES" value-ref="productCategoriesPopulator" />
                                <entry key="PROMOTIONS" value-ref="productPromotionsPopulator" />
                                <entry key="STOCK" value-ref="productStockPopulator" />
                                <entry key="REVIEW" value-ref="productReviewsPopulator" />
                                <entry key="CLASSIFICATION" value-ref="productClassificationPopulator" />
                                <entry key="REFERENCES" value-ref="productReferencesPopulator" />
                                <entry key="VARIANT_FULL" value-ref="variantFullPopulator" />
                        </map>
```

```
                </property>
        </bean>
```

As it was mentioned above, some Populators may use services to fill out the prototype Data Object rather than properties on the source model. You can review the **ProductPricePopulator** as an example of a Populator that uses a service to fill out part of the data model rather than the source object directly.

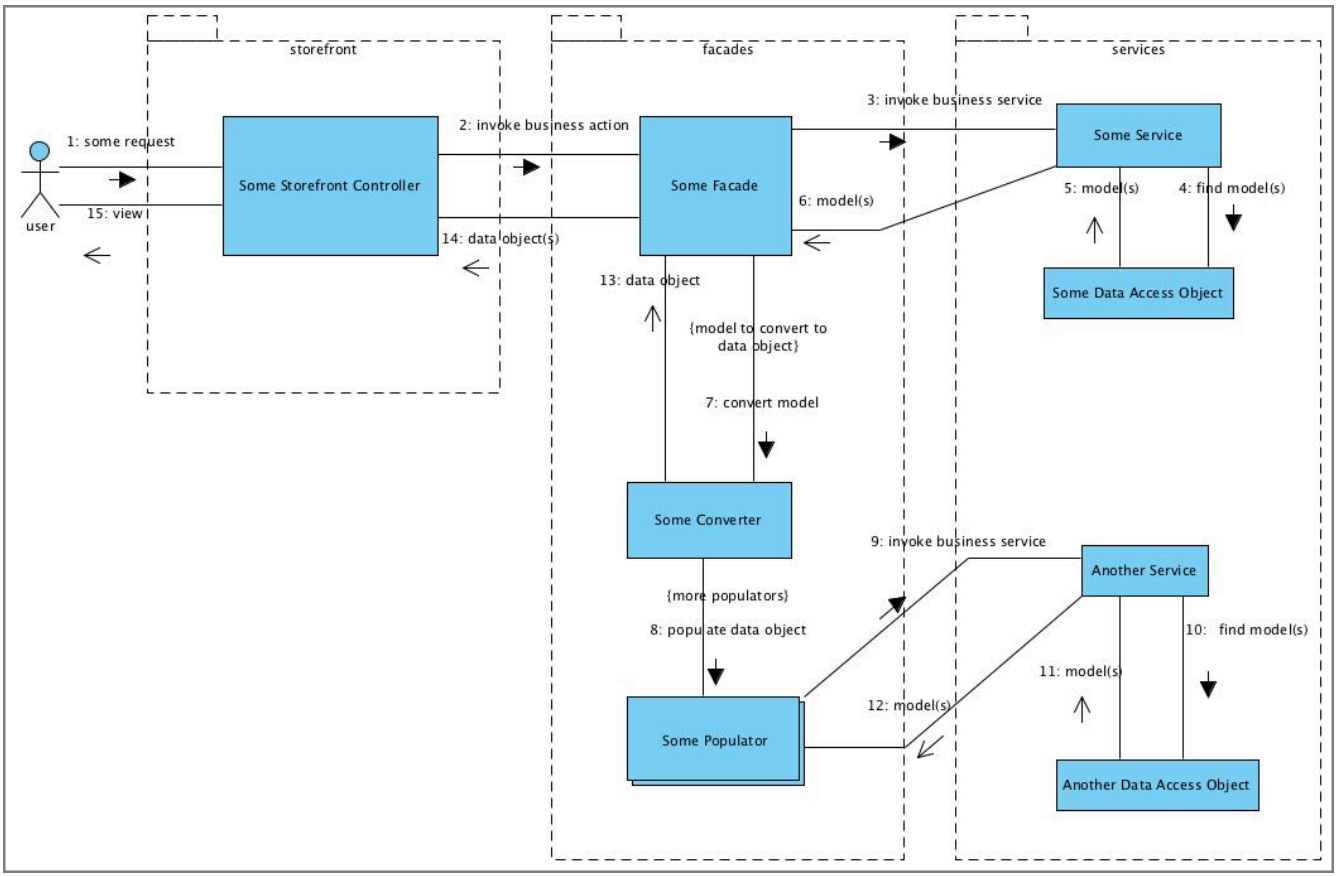
```
@Override
public void populate(final ProductModel productModel, final ProductData productData) throws ConversionException
{
        final PriceData.PriceType priceType;
        final PriceInformation info;
        if (CollectionUtils.isEmpty(productModel.getVariants()))
        {
                priceType = PriceData.PriceType.BUY;
                info = getCommercePriceService().getWebPriceForProduct(productModel);
        }
        else
        {
                priceType = PriceData.PriceType.FROM;
                info = getCommercePriceService().getFromPriceForProduct(productModel);
        }

        if (info != null)
        {
                final PriceData priceData = getPriceDataFactory().create(priceType, BigDecimal.valueOf(info.getPriceValue().getValue()
                info.getPriceValue().getCurrencyIso());
                productData.setPrice(priceData);
        }
        else
        {
                productData.setPurchasable(Boolean.FALSE);
        }
}
```

Extending

- Adding new facades
- Extending existing facades interface and implementation to add additional methods
- Extending existing facade implementation to replace the original method implementation
- Creating a different or additional versions of a converter for a particular type
- Extending an existing converter implementation
- Adding a new Populator to a Populator pipeline
- Removing or replacing Populator implementations
- Adding new data objects and counterpart Populators and Converters
- Extending existing data objects and adding additional converter and/or Populator logic

Conceptual Interaction Diagram

The following interaction diagram describes conceptually how the various components interact during a typical storefront request:ProductData getProductForCode(String code, Set<ProductOption> options) throws UnknownIdentifierException, IllegalArgumentException;



A typical storefront interaction can be described as follows:

1. A user performs some action which is processed by a storefront component such as an MVC controller.

2. The controller translates the request and invokes a business action on the appropriate facade.

3. The facade invokes one or more business services to fulfill the action.

4-6. The service most likely needs to return some model object and perhaps uses Data Access Objects to find the models to return to the calling facade.

7. The facade uses converters to create front-end data objects from service layer models.

8. Converters use a pipeline of Populators fill the skeleton front end data objects graph.

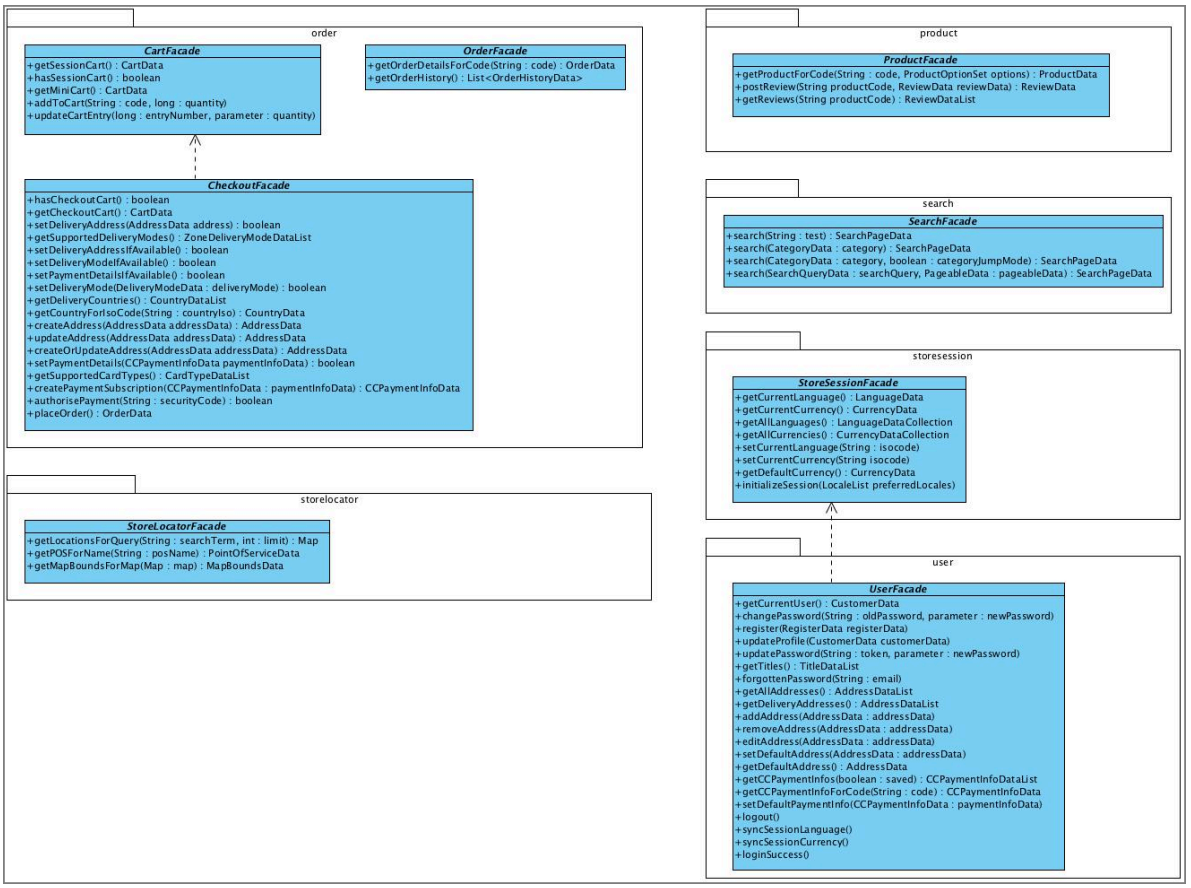
9-12. Populators may also need to invoke additional business logic to fill the front end data object graph so may also invoke services in order to provide the necessary data required by the front-end data object.

13-14. The filled data object is returned to the facade and the facade then returns this to the controller.

15. The controller uses the data object as a model to construct the view which will be returned to the user.

Packaged Facades

The following class diagram outlines the various facades offered by the **commercefacades** extension.



Search

The search facade exposes product free text search and facetted navigation capability.

Product

The product facade delivers product information. It is possible to control the amount of information returned for the product by specifying data options. The product facade also enables reviews to be posted for products.

The `getProductForOptions()` method:

- `DEVICE_BUNDLE_TABS` - to populate **Devices**. The **BundleTabs** section is populated with additional information about the service plans sorted by package and frequency.
- `SERVICE_PLAN_BUNDLE_TABS` - to populate **Service Plans**. The whole page is populated with all existing service plans sorted by package and frequency.
- `SERVICE_ADDON_BUNDLE_TABS` - to populate **Service AddOns**. The whole page is populated with all existing service plans sorted by package and frequency.

Order

The order facades provide the full shopping capability with all the typical cart, checkout and order history functions. The functionality is broken down into three separate facades, cart facade for the shopping phase, checkout facade for the order placement process and order facade for bringing up details of submitted orders and order history.

User

The User facade provides user account operations such as registration, payment and address book management, profile updates including password amendments.

Store Session

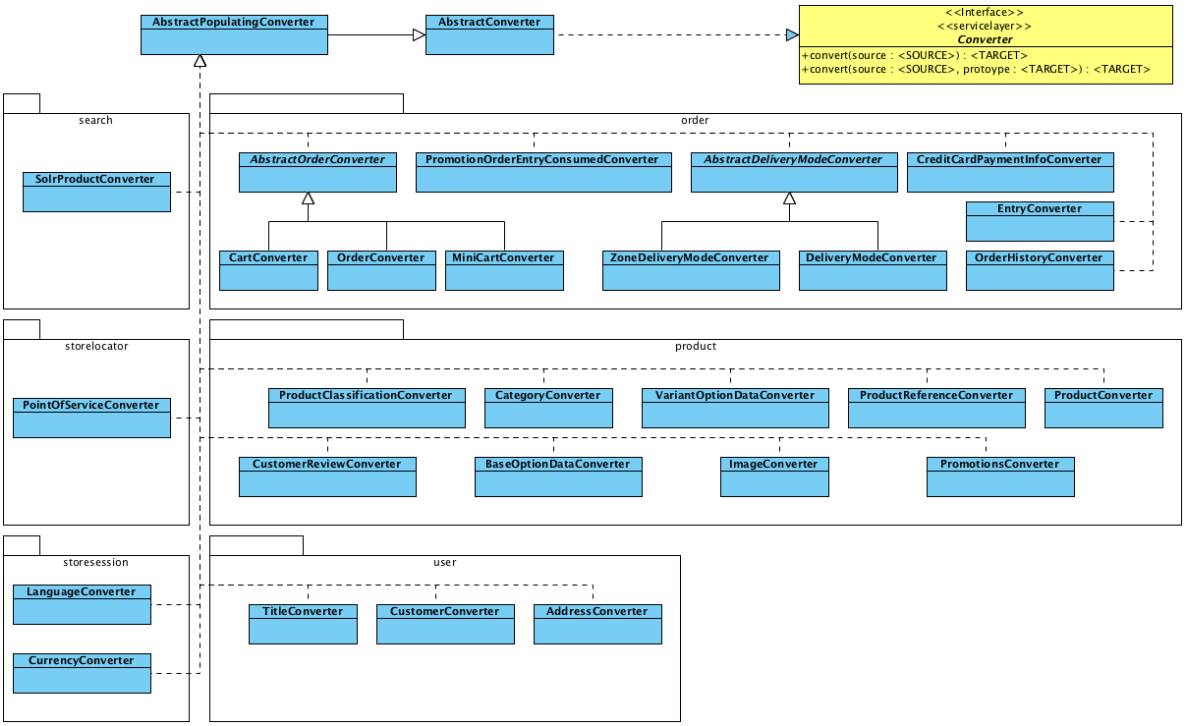
The Store Session facade provides access to various internationalization switches that the user can make when they visit a specific storefront.

Store Locator

The Store Locator facade exposes Point Of Sale search capability and provides the ability to expose store information and store specific content. Find more information on the Store Locator within the hybris Accelerator in the Store Locator Configuration document.

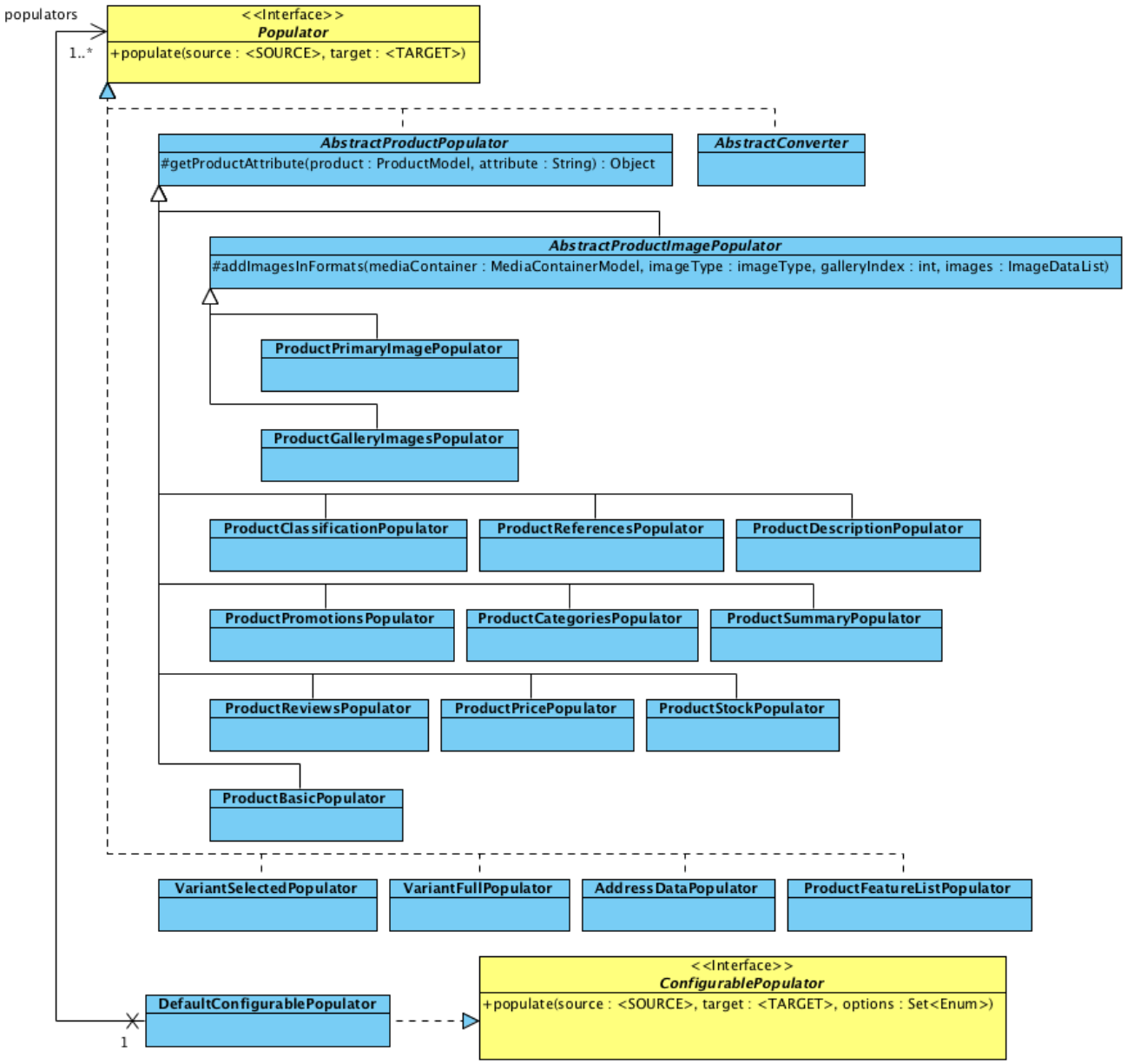
Packaged Converters

The following diagram shows the Converters shipped with the **commercefacades** extension:



Packaged Populators

The following diagram shows the Populators shipped with the **commercefacades** extension:



Extending Facades

Facades are singleton scoped Spring managed beans that have an interface and a default implementation class that can be customized by a number of means including replacing the entire bean, subclassing the bean class and overriding specific methods or customization by composition, typically by populating the data object model using different or additional instances of converters and populators.

You should extend the **commercefacades** extension facades in a counterpart **facades** extension for your project. The **yacceleratorfacades** extension is the template shipped with the hybris Accelerator that you can use as a starting point for your own extension. This extension must have a dependency on the **commercefacades** extension in its **`extensioninfo.xml`** file as well as any other necessary business layer extension required to complete the functionality offered by your customized or additional facades.

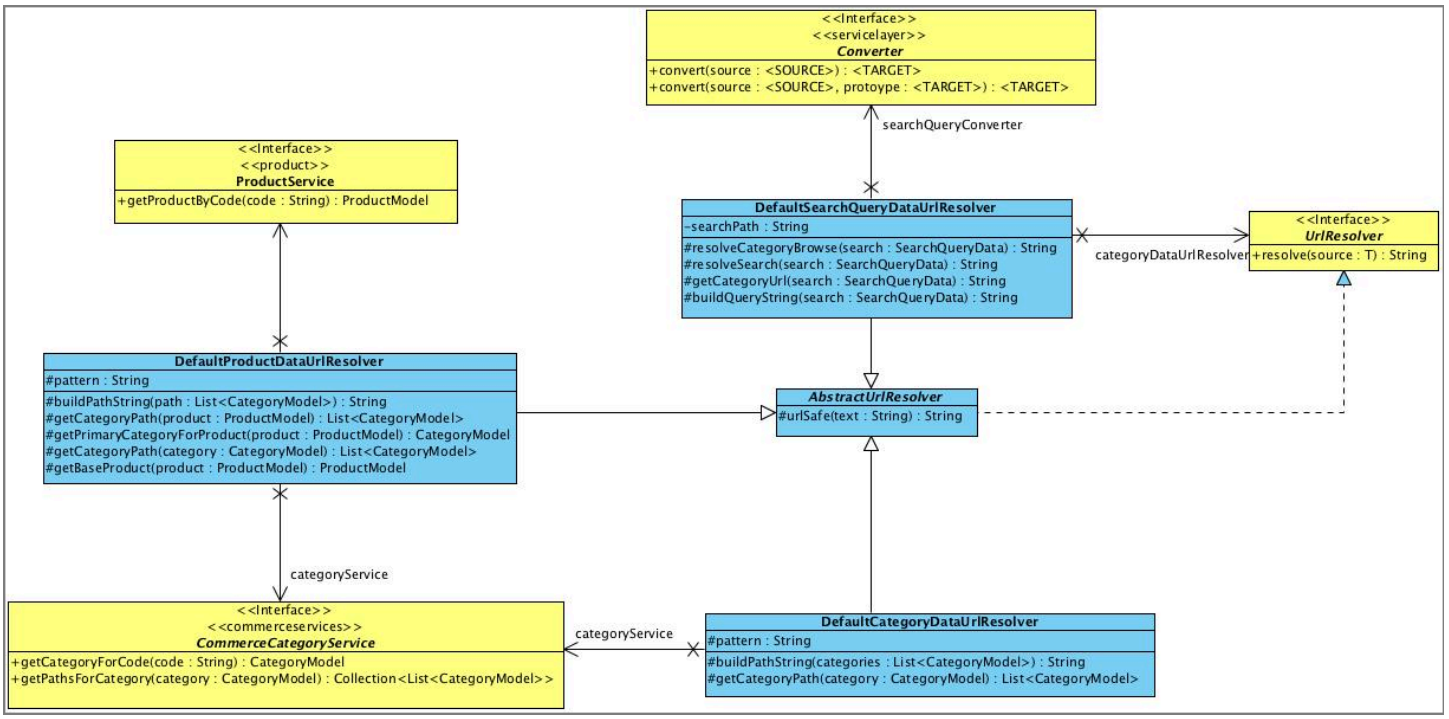
ⓘ Note

Implementation Detail

Adding the dependency on the **commercefacades** extension in the **`extensioninfo.xml`** ensures you can replace bean definitions defined in the **commercefacades** extension application context. You can find more information on extending Spring beans can in Spring Framework in the SAP Commerce, ServiceLayer and more specifically How to Extend a Service documents.

URL Resolvers

A URL resolver creates a relative URL to a provided source item that can be used on the front end to generate a link back to the source item. This ensures the facade layer is able to supply URLs in the data object model without the MVC layer having to execute an additional pass over the data object model. This approach also allows multiple client applications to use different URL formats by simply using facades and converters configured with different resolver instances. Out of the box, SEO friendly URLs can be generated for product and category pages as well as facetted navigation pages.



Save Cart

The save cart functionality primarily allows a customer to save and restore their saved carts at a later date. This functionality is provided as a set of methods embedded within independent strategies that can be wired-up, relative to the different business requirements and the front-end implementations they are used for. These strategies can be easily extended with pre/post-hooks.

The following save cart operations are currently supported:

- Save a session cart as a saved cart
- Save specific cart IDs, for a back-end operations, as saved carts
- Display a list of saved carts
- Display the details of a saved cart
- Restore a saved cart to an active session cart
- Delete saved carts
- Clone saved carts
- Edit the name and description of a saved cart

In order to support this functionality, a new **SaveCartFacade** interface has been implemented. This interface contains the following methods:

Method Signature	Returns
saveCart(CommerceSaveCartParameterData)	**CommerceSaveCartResultData**
flagForDeletion(String)	**CommerceSaveCartResultData**
getCartForCodeAndCurrentUser(CommerceSaveCartParameterData)	**CartData**
restoreSavedCart(CommerceSaveCartParameterData)	**CartRestorationData**
getSavedCartsForCurrentUser(PageableData , List<OrderStatus>)	**SearchPageData<CartData>**
cloneSavedCart(CommerceSaveCartParameterData)	**CommerceSaveCartResultData**

Upon calling the **saveCart()** method, it expects that, amongst other parameters, a **name** and **description** of the cart is provided. If neither of them are provided, the method generates a name and description based on the **cartModel** of the cart about to be saved.

ℹ Note

If you want to change this behavior, you have to overwrite the entire **saveCart()** method in your implementation of the **SaveCartFacade** interface.

Apart from implementing the **SaveCartFacade**, your custom implementation of this interface has to extend an implementation of the **CartFacade** interface in order to support operations on **Cart** objects. In other words, if your implementations of the **CartFacade** and **SaveCartFacade** interfaces are named **DefaultCartFacade** and **DefaultSaveCartFacade** respectively, the class header looks as follows:

DefaultCartFacade

```
public class DefaultSaveCartFacade extends DefaultCartFacade implements SaveCartFacade
```

CommerceCartFacade

The `CommerceCartFacade.addToCart(AddToCartParams addToCartParams)` method is used to populate new entry with some module-specific data. For example: `configurablebundles` use this method to add bundle group to entry.

Bean Configuration

Methods of the **SaveCartFacade** use or return the following objects:

- **CommerceSaveCartResultData**,
- **CommerceSaveCartParameterData**
- **CartRestorationData**

These objects are generated resources created from Java beans and you can configure them by changing their properties in the **`commercefacades-beans.xml`** file:

commercefacades/resources/commercefacades-beans.xml

```
<beans>
...
        <bean class="de.hybris.platform.commercefacades.order.data.CommerceSaveCartParameterData">
                <property name="cartId" type="String"/>
                <property name="name" type="String"/>
                <property name="description" type="String"/>
                <property name="enableHooks" type="boolean"/>
        </bean>
        <bean class="de.hybris.platform.commercefacades.order.data.CommerceSaveCartResultData">
                <property name="savedCartData" type="de.hybris.platform.commercefacades.order.data.CartData"/>
        </bean>

        <bean class="de.hybris.platform.commercefacades.order.data.CartRestorationData">
                <property name="modifications"
                          type="java.util.List&lt;de.hybris.platform.commercefacades.order.data.CartModificationData&gt;"/>
        </bean>
...
</beans>
```

Exceptions

Generally, in case of any issues, the methods of the **SaveCartFacade** interface throw a **CommerceSaveCartException**. For more information, see the **Exceptions** section of the commerceservices Extension -Technical Guide document.

Customer Lists Framework

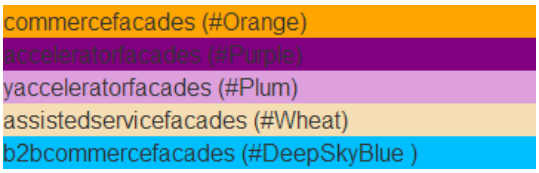
The *Customer List Framework* is a generic framework that can display different customer lists to Customer Support Agents relative to different business applications. The customer records within the customer lists contain ASM deep links that when clicked on by a CSA, take them to their respective pages such as the customer's profile, cart or orders page for example. This allows a CSA to prepare themselves with customer specific information before providing personalized support to the customer. Currently, the following implementations are available:

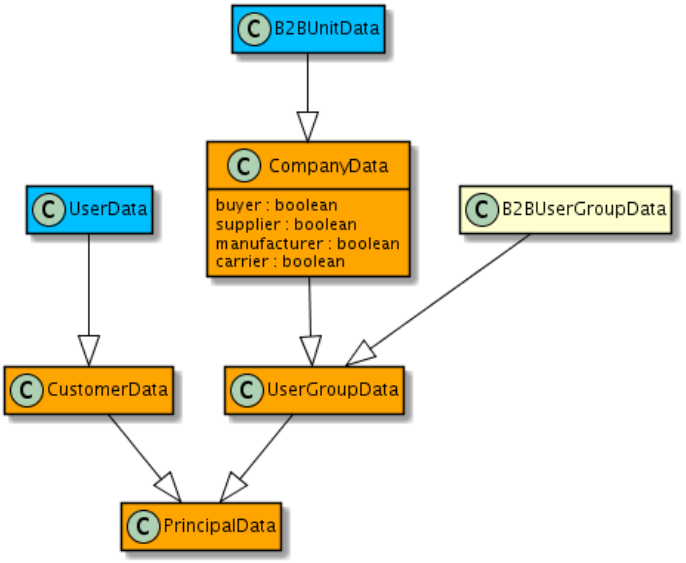
Customer List Name	Description	Documentation Link
Current In-Store Customers	Displays a list of customers that are currently in-store or in different places within a store as detected by one or multiple hardware devices so that a Customer Support Agent can review a customers profile and approach the customer prepared. By default the customer data is mocked, however in a real life implementation this data will come from a single, or multiple, physical in-store IoT devices.	assistedserviceservices Extension document, section **Current In-Store Customers**
My Recent Customer Sessions	Displays a list of customers whose sessions were recently started by the Customer Support agent so that a Customer Support Agent can quickly return to a previous customer support user journey.	assistedservicefacades Extension document, section **My Recent Customer Sessions**
Pick-Up In-Store Customers	Displays a list of customers who bought online and are picking their orders up in-store so a Customer Support Agent can prepare their order.	assistedserviceservices Extension document, section **Pick-Up In-Store Customers**

Data Model

Below, you can find the UML diagram of the data model of the Customer Lists Framework relative to the `commercefacades` extension:

i Note





CustomerListFacade

The `CustomerListFacade` interface is a class which contains the methods used for:

- getting the customer lists for specific employee tobe displayed on the frontend
- getting paginated customers for a specific customer list UID with extra parameters, if needed.

The `getPagedCustomersForCustomerListUID()` method calls the strategy from inside and returns a list of paginated customers and converts the customer models into customer data using populators.

The default implementation of the `CustomerListFacade` interface contains the following methods:

```
/**
 *
 * Default customer list facade concrete class which implements {@link CustomerListFacade}
 *
 */
public class DefaultCustomerListFacade implements CustomerListFacade
{
```

```
	//holds different customer list implementations
	private CustomerListService customerListService;
	private CustomerListSearchService customerListSearchService;
	private Converter<CustomerListModel, UserGroupData> userGroupConverter;
	private Map<String, Converter<UserModel, CustomerData>> customerListImplementationStrategiesConverter;
	private Converter<UserModel, CustomerData> customerConverter;


	@Override
	public List<UserGroupData> getCustomerListsForEmployee(final String employeeUid)
	{
		validateParameterNotNullStandardMessage("employeeUid", employeeUid);

		return getUserGroupConverter().convertAll(getCustomerListService().getCustomerListsForEmployee(employeeUid));
	}

	@Override
	public <T extends CustomerData> SearchPageData<T> getPagedCustomersForCustomerListUID(final String customerListUid,
			final String employeeUid, final PageableData pageableData, final Map<String, Object> parameterMap)
	{
		validateParameterNotNullStandardMessage("customerListUid", customerListUid);

		validateParameterNotNullStandardMessage("pageableData", pageableData);

		validateParameterNotNullStandardMessage("employeeUid", employeeUid);

		final CustomerListModel customerListModel = getCustomerListService().getCustomerListForEmployee(customerListUid,
				employeeUid);

		validateParameterNotNull(customerListModel,
				String.format("No CustomerList found for customerListUid '%1$s'", customerListUid));

		final String implementationType = customerListModel.getImplementationType();

		final SearchPageData<CustomerModel> searchPageData = getCustomerListSearchService().getPagedCustomers(customerListUid,
				employeeUid, pageableData, parameterMap);

		List<CustomerData> customerDataList = null;

		final Converter<UserModel, CustomerData> strategyConverter = getCustomerListImplementationStrategiesConverter()
				.get(implementationType);

		if (null == strategyConverter)
		{
			customerDataList = getCustomerConverter().convertAll(searchPageData.getResults());
		}
		else
		{
			customerDataList = strategyConverter.convertAll(searchPageData.getResults());
		}

		final SearchPageData<T> customersSearchPageData = new SearchPageData<T>();

		customersSearchPageData.setResults((List<T>) customerDataList);
		customersSearchPageData.setPagination(searchPageData.getPagination());
		customersSearchPageData.setSorts(searchPageData.getSorts());

		return customersSearchPageData;
	}


	protected CustomerListService getCustomerListService()
	{
		return customerListService;
	}

	@Required
	public void setCustomerListService(final CustomerListService customerListService)
	{
		this.customerListService = customerListService;
	}

	protected Converter<CustomerListModel, UserGroupData> getUserGroupConverter()
```

```
	{
		return userGroupConverter;
	}

	@Required
	public void setUserGroupConverter(final Converter<CustomerListModel, UserGroupData> userGroupConverter)
	{
		this.userGroupConverter = userGroupConverter;
	}

	protected Map<String, Converter<UserModel, CustomerData>> getCustomerListImplementationStrategiesConverter()
	{
		return customerListImplementationStrategiesConverter;
	}

	@Required
	public void setCustomerListImplementationStrategiesConverter(
			final Map<String, Converter<UserModel, CustomerData>> customerListImplementationStrategiesConverter)
	{
		this.customerListImplementationStrategiesConverter = customerListImplementationStrategiesConverter;
	}

	protected Converter<UserModel, CustomerData> getCustomerConverter()
	{
		return customerConverter;
	}

	@Required
	public void setCustomerConverter(final Converter<UserModel, CustomerData> customerConverter)
	{
		this.customerConverter = customerConverter;
	}

	protected CustomerListSearchService getCustomerListSearchService()
	{
		return customerListSearchService;
	}

	@Required
	public void setCustomerListSearchService(final CustomerListSearchService customerListSearchService)
	{
		this.customerListSearchService = customerListSearchService;
	}
}
```

Populators

The `getPagedCustomersForCustomerListUID()` method contains a reference to a map for strategies converters. The keys of that map are the customer list IDs and the values are the actual converters that convert the model instances returned from the `CustomerListSearchService` invocation to data instances instead. If there is no converter for specific customer list then a default converter is used instead which populates the address of the customer , profile picture and session cart, if exists.

Spring Configuration

In the `commercefacades-spring.xml` file, the `customerListImplementationStrategiesConverter` map is defined. This map holds different types of converters used to convert models to data. Also, you can find the definition of the `customerListFacade` bean and its dependent services in the `commercefacades-spring.xml` file:

```
<alias name="defaultCustomerListFacade" alias="customerListFacade"/>
	<bean id="defaultCustomerListFacade"
			class="de.hybris.platform.commercefacades.customer.impl.DefaultCustomerListFacade">
		<property name="customerListSearchService" ref="customerListSearchService"/>
		<property name="customerListImplementationStrategiesConverter" ref="customerListImplementationStrategiesConverter"/>
		<property name="customerListService" ref="customerListService"/>
		<property name="userGroupConverter" ref="userGroupConverter"/>
		<property name="customerConverter" ref="customerConverter"/>
	</bean>
```

For more information about the *Customer Lists Framework*, see the commerceservices Extension document.

Generic Mechanism for Adding Columns

The following snippet from `commercefacades/resources/commercefacades-spring.xml` shows the generic mechanism that allows you to add columns to the customer list:

```
<alias name="defaultCustomerListConverter" alias="customerListConverter"/>
<bean id="defaultCustomerListConverter" parent="abstractPopulatingConverter">
	<property name="targetClass" value="de.hybris.platform.commercefacades.user.data.CustomerListData"/>
	<property name="populators">
		<list>
			<ref bean="customerListPopulator"/>
		</list>
	</property>
</bean>

<alias name="defaultCustomerListPopulator" alias="customerListPopulator"/>
<bean id="defaultCustomerListPopulator" class="de.hybris.platform.commercefacades.user.converters.populator.CustomerListPopulator">
	<property name="customerListAdditionalColumnsMap" ref="customerListAdditionalColumnsMap"/>
</bean>

<!-- The keys of the map are CustomerList.additionalColumnsKeys of the added columns and the values of the map are the EL syntax of Cu
<util:map id="customerListAdditionalColumnsMap" key-type="java.lang.String" value-type="java.lang.String" scope="tenant"/>
```

For more information on adding your own columns to the customer list, see ASM Integration.

Future Stock Facade

Facade Layer

A `DefaultFutureStockFacade` implementation of the `FutureStockFacade` interface connects to `FutureStockService`. This connection is used to query the future stock data. The request is passed by `FutureStockFacade` as a list to `FutureStockService`.

The `DefaultFutureStockFacade` provides the following methods:

- Gets the future product availability, such as the stock level and available date, for the specified product.
  - **Parameter:** `productCode`. This parameter is product code.
  - **Returns:** `List<FutureStockData>`. oReturns a list of future stock data ordered by date.
  - If there is no future availability for this product, an empty list is returned.
- Gets the future product availability, such as the stock level and available date, for the list of specified products.
  - **Parameter:** `List<String> productCodes`. This parameter is list of product codes.
  - **Returns:** `Map<String, List<FutureStockData>>`. Returns a map of product codes with a list of future stock data ordered by date.
  - If there is no future availability for these products, an empty map is returned.
- Gets the future product availability, such as the stock level and available date, for the list of specified variants related to a given product.
  - **Parameter:** `productCode, List<String> skus`. This parameter includes two parts. The first is product code of the base product. If a product variant is given, the corresponding base product will be derived. The second one is list of SKUs (product variants codes for the base product).
  - **Returns:** `Map<String, List<FutureStockData>>`. Returns a map of product codes that are included in the given SKUs and match the given base product, with corresponding list of future stock data ordered by date.
  - If the first parameter is a base product without any variants, returns null.

Data Model

Below, you can find the data model of future stock facade:

Integration of a Custom Future Stock Service

If you want to insert your own Future Stock Availability implementation, you need to implement the `FutureStockFacade` interface and configure it. You can either replace the current configuration or create your own overriding XML config for this bean.

Here is the spring configuration for `FutureStockFacade`.

```
<alias name="defaultFutureStockFacade" alias="futureStockFacade"/>
<bean id="defaultFutureStockFacade" class="de.hybris.platform.commercefacades.futurestock.impl.DefaultFutureStockFacade" >
  <property name="commerceCommonI18NService" ref="commerceCommonI18NService"/>
  <property name="futureStockService" ref="futureStockService"/>
  <property name="productService" ref="productService"/>
</bean>
```

The controller uses the implementation you insert for `futureStockFacade` and can stay unchanged.

Related Information

SAP Commerce Accelerator Documentation

Using Facades and DTOs - Best Practice
Models
ServiceLayer
How to Extend the ProductFacade
Search and Navigation

commercewebservicescommons Extension

The `commercewebservicescommons` extension enables Commerce web services to be extended and customized using the addon concept.

Features of `commercewebservicescommons` Extension

Errors

SAP Commerce provides standard WebserviceExceptions used in the `ycommercewebservices`. You can use the default exceptions or you can implement your own exceptions by extending the `WebserviceException/WebserviceValidationException`. SAP Commerce provides the following default exceptions:

- WebserviceException
- WebserviceValidationException
- CartAddressException
- CartEntryException
- CartException
- LowStockException
- ProductLowStockException
- RequestParameterException
- SessionAttributeException
- StockSystemException

SAP Commerce also provides a mechanism for converting objects into errors, known as ErrorData objects. The following table lists the default converters provided by SAP Commerce:

Converter	Object
WebserviceExceptionConverter	de.hybris.platform.commercewebservicescommons.errors.exceptions.WebserviceException
ExceptionConverter	java.lang.Exception
ValidationErrorConverter	org.springframework.validation.Errors
CartModificationDataListErrorConverter	de.hybris.platform.commercefacades.order.data.CartModificationDataList
CartModificationDataErrorConverter	de.hybris.platform.commercefacades.order.data.CartModificationData

DataMapper

All classes related to the DataMapper (filters, converters, mappers, etc.) are available in DTO Mapping and Response Configuration.

Pagination and Sorting

The following objects related to pagination and sorting in the `commercewebservicescommons` extension have been deprecated:

- SortWsDto
- PaginationWsDto
- PagableWsDto

The replacement objects can be found in the `webservicescommons` extension.

For more information, see Pagination and Sorting.

Related Information

OCC AddOns Architecture

ycommercewebservicestest Extension

The `ycommercewebservicestest` extension was created to enable third parties to extend and extension provides a set of tests written in Groovy that are intended to test `ycommercewebservices` REST calls.

⚠ Caution

This page refers to software that has been deprecated as part of the Accelerator UI and older OCC template extensions deprecation. For more information, see Deprecation of Accelerator UIs and Older OCC Template Extensions.

The Groovy tests are based on a sample data set, **wsTest**, which is also included in this extension. A template extension, `ycommercewebservices` is also included. Therefore, it is possible to extend the test suite after extending the `ycommercewebservices` functionality.

Sample Data Set

Data is loaded before running the test suite and removed after tests are finished. The data set consists of over twenty ImpEx files available in the following directory: `ycommercewebservicestest/resources/ycommercewebservicestest/import`. The data set contains a separate store called the wsTest Store and can only be loaded for junit tenant when the entire test suite is running.

Files are imported in the `createProjectData` function in `YCommerceWebServicesTestSetup` class. Any new files have to be listed in this class. Groovy test suites loads data using the `TestSetupUtils` class, which contains static method to load data and run embedded server.

Test Suite

The tests included in the `ycommercewebservicestest` extension are written in Groovy.

ⓘ Note

Test suites contain logic used to load data needed for the test and starting the embedded tomcat server. If the mentioned two steps are omitted, the test will not run properly. That's why running single test will not pass and tests should be run as whole test suite.

Test Suite Name	Description
`de.hybris.platform.ycommercewebservicestest.test.groovy.webservicetests.v2.spock.AllSpockTests`	a test suite for v2 version of `ycommercewebservices`
`de.hybris.platform.ycommercewebservicestest.test.groovy.webservicetests.addons.v2.spock.AllAccSpockTests`	a test suite for endpoints defined in the `acceleratorwebservicesaddon`
`de.hybris.platform.ycommercewebservicestest.test.groovy.webservicetests.v1.AllTests`	a test suite for v1 version of `ycommercewebservices`
`de.hybris.platform.ycommercewebservicestest.test.groovy.webservicetests.addons.AllAccTests`	a test suite for endpoints defined in acceleratorwebservicesaddon and related to v1 version.

Tests are available in the following directory: `ycommercewebservicestest\testsrc\de\hybris\platform\ycommercewebservicestest\test\groovy`.

The test configuration can be found in the following file: `ycommercewebservicestest\resources\groovytests-property-file.groovy`

ⓘ Note

Notice that tests in Groovy use regular Java classes, so the `ycommercewebservicestest` extension must be compiled before starting tests.

Tests may be run as the standard SAP Commerce integration tests from SAP Commerce Testweb Frontend (for details see: The SAP Commerce Testweb Front End) or by using the following ant command:

```
ant integrationtests -Dtestclasses.packages=de.hybris.platform.ycommercewebservicestest.test.groovy.webservicetests.v2.spock.AllSpockT
```

You can also run a single spock test with the following ant command:

```
ant manualtests -Dtestclasses.packages=de.hybris.platform.ycommercewebservicestest.test.groovy.webservicetests.v2.spock.carts.CartResc
```

Bear in mind that running a single test does not import the sample data. To do so, run the following command:

```
ant importwstestdata
```

Related Information

OCC AddOns Architecture

ycommercewebservices Extension

The `ycommercewebservices` extension exposes part of the Commerce Facades as REST-based web services, including calls for product search and product details. extension was created to enable third parties to extend and customize Web services using the addon concept.

⚠ Caution

This page refers to software that has been deprecated as part of the Accelerator UI and older OCC template extensions deprecation. For more information, see Deprecation of Accelerator UIs and Older OCC Template Extensions.

The `ycommercewebservices` extension currently exposes part of the Commerce Facades as REST-based web services, including calls for product search and product details. The focus is to provide a working example of how a REST-based API can be exposed. As the Commerce Web Services are based on standard Spring MVC, you can easily customize or extend them.

i Note

An SAP Commerce extension may provide functionality that is licensed through different SAP Commerce modules. Make sure to limit your implementation to the features defined in your contract license. In case of doubt, please contact your sales representative.

General Information

A web service is a method of communication between two electronic devices over a network. The client typically uses a REST API to communicate with the web services back-end. It can also use plain HTTP connections, for instance through AJAX or any other HTTP-based networking classes.

There are many different clients, both mobile and desktop, that could potentially consume the web services. The clients might or might not be written in the same language as the server code, and hence it makes sense to use a universal language to communicate. The Commerce Web Services support both XML and JSON data formats, which can easily be parsed on all mobile operating systems as well as desktop clients.

The Commerce Web Services are essentially a relatively thin additional layer on top of the Commerce Facades, which perform certain object conversions. The Commerce Web Services are dependent on the `commercefacades` extension. The dependencies are defined in the `extensioninfo.xml` file in the `ycommercewebservices` extension.

```
...
<requires-extension name="commercefacades"/>
...
```

Configuration of Commerce Web Services

By default, the Commerce Web Services are available under the **rest** web context, as specified in the `extensioninfo.xml` file.

```
...
<webmodule jspcompile="false" webroot="/rest" />
...
```

i Note

Commerce Web Services are available in two versions: Version 1 and Version 2 (the default version). For details see: v1 and v2 in ycommercewebservices.

- For V1 calls reference along with sample URLs see: Calls Reference - v1 and OCC Sample Flows.
- For V2 calls reference along with sample URLs see: Calls Reference - v2.

The Commerce Web Services use basic authentication. Each call, even search and get product details, need to send the authentication headers as defined in the basic authentication scheme. The authenticated user needs to be in the customer group to be successfully authenticated.

Note

The `ycommercewebservices` `project.properties` file contains a number of Cross-Origin Resource Sharing (CORS) settings that you can configure to use with a trusted third-party web application. If you are running a production environment, the following `corsfilter` settings need to be adjusted:

```
corsfilter.ycommercewebservices.allowedOrigins=http://localhost:4200 https://localhost:4200
corsfilter.ycommercewebservices.allowedMethods=GET HEAD OPTIONS PATCH PUT POST DELETE
corsfilter.ycommercewebservices.allowedHeaders=origin content-type accept authorization
```

For more information, see Cross-Origin Resource Sharing Support.

OAuth 2.0 Protocol in Commerce Web Services

OAuth 2.0 is the next evolution of the OAuth protocol created in late 2006. OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.

The key benefit of using OAuth 2.0 (compared to basic authentication, even over HTTPS) is that the API client does not have to save or, in some cases, even obtain the user's credentials. Instead of that, access tokens are returned to the client that can use refresh tokens to obtain new access tokens once they have expired.

The SAP Commerce suite does not provide sample OAuth clients, therefore you have to define them on your own. For details see: Configuring OAuth Clients.

Caution

When defining the clients remember to assign either the ROLE_CLIENT or ROLE_TRUSTED_CLIENT to them, because these roles allow client access to the `ycommercewebservices`extension. However, be careful with the ROLE_TRUSTED_CLIENT because it has got specific, extended rights.

Related Information

SAP Commerce Accelerator Documentation
OAuth 2.0
OCC AddOns Architecture

Cart Validation for OCC

OCC (Omni Commerce Connect) uses the same cart validation as the accelerator. Validation includes not only checks for sufficient stock, but also all registered validation hooks. The logic used for validation may immediately modify the cart to ensure that its state is consistent. For example, if sufficient stock for a product is unavailable, the quantity is reduced to match the available stock or if no stock is available, the item is removed from the cart entirely.

Note

You can use a simplified validation that checks only for sufficient stock but does not check for registered hooks or immediately modify the cart. If you wish to use the simplified validation, replace:

```
<bean id="defaultCommerceWebServicesCartService" parent="defaultCommerceCartService">

  <property name="cartValidationStrategy" ref="defaultCartValidationStrategy"/>

  <property name="productConfigurationStrategy" ref="productConfigurationStrategy"/>

</bean>
```

with

```
<bean id="defaultCommerceWebServicesCartService" parent="defaultCommerceCartService">
  <property name="cartValidationStrategy" ref=" cartValidationWithoutCartAlteringStrategy"/>
  <property name="productConfigurationStrategy" ref="productConfigurationStrategy"/>
</bean>
```

Cart Validation Before Checkout

The API endpoint `/{baseSiteId}/users/{userId}/carts/{cartId}/validate` can be used to validate a cart before checkout to ensure that it is consistent. The endpoint triggers a cart validation and returns the valdiation result. The cart may be modified as part of the validation.

yoccaddon Extension

The `yoccaddon` extension template is a predefined extension that serves as a starting point for creating a new AddOn for Commerce Web Services. The `yoccaddon` extension is essentially an empty extension with minimal implementations needed to create an AddOn.

⚠ Caution

This page refers to software that has been deprecated as part of the Accelerator UI and older OCC template extensions deprecation. For more information, see Deprecation of Accelerator UIs and Older OCC Template Extensions.

Contents of the yoccaddon Extension

The following sections describe the contents of the `yoccaddon` extension.

Impex Files

The `yoccaddon` extension contains the following emtpy impex files:

- `essentialdata_yoccaddon.impex`
- `projectdata_yoccaddon.impex`

These files are copied to the generated AddOn and have the following convention:

```
essentialdata_{addonname}.impex and projectdata_{addonname}.impex
```

The files are automatically imported during the update and initialization.

Web Spring Context Extending Commerce Web Services Context

The `yoccaddon` `yoccaddon-web-spring.xml` file used for extending the web context of Commerce Web Services. This file is located in the extension contains the `resource\yoccaddon\web\spring` directory. It contains the component scan configuration for the controllers package and beans needed for the cache configuration.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="yoccaddonpackage.controllers"/>

        ...

</beans>
```

ⓘ Note

Once the extgen process is completed, the file is adapted to include the generated AddOn name and package.That means the Web Spring Context file is located in `resource\{addonname}\web\spring\{addonname}-web-spring.xml`

Template File for Properties

The `yoccaddon` extension contains the `project.properties.template` file that is necessary for AddOns. This file is a template for the `project.properties` file, which is generated during the addOn installation process. For details refer to Installing an AddOn for a Specific Storefront.

```
# Specifies the location of the spring context file added automatically to the global platform application context.
yoccaddon.application-context=yoccaddon-spring.xml

# Specifies the location of the spring context file which will be added to commerce web services web context
ycommercewebservices.additionalWebSpringConfigs.yoccaddon=classpath:/yoccaddon/web/spring/yoccaddon-web-spring.xml
```

ⓘ Note

Once the extgen process is completed, the content of the `{addonname}.application-context={addonname}-spring.xml` file is adapted to include the generated AddOn name :

```
ycommercewebservices.additionalWebSpringConfigs.{addonname}=classpath:/{addonname}/web/spring/{addonname}-web-spring.xml
```

Message Bundle

The `yoccaddon` extension also contains `basic.properties`, `basic_en.properties`, `basic_de.properties` files located in `/acceleratoraddon/web/webroot/WEB-INF/messages` directory. In these files addOn can define localized message bundle, which is used in Commerce Web Services.

ⓘ Note

For details refer to Extend Commerce Services.

Cache Configuration

The AddOn can also extend the configuration for `ehcache` used in Commerce Web Services. It can be done in the `ehcache.xml` file located in the `/acceleratoraddon/web/webroot/WEB-INF/cache` directory.

Note

For details refer to Extend Commerce Services.

commercewebservices Extension

The `commercewebservices` extension exposes part of the Commerce facades as REST-based web services, including calls for product search and product details.

This extension is based on the `ycommercewebservices` template.

Note

The `commercewebservices` extension is no longer a template (unlike `ycommercewebservices`), but a regular extension.

In `commercewebservices` extension:

- API V1 isn't present
- Web root is `/occ/v2`
- Package names start with `de.hybris.platform.commercewebservices.core`
- Extension name `commercewebservices` is used in code and properties

For more information, see OCC Extension Architecture and ycommercewebservices Extension.

commercewebservicestests Extension

The `commercewebservicestests` extension provides a set of tests written in Groovy that are intended to test commercewebservices REST calls.

This extension is based on `ycommercewebservicestest` extension.

Note

The `commercewebservicestests` extension isn't a template (unlike `ycommercewebservicestest`), but a regular extension.

In `commercewebservicestests` extension:

- API V1 isn't present
- Web root is `/occ/v2`
- Package names start with `de.hybris.platform.commercewebservicestests`
- Extension name `commercewebservicestests` is used in code and properties

For more information, see OCC Extension Architecture and ycommercewebservicestest Extension.

yocc Extension

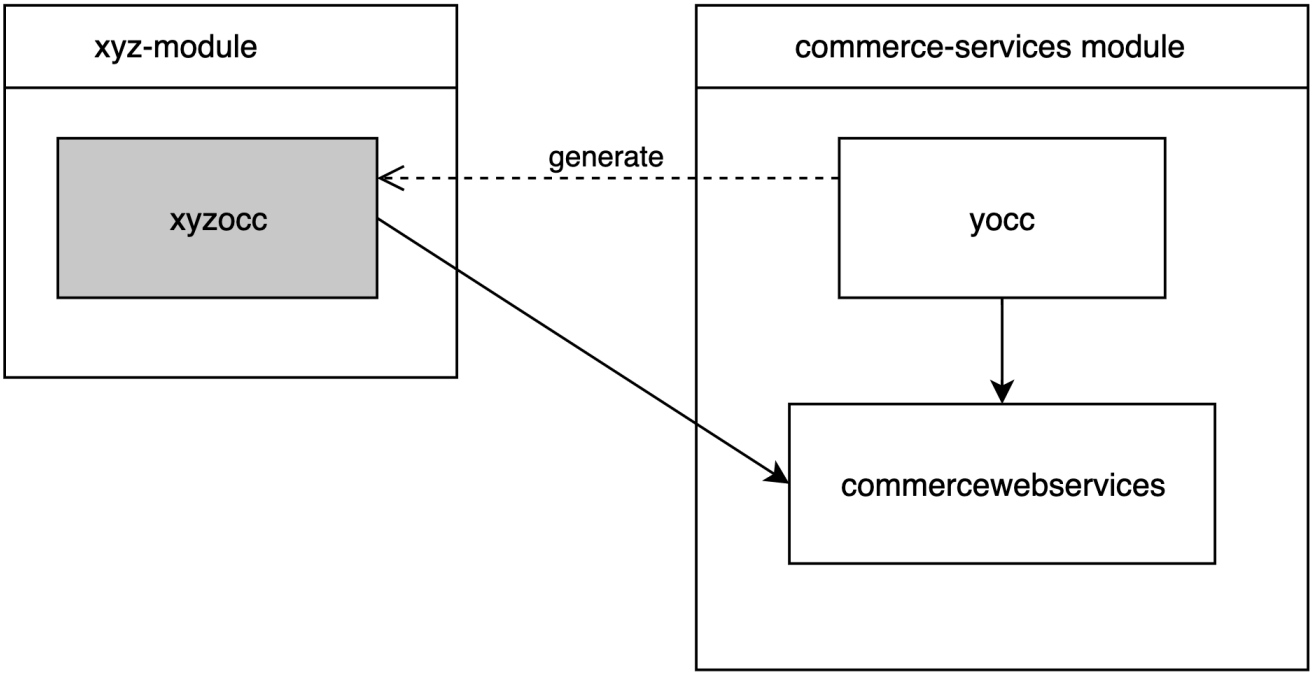
The `yocc` extension template is a predefined extension that serves as a starting point for creating a new extensions for Commerce Web Services.

The `yocc` extension is an empty extension with minimal implementations necessary to create OCC Extension.

Based on `yocc` template, new extension can be generated by providing a new name of the extension that must end with **occ**, for example **xyzocc**.

Dependencies Between Extensions

The `yocc` depends on the `commercewebservices` extension, so that the utility classes can be reused.



Directory Structure of yocc Extension

The directory structure of this extension follows the structure of a regular extension.

Directory / File name	Description
`extensioninfo.xml`	Provides the extension configuration, specifies the dependency on the `commercewebservices` extension.
`project.properties`	Contains configuration properties for the generated OCC Extension.
`/src`	Contains the REST Controller classes and other OCC Extension classes.
`/testsrc`	Contains test controllers for extension generated by the yocc template.
`/resources`	Contains XML files that include: • `{extension_name}-beans.xml` - data model, POJO definitions, • `{extension_name}-spring.xml` - Spring bean definitions, • `{extension_name}-items.xml` - type definitions
`/resources/impex`	The `yocc` extension has no ImpEx files, but to enable the possibility for the newly generated extensions to import data via ImpEx files, they can be stored in this location. ImpEx files in OCC extensions are not loaded automatically unless they follow the convention over the configuration.
`/resources/occ/v2/yocc/messages`	Contains message files.
`/resources/occ/v2/yocc/web/spring/yocc-web-spring.xml`	File that contains Spring bean definitions of the OCC Extension.

For more information, see OCC Extension Architecture.

yocctests Extension

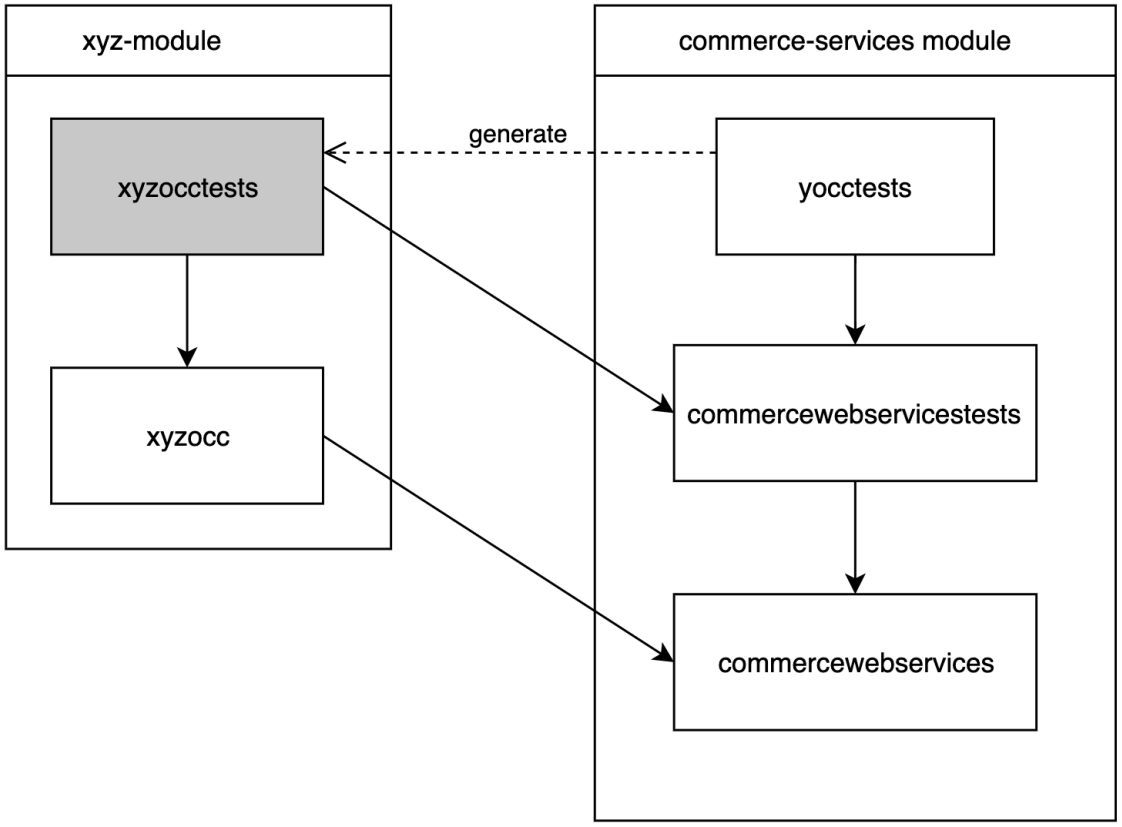
The `yocctests` is a template extension used to generate a new testing extension.

The generated tests extension should provide manual and integration tests that allow for testing all functionality of the OCC Extension.

Dependencies Between Extensions

The `yocctests` extension depends on the `commercewebservicestests`, so that the utility classes can be reused.

The generated tests extension, for example **xyzocctests**, has the dependency on the OCC Extension, for example **xyzocc**.



Generating Custom Tests Extension

The `yocctests` template extension contains files with occurrences of `yocctests` placeholder. The placeholders are replaced during the generating process with the actual name of the OCC Extension.

1. Use `ant extgen` command to generate a custom tests extension based on the `yocctests` template. As a result, you get a tests extension without any tests classes and with the minimal implementation.
2. In the `extensioninfo.xml` add the dependency to the OCC Extension that is to be tested.
3. Customize the tests extension and create the tests and test suites for testing OCC Extension functionality. It is recommended to write tests using the Spock framework.
4. Optionally, add test data to the `essential-data.impex` file, that is automatically imported during the JUnit tenant initialization.

Directory Structure of yocctests Extension

The directory structure of this extension follows the structure of a regular extension.

Directory / File name	Description
`extensioninfo.xml`	Provides the extension configuration, specifies the dependency on the `commercewebservicestests` extension.
`project.properties`	Contains configuration properties for the generated OCC tests extension.
`tenant_junit.properties`	JUnit tenant properties file for the generated tests extension.
`buildcallbacks.xml`	Contains `yocctests_importwstest` data macro definition, called from the `commercewebservicestests` extension when `ant importwstestdata` command is invoked. This macro definition can be used for loading test data of the generated tests extension during `ant importwstestdata` command invocation from the `commercewebservicestests` extension. In the generated extension the `yocctests_importwstest` name of the macro changes to the `{test_extension_name}_importwstestdata` name.
`/src`	Contains the source code files of the extension.
`/testsrc`	The generated extension should contain test and test suite classes. These can be written in Java or Groovy.
`/resources`	Contains XML files that include:

Directory / File name	Description
	• `{test_extension_name}-beans.xml` - data model, POJO definitions, • `{test_extension_name}-spring.xml` - Spring bean definitions, • `{test_extension_name}-items.xml` - type definitions Also contains localizations and ImpEx files with test data.
`/resources/yocctests/groovytests-property-file.groovy`	The configuration file of test environments, used when test data are loaded from `commercewebservicestests` extension using `TestSetupUtils` class.
`/resources/yocctests/log4j.properties`	The configuration file of logging mechanism.
`/resources/yocctests/import/coredata/common`	Contains ImpEx files with test data loaded during the JUnit tenant initialization process.

For more information, see OCC Extension Architecture.

commerceservicesbackoffice Extension

The `commerceservicesbackoffice` extension contains components of Backoffice Administration Cockpit that provide configuration for models and features used in the Commerce Services module.

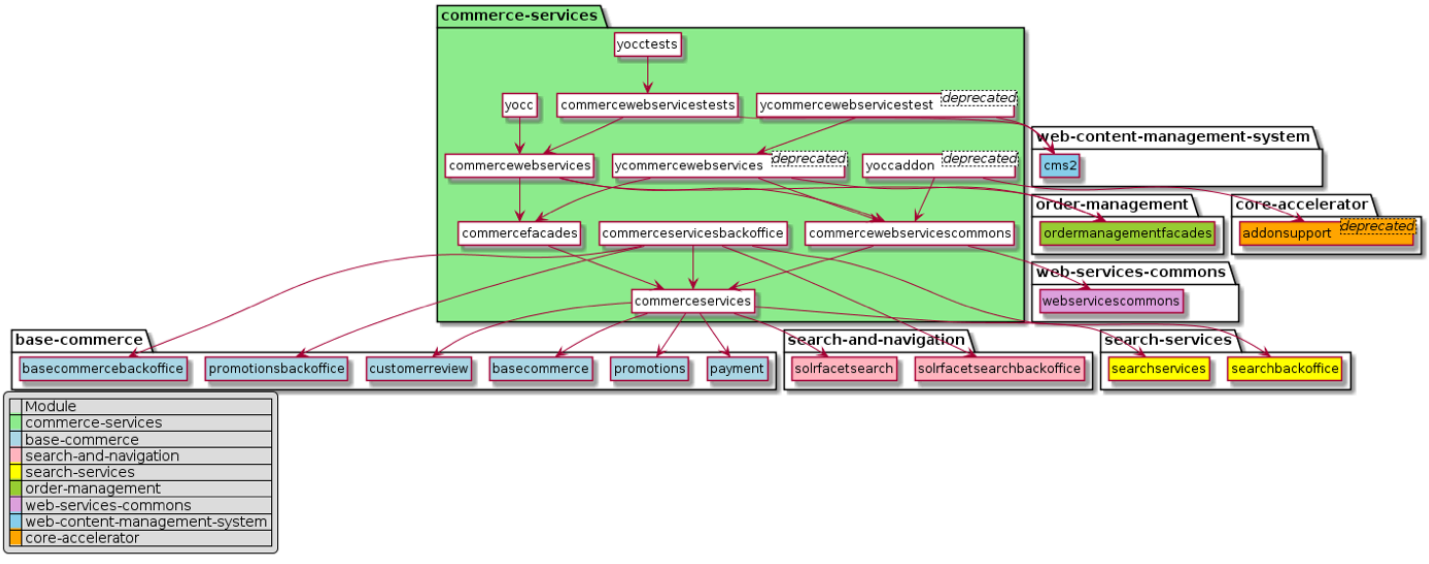
About the Extension

Name	Directory	Related Module
`commerceservicesbackoffice`	`hybris/bin/modules/commerce-services`	Commerce Services Module

The `commerceservicesbackoffice` extension contains widget definitions, editors, and actions that allow you to configure the Sales Organization perspective in Backoffice. The widget with the Sales Organization perspective is defined in the `resources/perspective/organization/organization-backoffice-widgets.xml` file.

The extension contains create wizards such as New Sales Unit, Organization Unit Employee, Consent Template, and Promotion Restriction. It also provides user interface components in Backoffice Administration Cockpit that allow you to configure and search for data models used in the Commerce Services module. These components are defined in the `resources/commerceservicesbackoffice-backoffice-config.xml` file and are localized through property files in the `resources/commerceservicesbackoffice-backoffice-labels` folder.

Dependencies



Dependencies Diagram

Commerce Services Implementation

The section explains how to implement the features of the Commerce Services module including cart management, payment options, and security.

Configurable Products

Finding User by Property

The service `userMatchingService` looks up a user based on various attributes. This service is used by facades and validators to support all places of the Commerce Web Services, where the `userId` comes directly from a URL as a path variable or request parameter. The implementation of this service is

`DefaultUserMatchingService`, which contains a list of strategies of type `UserPropertyMatchingStrategy`.

[Inserting Customer IDs in API Calls]

Inserting a customer ID into an OCC API call allows a service agent to act on behalf of the customer. A typical use case for this feature is emulating customer actions in the Assisted Service Module, which uses OCC APIs to support customers.

[Cart Merging]

The cart merge functionality is intended to provide a consistent cart experience across multiple touchpoints.

[Extending CommerceCartService]

This section describes how to extend `CommerceCartService`.

[Extending CommerceCheckoutService]

This section describes how to extend `CommerceCheckoutService`.

[Converters and Populators]

Data objects are constructed from Models or other Service Layer objects using Converters and Populators. Converters create new instances of Data objects and call Populators to populate these data objects.

[Value Resolvers]

The value resolvers are a more efficient replacement for the current value providers.

[Populating the In-Store Customers List with IoT Device Data]

An Assisted Service agent working physically in a store can look up the customers who are currently in that particular store. At the moment, the information about which customers are present in a given store is mocked and imported via an ImpEx file, but this data may come from any source, including an IoT device.

[Commerce Quotes]

Learn about the classes in `commerceservices` that are used for the Commerce Quotes feature.

[Cart Entry Grouping Functionality]

Entry Grouping functionality allows to group multiple cart entries into single package and operate with it as with one entity.

[Voucher Redemption, Validation, and Brute Force Attack Detection]

Vouchers are redeemed upon placing an order. Previously this functionality was left up to the partner to implement.

[Customizing Commerce Services]

Guidelines on how to customize Commerce Services.

[Integrating with Adobe Experience Manager]

SAP Commerce can easily integrate with Adobe Experience Manager (AEM). This allows you to connect the SAP Commerce solution responsible for controlling product data, shopping carts, checkout and order fulfillment, while AEM controls the data display and marketing campaigns.

[Removing Old Carts with Cronjob]

Carts are not deleted after the session is closed. This way it is still possible to restore the cart later when the user gets back online. However, to avoid storing large amount of old carts you can use the cronjob to remove them.

[Extend Commerce Services]

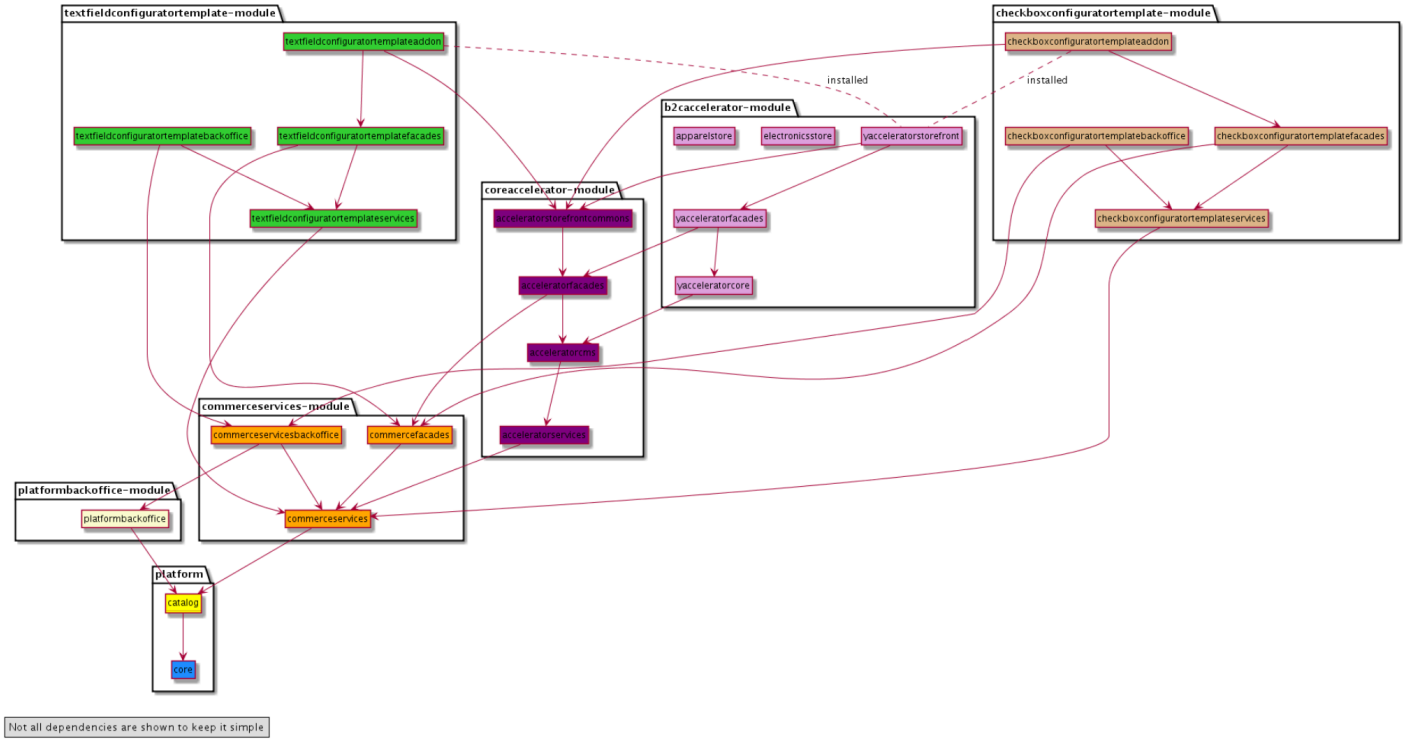
Commerce Services can be extended by creating an AddOn with one additional `Controller` class and using appropriate methods. However, in most cases some more complex changes are required to build an AddOn which adds the functionality to Commerce Services.

Configurable Products

Extension Dependencies

Learn about the extension dependencies for configurable products.

This UML diagram gives a general overview of the most important extension dependencies.

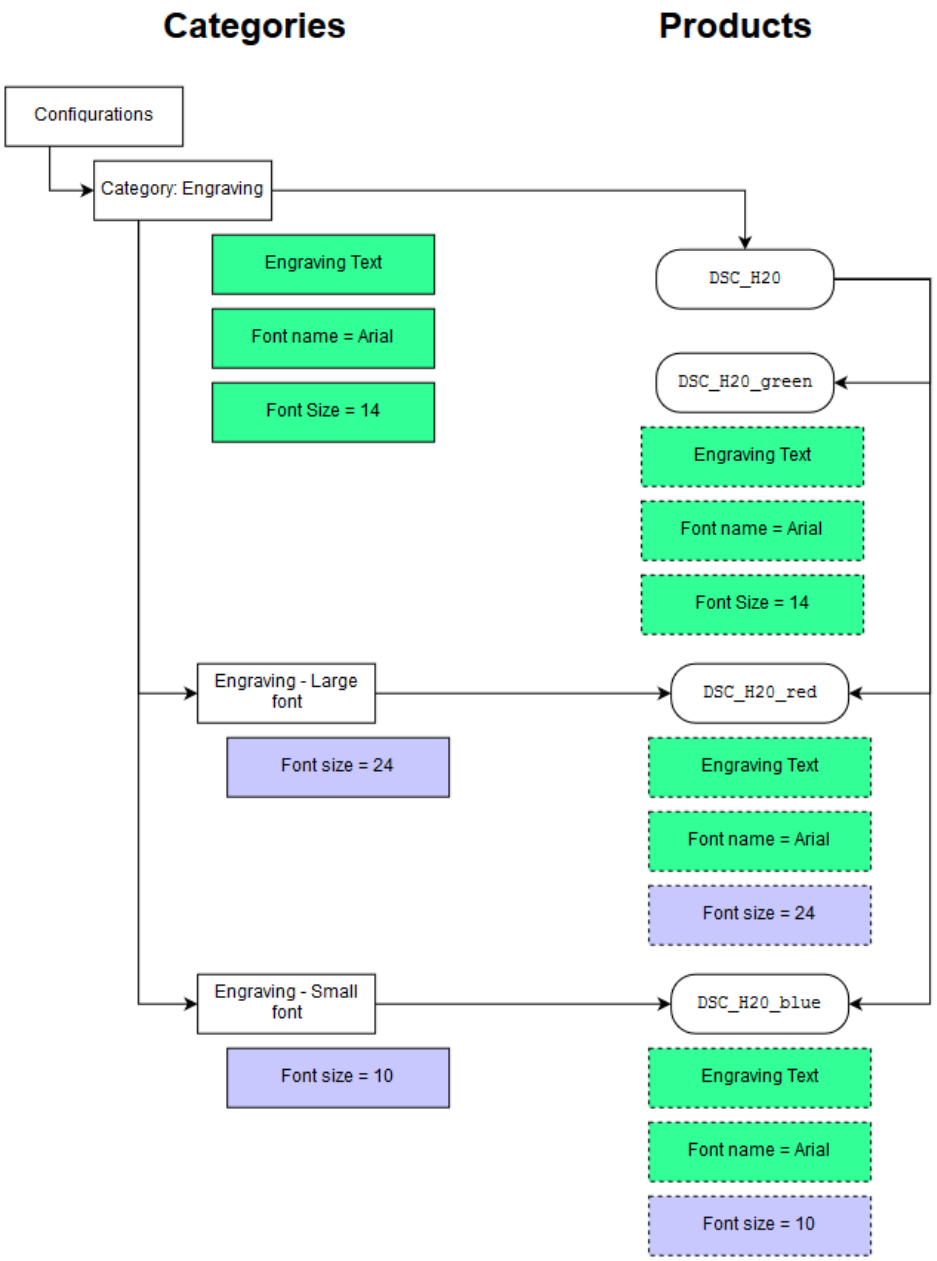


Configurable Category Examples

Depending on the product variant configuration, the variant category can inherit, override, or combine the configuration from a base product category.

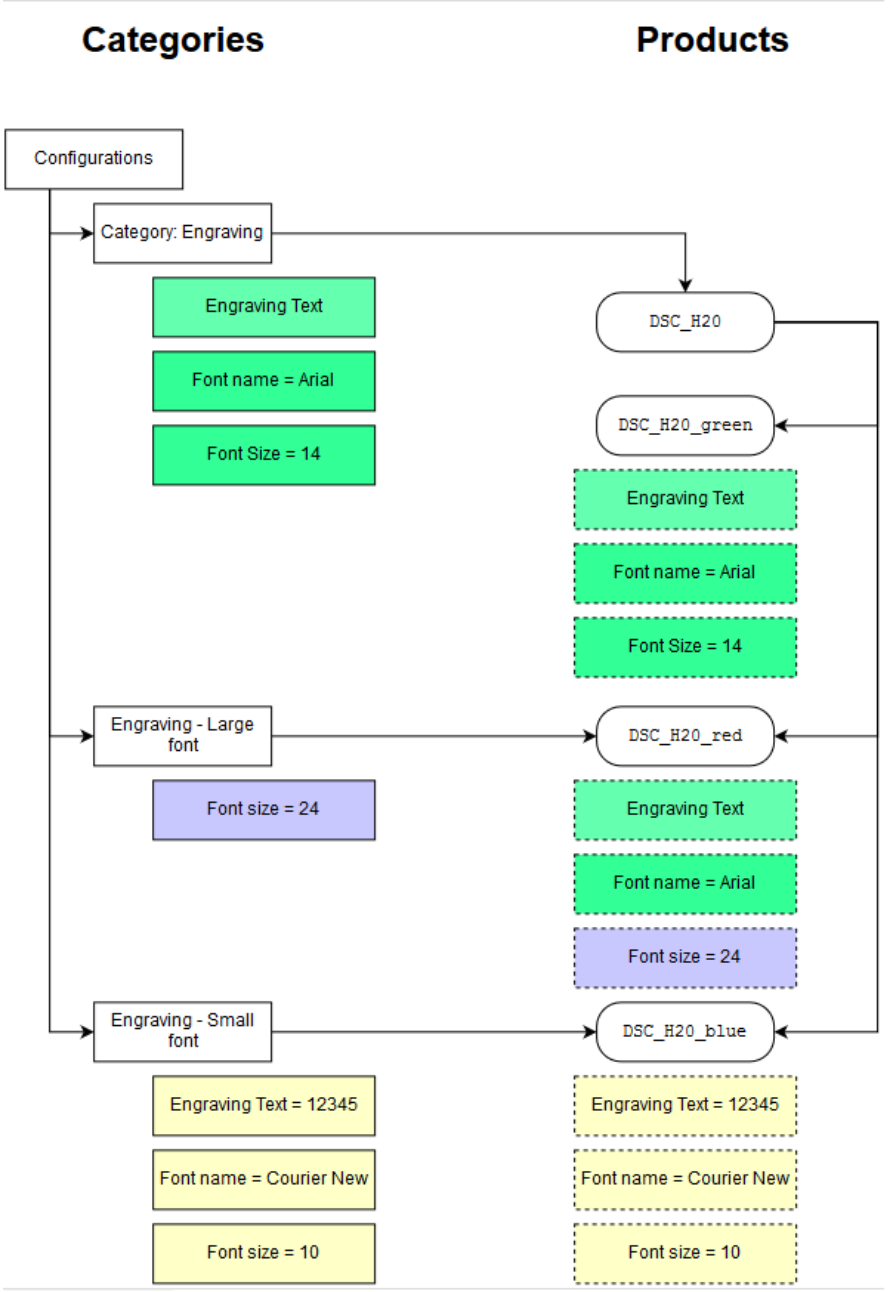
Option 1 - Variant's Category Inherits Configuration and Overrides Some Settings

The parent category is also assigned to the base product, so by default the product variants have the base product's configuration set.



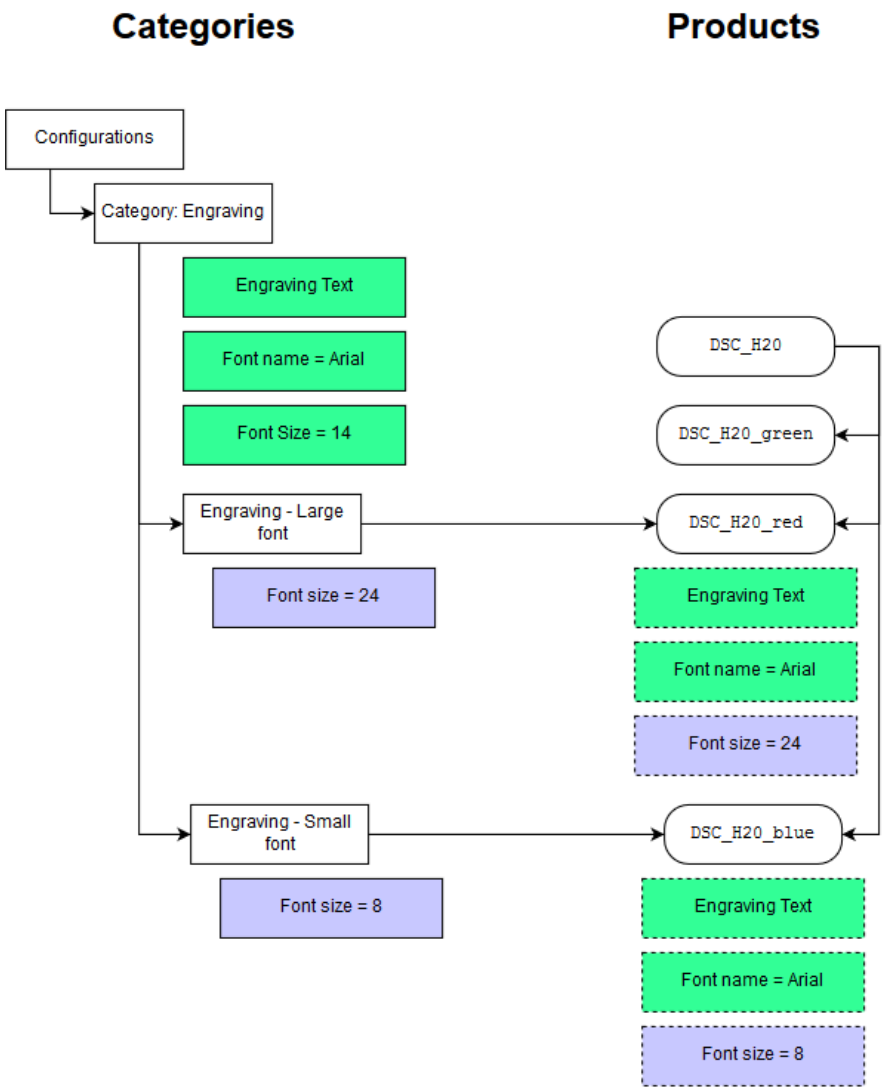
Option 2 - Plain Category Structure

A variant combines the base product's configurations with its own set. This set has a priority in case of conflicts. If conflicts occur when the base product and variant both have a category of the same type, the variant's version completely overrides the base version.



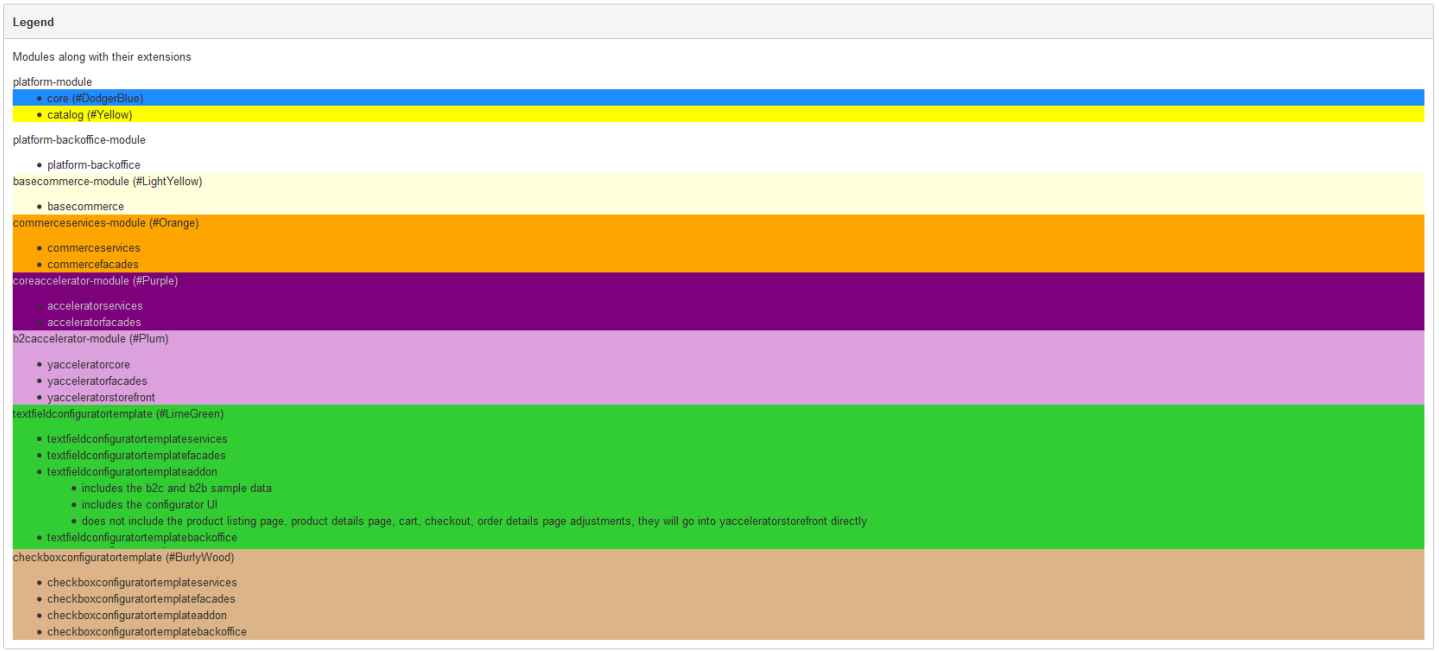
Option 3 - Base Product Does Not Have Configurations

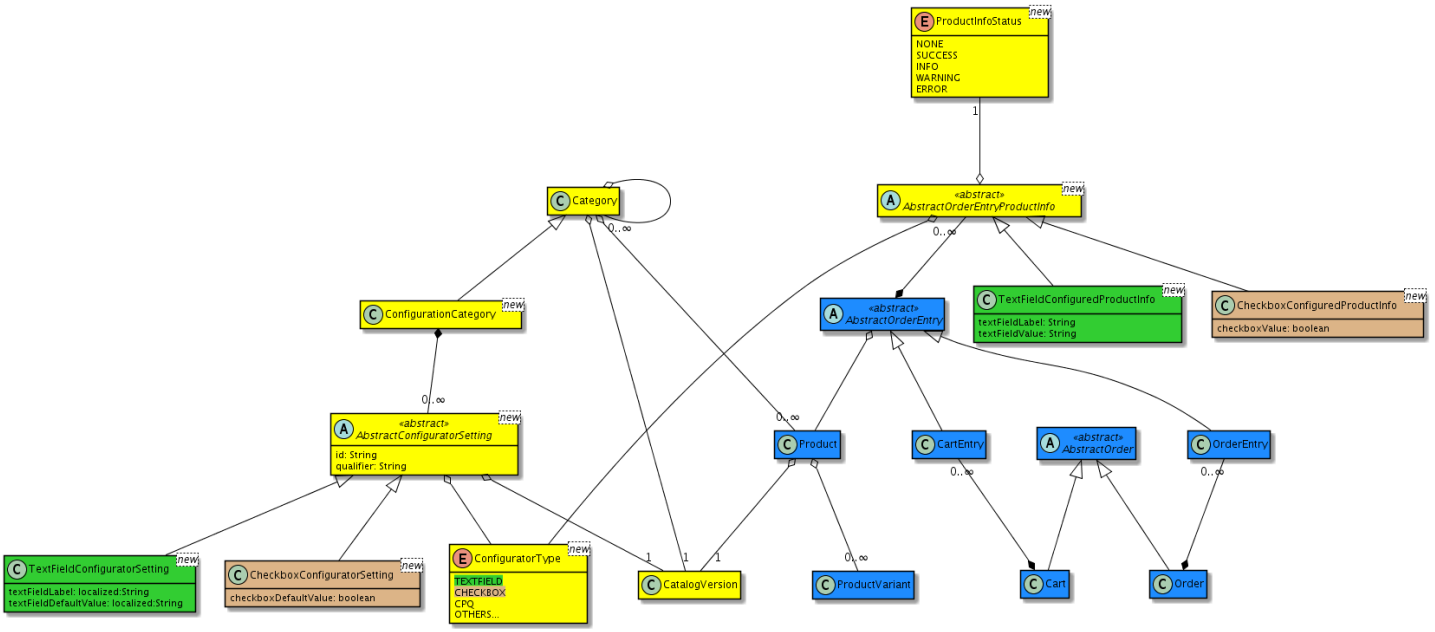
Base product has no configurations and configurations are defined for variants directly. In the example, configurations for the **Engraving** category are merged with variant-specific configurations.



Service Layer

Items

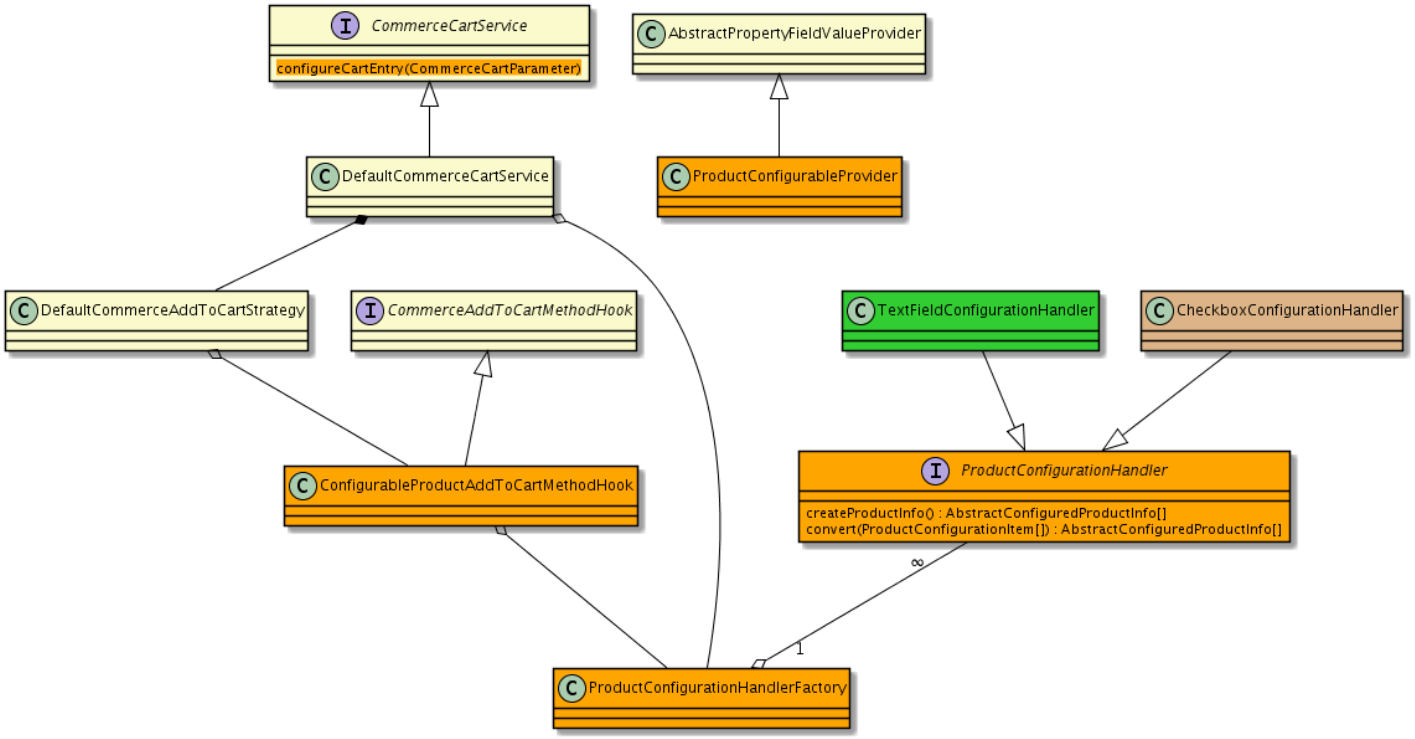




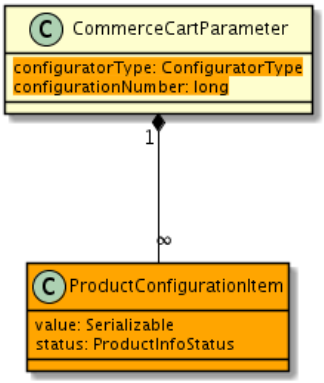
Services

The following changes are introduced to classes and interfaces:

- `CommerceCartService`: adds an extra method *<updateCartEntry>* that can change a single `AbstractConfiguredProductInfo` in a given cart entry.
- `DefaultCommerceAddToCartStrategy`: adds to the new entry instances of `AbstractConfiguredProductInfo` according to the list of `AbstractConfiguratorSettings` provided in a product.
- New interface - `ProductConfigurationHandler`: incapsulates configurator-type-specific logic on the service level. `ProductConfigurationHandlerFactory` dispatches calls to the correct implementation of `ProductConfigurationHandler`.

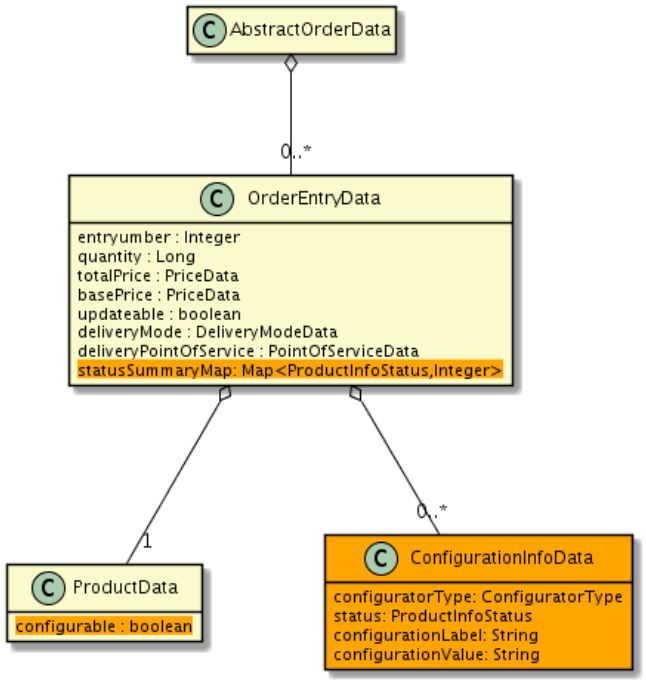


Intermediate DTOs

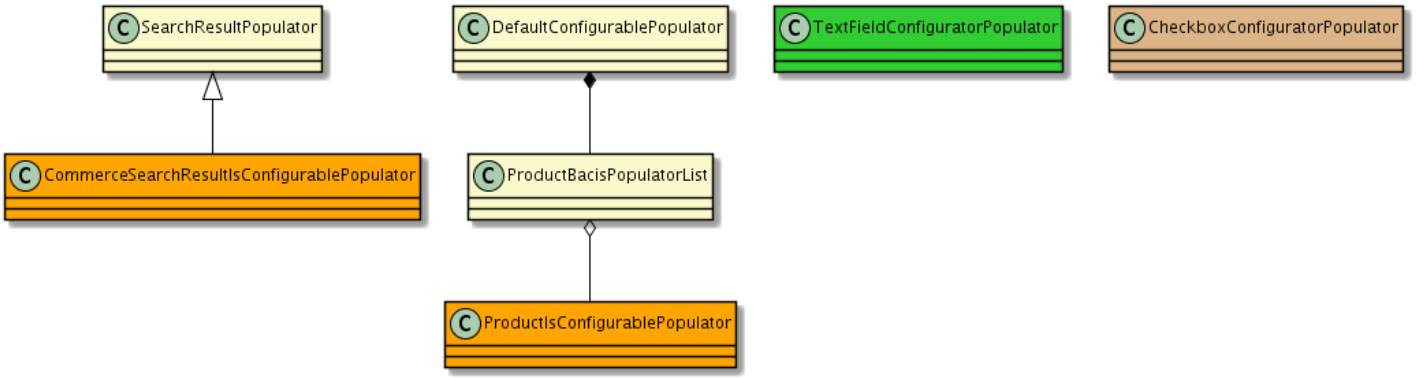


Facade Layer

DTOs



Populators of Facades



Related Information

textfieldconfiguratortemplateservices Extension
textfieldconfiguratortemplatebackoffice Extension
textfieldconfiguratortemplatefacades Extension
textfieldconfiguratortemplateaddon Extension
Configurable Products Trail

Finding User by Property

The service `userMatchingService` looks up a user based on various attributes. This service is used by facades and validators to support all places of the Commerce Web Services, where the `userId` comes directly from a URL as a path variable or request parameter. The implementation of this service is `DefaultUserMatchingService`, which contains a list of strategies of type `UserPropertyMatchingStrategy`.

User matching in Commerce Services

There are two implementations of this strategy interface in Commerce Services:

- `userUIDMatchingStrategy` - this strategy is searching for different kinds of users by the `uid` attribute. It uses the `userService`
- `customerIdMatchingStrategy` - this strategy is searching for customers by the `customerId` attribute. It uses the `customerService`

Each strategy is searching for a user by using matching based on one particular attribute, such as `uid`, `customerId`. This attribute should be unique among the searched users.

The strategy must also ensure that the type of the returned object is correct. If it differs from the type required by the method, the strategy returns an empty `java.util.Optional` result. The matching is performed in the order of the strategies in the list and ends when the first user is found. In case a user is not found for the given strategy, the next strategy from the list is chosen and checked. If more than one user is found for a given attribute within a given strategy, an exception is thrown.

By default, the `userMatchingService` uses only `userUIDMatchingStrategy`.

The list of strategies used by the service can be configured in `userMatchingService` definition in `commerceservices-spring.xml` file.

```
<alias alias="userMatchingService" name="defaultUserMatchingService"/>
<bean id="defaultUserMatchingService" class="de.hybris.platform.commerceservices.user.impl.DefaultUserMatchingService">
    <property name="matchingStrategies">
        <list>
            <ref bean="userUIDMatchingStrategy"/>
        </list>
    </property>
</bean>
```

Customer matching in Commerce Web Services

There are a lot of cases where it is required for a URL to have an attribute `userId` as a path variable or as a request parameter in Commerce Web Services. The system looks up a user based on the given value of the `userId` attribute. In Commerce Services, the identifier for the `userId` attribute is the e-mail address of a user, which is stored as the `uid` property of a user.

Avoid using sensitive data explicitly in the URL of the request. It is possible to configure and use another user attributes as the `userId` parameter in the URL of REST API. One of such attributes is the `customerId` attribute. It is unique among all customers and generated during customer registration. For user matching, use the universally unique identifier, UUID, format for `customerId` attribute.

ⓘ Note

Using `customerId` attribute does not break the existing REST API. The existing use of the e-mail address in the `uid` attribute continues to work. It is also possible to change the configuration to identify a user by a different attribute, such as a phone number. During the registration of a new user and authorization in the OAuth2 server, e-mail is still used as the identifier.

In Commerce Web Services the strategy list includes the strategy `customerIdMatchingStrategy`. It uses a service that implements a `CustomerService` interface. This service allows you to find a customer if the `customerId` is in valid UUID format. Otherwise it returns an empty `java.util.Optional` result. If the customer is not found, then the search process goes ahead with the next matching strategies.

The UUID is validated against a regular expression provided in the service definition.

```
<alias alias="customerService" name="defaultCustomerService"/>
<bean id="defaultCustomerService" class="de.hybris.platform.commerceservices.customer.impl.DefaultCustomerService">
    <constructor-arg ref="customerDao"/>
    <constructor-arg type="java.lang.String" value="^[0-9a-f]{8}-[0-9a-f]{4}-[1-5][0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$"/>
</bean>
```

You can search for customers by `customerId` and then by `uid`. This configuration is used for REST API endpoints where `userId` is taken directly from the path variable or request parameter and used to look up a user. The use of `customerId` helps you to avoid using sensitive data, such as e-mail address, in URLs.

To customize the user search, a new bean for `userMatchingService` is added - `wsUserMatchingService`. The list of strategies in this service is extended by `customerIdMatchingStrategy`.

The order of the strategies on the list matters. The `customerIdMatchingStrategy` appears higher on the list and takes precedence over the `userUIDMatchingStrategy`. It means that the user search process first searches for a customer with the `customerId` attribute. If it can't find such customer, it attempts to find a user with the `uid` attribute.

```
<alias name="defaultWsUserMatchingService" alias="wsUserMatchingService"/>
<bean id="defaultWsUserMatchingService" parent="userMatchingService">
    <property name="matchingStrategies">
        <list>
            <ref bean="customerIdMatchingStrategy"/>
            <ref bean="userUIDMatchingStrategy"/>
        </list>
    </property>
</bean>
```

Facades and validators located in the Commerce Services that inject `userMatchingService` are used not only by Commerce Web Services. There are separate bean definitions of facades and validators in Commerce Web Services that require the use of `wsUserMatchingService`.

The facades are in the `ycommercewebservices-spring.xml` file:

- `wsUserFacade`
- `wsCustomerFacade`
- `wsCustomerGroupFacade`

The validators are in the `validators-v2-spring.xml` file:

- `wsPrincipalListDTOValidator`
- `wsUserGroupDTOValidator`

Examples

The request for getting user details for the electronics store on the local host has the following pattern:

```
GET https://localhost:9200/rest/v2/electronics/users/{userId}
```

The examples assume that the customer exists and its `uid` is **john.doe@test.com** and its `customerId` is 6a2a41a3-c54c-4ce8-a2d2-0324e1c32a21.

There are two examples of using the `userId` URL parameter: as a user `uid` or as a `customerId` parameter of the customer.

- `uid` as `userId`:

  When you use the `uid` attribute of a user as `userId` URL parameter, the effective URL is:

  ```
  https://localhost:9002/rest/v2/electronics/users/john.doe@test.com
  ```

  On the server side, the `wsUserMatchingService` first tries to find the customer by using the `customerIdMatchingStrategy`, but it fails due to an invalid UUID format of the `customerId` attribute. At the next attempt, the `userUIDMatchingStrategy` is used and it finds the customer.

- `customerId` as `userId:`

  When you use the `customerId` attribute of a user as `userId` URL parameter, the effective URL is:

  ```
  https://localhost:9002/rest/v2/electronics/users/6a2a41a3-c54c-4ce8-a2d2-0324e1c32a21
  ```

  On the server side, the `wsUserMatchingService` finds the customer at the first attempt using the `customerIdMatchingStrategy`.

Inserting Customer IDs in API Calls

Inserting a customer ID into an OCC API call allows a service agent to act on behalf of the customer. A typical use case for this feature is emulating customer actions in the Assisted Service Module, which uses OCC APIs to support customers.

A service agent emulating a customer must have the CUSTOMERMANAGERGROUP role, which is granted using OAuth2 authentication. To emulate the customer, the customer ID is provided in the API request. This correctly identifies them, and assigns them as the current user for the API call. How you insert the customer ID depends on the type of API call.

Endpoint URLs That Include the Customer ID

For endpoint URLs that include the customer ID, for example when a service agent fetches user carts, provide the customer ID directly in the request path parameter as follows:

```
occ/v2/{baseSiteId}/users/{customerId}/carts
```

In this case, the customer is always identified with the request path parameter, and any customer ID provided in HTTP headers is ignored.

Endpoint URLs That Don't Include the Customer ID

For endpoints that do not include the customer ID in the request path, for example when service agent is searching products on behalf the customer, provide the customer ID in the HTTP headers using the `sap-commerce-cloud-user-id` field as follows:

```
sap-commerce-cloud-user-id: {customerId}
```

Cart Merging

The cart merge functionality is intended to provide a consistent cart experience across multiple touchpoints.

The following is a typical use case for cart merging: a customer is logged in to a webstore and saves items to their cart. At a later point, the customer returns to the webstore without logging in, and decides to add items to a new cart. Now, if the customer decides to complete their purchase, they must log in to their account, and at this point, the cart merge feature ensures that the items previously saved in the customer's cart are merged with the items the customer just added before logging in.

Cart Merging Strategy

The functionality is implemented in the a strategy called **`CommerceCartMergingStrategy`**, which was added to the `commerceservices` extension. It consists of one method, **`mergeCarts`**. The method requires the following parameters:

- `fromCart` - this parameter indicates from which cart to merge entries.
- `toCart` - this parameter indicates to which cart to merge entries.
- `modifications` - this parameter indicates a list of modifications to which new modifications, created by the merging strategy, will be added

CommerceCartMergingStrategy.java

```
/**
                * A strategy for merging carts.
                */
                public interface CommerceCartMergingStrategy
                {
                /**
                * Merge two carts and add modifications
                *
                * @param fromCart
                *           - Cart from merging is done
                * @param toCart
                *           - Cart to merge to
                * @param modifications
                *           - List of modifications
                * @throws CommerceCartMergingException
                */
                void mergeCarts(CartModel fromCart, CartModel toCart, List<CommerceCartModification> modifications) throws CommerceCar
                }
```

The default implementation iterates on each cart entry of a **`CartModel`** object and adds it to the target **`CartModel`** object. It uses the **`CommerceAddToCartStrategy`** to add a cart entry, so all functionality, such as checking stock levels, is included. While merging, the strategy creates a list of modifications made to the target cart.

ℹ Note

The default implementation is also removing the cart from which merging was done, so using it will be impossible after merging.

Application in commercefacades

In `commercefacades`, cart merging is implemented using the following methods: `restoreCartAndMerge` and `restoreAnonymousCartAndMerge`. Both methods merge carts using the **`CommerceCartMergingStrategy`**, and both methods perform a cart restoration after the carts have been merged. The only difference between the methods is that `restoreCartAndMerge` requires, as parameters, two carts that belong to the same user, while the `restoreAnonymousCartAndMerge` method requires the parameter of an anonymous cart to merge from.

Extending CommerceCartService

This section describes how to extend `CommerceCartService`.

ℹ Note

An SAP Commerce extension may provide functionality that is licensed through different SAP Commerce modules. Make sure to limit your implementation to the features defined in your contract license. In case of doubt, please contact your sales representative.

Parameter Object

The parameter object `CommerceCartParameter` is the only parameter for most of the methods in `CommerceCartService`. This object is defined as a bean in `/resources/commerceservices-beans.xml`, as follows:

/resources/commerceservices-beans.xml

```
<bean class="de.hybris.platform.commerceservices.service.data.CommerceCartParameter">
                                <property name="cart" type="de.hybris.platform.core.model.order.CartModel">
                                <description>The CartModel</description>
                                </property>
                                <property name="product" type="de.hybris.platform.core.model.product.ProductModel">
                                <description>The ProductModel</description>
                                </property>
                                <property name="quantity" type="long">
                                <description>The quantity to add</description>
                                </property>
                                <property name="unit" type="de.hybris.platform.core.model.product.UnitModel">
                                <description>The units to add</description>
                                </property>
                                ...
                                </bean>
```

The client of this service does not have to overload the methods when there is a need to pass their own parameters to an existing method. Instead, you add another property to `CommerceCartParameter` in the `bean.xml` file of your extension. The following is an example:

beans.xml

```
<bean class="de.hybris.platform.commerceservices.service.data.CommerceCartParameter">
                                <property name="customProperty" type="String">
                                <description>My custom property</description>
                                </property>
                                ...
                                </bean>
```

Default Strategies

As a rule of thumb, each of the methods of the `CommerceCartService` delegates the work to a default strategy. These strategies are found in the `de.hybris.platform.commerceservices.order.impl` package, as follows:

Some of the default strategies are:

```
DefaultCommerceCartCalculationStrategy
                                DefaultCommerceAddToCartStrategy
                                DefaultCommercePaymentAuthorizationStrategy
                                DefaultCommercePaymentInfoStrategy
                                DefaultCommerceCartRestorationStrategy
                                DefaultCommercePlaceOrderStrategy
                                DefaultCommerceCartHashCalculationStrategy
                                DefaultCommerceUpdateCartEntryStrategy
                                DefaultCommercePaymentProviderStrategy
                                DefaultCommerceDeliveryAddressStrategy
                                DefaultCommerceCartSplitStrategy
                                DefaultCommerceCartEstimateTaxesStrategy
                                DefaultCommerceDeliveryModeStrategy
                                DefaultCommerceDeliveryModeValidationStrategy
                                DefaultCheckoutCartCalculationStrategy
                                DefaultCommerceRemoveEntriesStrategy
```

Method Hooks

It is sometimes necessary to have a hook into a method where you want to execute some custom logic before the method, after the method, and perhaps before some other event in the method.

The following are some of the methods in the `CommerceCartService` support hooks that are found in the `de.hybris.platform.commerceservices.order.hook` package:

Hook Interfaced Provided Out of the Box

```
AthorizePaymentMethodHook
                                CommerceAddToCartMethodHook
                                CommerceCartCalculationMethodHook
                                CommercePlaceOrderMethodHook
```

CommerceAddToCartMethodHook

The following code block shows the `CommerceAddToCartMethodHook` interface definition, which you need to implement to hook into the `addToCart` method.

`de.hybris.platform.commerceservices.order.hook.CommerceAddToCartMethodHook`

```
public interface CommerceAddToCartMethodHook
                                        {
                                        void beforeAddToCart(CommerceCartParameter parameters) throws CommerceCartModificationExceptic
                                        void afterAddToCart(CommerceCartParameter parameters, CommerceCartModification result) throws
                                        }
```

Spring Configuration for Method Hooks

The bean with id `commerceAddToCartMethodHooks` is defined in `/resources/commerceservices-spring.xml`, as follows:

```
<util:list id="commerceAddToCartMethodHooks" value-type="de.hybris.platform.commerceservices.order.hook.CommerceAddToCartMethodHook"/>
```

You have to implement the `CommerceAddToCartMethodHook` interface and alias the `commerceAddToCartMethodHooks` bean with your implementation. You can provide multiple hooks if you wish.

You could use `ListMergeDirective` to add elements to the `commerceAddToCartMethodHooks` bean and control their order if necessary. For more information, see commerceservices Extension.

```
<bean id="customCommerceAddToCartMethodHookMergeDirective" depends-on="commerceAddToCartMethodHooks" parent="listMergeDirective">
                                        <property name="add" ref="customCommerceAddToCartMethodHook"/>
                                        </bean>
```

The configuration above would add the `customCommerceAddToCartMethodHook` bean to the `commerceAddToCartMethodHooks` list.

What this will do is for every call to `de.hybris.platform.commerceservices.order.impl.DefaultCommerceCartService#addToCart(CommerceCartParameter)` the method `CommerceAddToCartMethodHook#` `beforeAddToCart` will be executed before `addToCart`,and `CommerceAddToCartMethodHook#afterAddToCart` will run after.

Disabling Method Hooks at Runtime

If for any reason you don't want the wired up method hooks to execute at runtime, you can set the `de.hybris.platform.commerceservices.service.data.CommerceCartParameter#enableHooks` to false when you are calling methods of the service.

However for this you would need to overwrite the facade method which is calling the service method. Another way to disable the hooks is to set a property key to `false` following the convention

`commerceservices.<NAME OF HOOK INTERFACE>.enabled` for example `commerceservices.commerceaddtocartmethodhook.enabled`

By default hooks are enabled.

property keys to disable or enabled method hooks

```
commerceservices.authorizepaymentmethodhook.enabled=true
                                        commerceservices.commerceaddtocartmethodhook.enabled=true
                                        commerceservices.commercecartcalculationmethodhook.enabled=true
                                        commerceservices.commerceplaceordermethodhook.enabled=true
                                        commerceservices.commerceupdatecartentryhook.enabled=true
```

Validators

AddToCartValidator

This interface allows you to define specific validations that are performed during the “add to cart” process. It contains the following methods:

```
boolean supports(final CommerceCartParameter parameter);

void validate(final CommerceCartParameter parameter) throws CommerceCartModificationException;
```

The specific validation in the `validate` method is invoked if the `supports` method returns `true` for the given parameters. Otherwise, it is ignored. The `validate` method throws `CommerceCartModificationException` if the parameters are not accepted. If there is no exception, the parameters are accepted and the process of adding to the cart continues.

To register a custom cart validator, proceed as follows:

1. Create a new class that implements `AddToCartValidator`.
2. Register the class as a bean in `<yourextension>-spring.xml` as follows:

   ```
   <bean id="customBeanId"
       class="com.custom.validators.impl.CustomValidator">
   </bean>
   ```

3. Add it to the list of validators by using the merge directive, referring to the list `addToCartValidators`.

   ```
   <bean id="customAddToCartValidatorsMergeDirective" depends-on="addToCartValidators"
       parent="listMergeDirective">
       <property name="add" ref="customBeanId" />
   </bean>
   ```

Related Information

Accelerators

Extending CommerceCheckoutService

This section describes how to extend `CommerceCheckoutService`.

i Note

An SAP Commerce extension may provide functionality that is licensed through different SAP Commerce modules. Make sure to limit your implementation to the features defined in your contract license. In case of doubt, please contact your sales representative.

Parameter Object

The parameter object `CommerceCheckoutParameter` is the only parameter for most of the methods in `CommerceCheckoutService`. This object is defined as a bean in `/resources/commerceservices-beans.xml`, as follows:

`/resources/commerceservices-beans.xml`

```
<bean class="de.hybris.platform.commerceservices.service.data.CommerceCheckoutParameter">
                                <property name="cart" type="de.hybris.platform.core.model.order.CartModel">
                                <description>The CartModel</description>
                                </property>
                                <property name="address" type="de.hybris.platform.core.model.user.AddressModel">
                                <description>the Address</description>
                                </property>
                                <property name="isDeliveryAddress" type="boolean">
                                <description>Delivery address flag</description>
                                </property>
                                <property name="deliveryMode" type="de.hybris.platform.core.model.order.delivery.DeliveryModeModel">
                                <description>The delivery mode</description>
                                </property>
                                <property name="paymentInfo" type="de.hybris.platform.core.model.order.payment.PaymentInfoModel">
                                <description>payment information</description>
                                </property>
                                ...
                                </bean>
```

The client of this service does not have to overload the methods when there is a need to pass their own parameters to some existing method. Instead, you add another property to `CommerceCheckoutParameter` in the `bean.xml` file of your extension. The following is an example:

`beans.xml`

```
<bean class="de.hybris.platform.commerceservices.service.data.CommerceCheckoutParameter">
                                <property name="customProperty" type="String">
                                <description>My custom property</description>
                                </property>
                                ...
                                </bean>
```

Default Strategies

As a rule of thumb, each of the methods of the `CommerceCartService` delegates the work to a default strategy. These strategies are found in the `de.hybris.platform.commerceservices.order.impl` package, as follows:

Some of the default strategies are:

```
DefaultCommerceCartCalculationStrategy
                                DefaultCommerceAddToCartStrategy
                                DefaultCommercePaymentAuthorizationStrategy
                                DefaultCommercePaymentInfoStrategy
                                DefaultCommerceCartRestorationStrategy
                                DefaultCommercePlaceOrderStrategy
                                DefaultCommerceCartHashCalculationStrategy
                                DefaultCommerceUpdateCartEntryStrategy
                                DefaultCommercePaymentProviderStrategy
                                DefaultCommerceDeliveryAddressStrategy
                                DefaultCommerceCartSplitStrategy
                                DefaultCommerceCartEstimateTaxesStrategy
                                DefaultCommerceDeliveryModeStrategy
                                DefaultCommerceDeliveryModeValidationStrategy
                                DefaultCheckoutCartCalculationStrategy
                                DefaultCommerceRemoveEntriesStrategy
```

Method Hooks

It is sometimes necessary to have a hook into a method where you want to execute some custom logic before the method, after the method, and perhaps before some other event in the method.

The following are some of the methods in the `CommerceCheckoutService` support hooks that are found in the `de.hybris.platform.commerceservices.order.hook` package:

Hook Interfaced provided Out of the Box

```
AthorizePaymentMethodHook
                                CommercePlaceOrderMethodHook
```

CommercePlaceOrderMethodHook

The following code block shows the `CommercePlaceOrderMethodHook` interface definition, which you need to implement to hook into the `placeOrder` method.

`de.hybris.platform.commerceservices.order.hook.CommercePlaceOrderMethodHook`

```
public interface CommercePlaceOrderMethodHook
                                        {
                                        void afterPlaceOrder(CommerceCheckoutParameter parameter, CommerceOrderResult orderModel);
                                        void beforePlaceOrder(CommerceCheckoutParameter parameter);
                                        void beforeSubmitOrder(CommerceCheckoutParameter parameter, CommerceOrderResult result);
                                        }
```

Spring Configuration for Method Hooks

The bean with id `commercePlaceOrderMethodHooks` is defined in `/resources/commerceservices-spring.xml`, as follows:

```
<util:list id="commercePlaceOrderMethodHooks" value-type="de.hybris.platform.commerceservices.order.hook.CommercePlaceOrderMethodHook"
```

You have to implement the `CommercePlaceOrderMethodHook` interface and alias the `commercePlaceOrderMethodHooks` bean with your implementation. You can provide multiple hooks if you wish.

You could use `ListMergeDirective` to add elements to the `commercePlaceOrderMethodHooks` bean and control their order if necessary. For more information, see commerceservices Extension.

```
<bean id="customCommercePlaceOrderMethodHooksMergeDirective" depends-on="commercePlaceOrderMethodHooks" parent="listMergeDirective">
                                        <property name="add" ref="customCommercePlaceOrderMethodHook"/>
                                        </bean>
```

The configuration above would add the `customCommercePlaceOrderMethodHook` bean to the `commercePlaceOrderMethodHooks` list bean.

What this will do is for every call to
`de.hybris.platform.commerceservices.order.impl.DefaultCommerceCheckoutService#placeOrder(CommerceCheckoutParameter)`:

- The `beforePlaceOrder(CommerceCheckoutParameter parameter)` will get executed before the `placeOrder` method
- The `afterPlaceOrder(CommerceCheckoutParameter parameter, CommerceOrderResult orderModel)` will get executed in the `finally` block of the `placeOrder` method
- The `beforeSubmitOrder(CommerceCheckoutParameter parameter, CommerceOrderResult result)` will get executed just before a call is made to `de.hybris.platform.order.OrderService#submitOrder`

Disabling Method Hooks at Runtime

If for any reason you don't want the wired up method hooks to execute at runtime, you can set the
`de.hybris.platform.commerceservices.service.data.CommerceCartParameter#enableHooks` to false when you are calling methods of the service.

However for this you would need to overwrite the facade method which is calling the service method. Another way to disable the hooks is to set a property key to `false` following the convention

`commerceservices.<NAME OF HOOK INTERFACE>.enabled` for example `commerceservices.commerceaddtocartmethodhook.enabled`

By default hooks are enabled.

property keys to disable or enabled method hooks

```
commerceservices.authorizepaymentmethodhook.enabled=true
                                        commerceservices.commerceaddtocartmethodhook.enabled=true
                                        commerceservices.commercecartcalculationmethodhook.enabled=true
                                        commerceservices.commerceplaceordermethodhook.enabled=true
                                        commerceservices.commerceupdatecartentryhook.enabled=true
```

Related Information

Accelerators

Converters and Populators

Data objects are constructed from Models or other Service Layer objects using Converters and Populators. Converters create new instances of Data objects and call Populators to populate these data objects.

Overview

The **commercefacades** extension provides a suite of facades that make up a unified multichannel storefront API that can be used by multiple front-ends. The facade's responsibility is to integrate existing business services from the full range of the hybris extensions and expose a Data object (POJO) response adjusted to meet the storefront requirements

Approach

The outline approach should be:

- No concrete Converters should be written, all converters should be Spring configured only and should use the **AbstractConverter** base class.
- No Populator should be called directly in code, Converters should be Spring injected and used.
- All conversion logic should exist in Populators and these should be well-encapsulated and independent.

Converters

Converters create new instances of Data objects and call Populators to populate these. The Data object is always created from a prototype-scoped spring bean that is defined in the **`beans.xml`** file for the extension.

Populators

Populators break the conversion process of filling out a Data Object down into a pipeline of population tasks or steps. Each Populator carries out one or more related updates to the Data Object prototype. Each population step can invoke services or copy data from the source business object to the prototype Facade Data object. Facades always use a Converter to create a new instance of a Data Object prototype and then invoke Populators or other Converters to fulfill the task of building up the Data Object.

Configurable Populators and Data Options

Configurable Populators extend the concept of Populators (not the Java interface) by allowing a collection of Enum type data options to be provided. The Configurable Populator then invokes only those Populators that add data for the given data options. This functionality may appear complex but is vital from a performance and bandwidth perspective. The point here is to only convert what is required when it is required by specifying exactly which populators should be used programmatically.

Converting the entire object may be costly from a performance point of view. For example consider evaluating promotions for a product unnecessarily or attaching classification data when not required. Also, for example, the product description can be rather sizable, and therefore returning it in a web service call is not normally necessary. Configurable Populators also allow for extra population logic to be injected into a population pipeline without the need to replace or override existing code.

ℹ Note

When the `ConfigurablePopulators` mechanism was moved from `commercefacades` extension into `platformservices` extension to make it available for all other extensions, this change included the following alterations:

- The package of configurable populators was changed from `de.hybris.platform.commercefacades.converter` to `de.hybris.platform.converters`.
- The code was also refactored and simplified.
- The classes in `de.hybris.platform.commercefacades.converter` package still exist, but are currently deprecated.

Modifiable Configurable Populators

The modifiable configurable populators feature enables you to modify **ConfigurablePopulators** that implement the **ModifiableConfigurablePopulator** interface. This includes adding populators to, and removing populators from already configured **ConfigurablePopulator** beans.

Modifications are configured as Spring beans that extend the abstract parent **configurablePopulatorModification** bean. They register themselves with the target populator during context initialization.

The **`commercefacades-spring.xml`** file contains the definition of the abstract **configurablePopulatorModification** parent bean, which defines the init method that executes the registration with the target populator, as follows:

```
<!-- Abstract bean used as a parent for beans that modify a ModifiableConfigurablePopulator. -->
<bean id="configurablePopulatorModification" class="de.hybris.platform.converters.config.ConfigurablePopulatorModification" abstract="
```

Sample Usages of Modifiable Configurable Populators

The following is an example of adding a populator:

```
<bean parent="configurablePopulatorModification">
        <property name="target" ref="defaultProductConfiguredPopulator" />
        <property name="keyType" value="de.hybris.platform.commercefacades.product.ProductOption" />
        <property name="key" value="VOLUME_PRICES" />
        <property name="add" ref="productVolumePricesPopulator" />
</bean>
```

The following is an example of removing a populator:

```
<bean parent="configurablePopulatorModification">
        <property name="target" ref="defaultProductConfiguredPopulator" />
```

```
        <property name="keyType" value="de.hybris.platform.commercefacades.product.ProductOption" />
        <property name="key" value="VOLUME_PRICES" />
        <property name="remove" ref="productVolumePricesPopulator" />
</bean>
```

The **ConfigurablePopulatorModification** type is able to resolve the following key types:

- java.lang.String
- java.lang.Class
- enum types (such as the `de.hybris.platform.commercefacades.product.ProductOption` enum type shown in the two examples above)

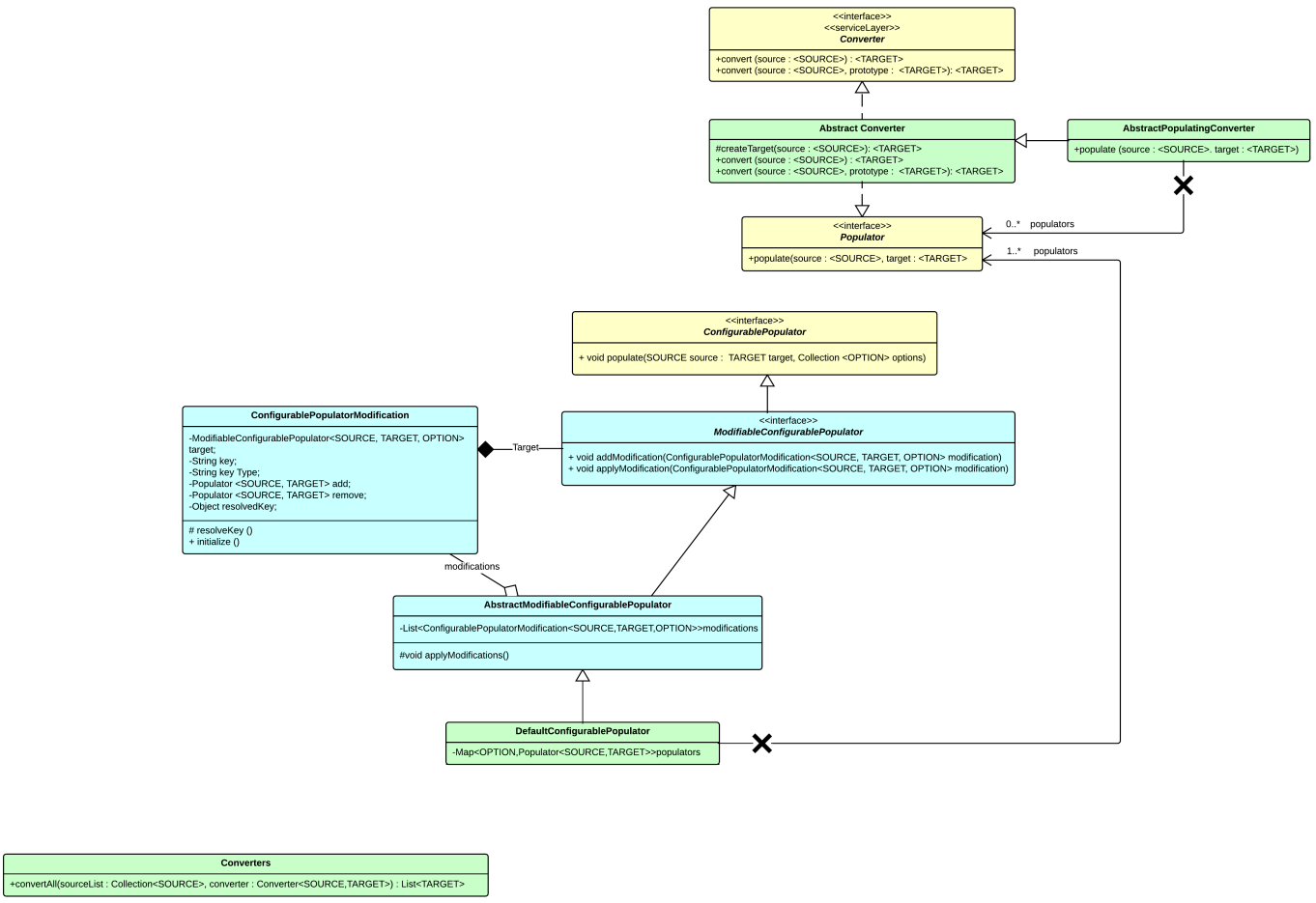
At runtime, when the populate method of a **ModifiableConfigurablePopulator** bean is called for the first time, it first applies all modifications registered to its ancestors (that is, the parent bean, the parent's parent bean, and so on) before it applies all modifications registered to itself. The implementation makes sure that the modifications are only applied one single time per bean instance.

Base Classes

Below you can find the description of Populators and Converters base classes:

- Since **Converters** can populate data objects and therefore can also be deemed **Populators**, **AbstractConverter** is a convenience base class that implements both the **Converter** and **Populator** interface and provides the basic implementation of the **Converter** interface delegating to a **createTarget** template method that allows a concrete sub-class to pick the appropriate target data object implementation.
- **AbstractPopulatingConverter** simply extends **AbstractConverter** and in addition provides an implementation of **populate** from the **Populator** interface. It is recommended that most Converters extend this base class and simply override the **createTarget** template method and inject the necessary pipeline of Populators to fill out the object graph of the target. This approach leads to the most flexible and adaptable code where much of the behavior can be adapted externally through spring bean configuration.
- **DefaultConfigurablePopulator** is a default implementation of a **Populator** pipeline where each population step is evaluated against a Set of **Enum** values passed by the caller. Each **Populator** in the pipeline will only run if it is mapped to an **Enum** that is contained in the Set passed by the caller. This implementation allows you to modify the list of populators in the pipeline.

The following class diagram presents the **Configurable** and **Modifiable Configurable Populators**.



Related Information

Define Converters and Populators
Accelerators
Using Façades and DTOs - Best Practice

Value Resolvers

The value resolvers are a more efficient replacement for the current value providers.

This document provides information on the value resolvers.

Below you will find a list of value resolvers along with description and supported parameters.

ProductAttributesValueResolver

A resolver for product attributes. It takes into consideration the variant product attributes (however not the generic variant attributes). If a value has not been found for a variant, it tries to get it from the base product. By default, if parameter attribute is not specified, it tries to get the attribute with the same name as the one configured for the indexed property.

Parameter	Default value	Description
optional	true	If false, indicates that the resolved values should not be null and not an empty string (for every qualifier). If these conditions are not met, an exception of type **FieldValueProviderException** is thrown.
attribute		If specified, this is the name of the attribute.
split	false	If true, splits any resolved value around matches of a regular expression (only if the value is of type **String**).
splitRegex	\s+	If split is true this is the regular expression to use.
skipVariants		If true, it ignores product variants and gets the values from the base product.
format		The ID of the Format Bean that is going to be used to format the attribute value object before applying the split

ProductClassificationAttributesValueResolver

A resolver for product classification attributes.

Parameter	Default value	Description
optional	true	If false, indicates that the resolved values should not be null and not an empty string (for every qualifier). If these conditions are not met, an exception of type **FieldValueProviderException** is thrown.
attribute		If specified, this is the name of the attribute.
split	false	If true, splits any resolved value around matches of a regular expression (only if the value is of type **String).**
format		The ID of the Format Bean that is going to be used to format the attribute value object before applying the split

ProductImagesValueResolver

A resolver for product images urls. It selects the first gallery image that supports the requested media format. By default, if parameter mediaFormat is not specified, it tries to retrieve the media format from the indexed property name. By default it finds character "-" and treats the following character sequence as a media format. If character "-" is not present, an exception of type **FieldValueProviderException** is thrown. The parsing method can be overridden.

Parameter	Default value	Description
optional	true	If false, indicates that the resolved values should not be null and not an empty string (for every qualifier). If these conditions are not met, an exception of type **FieldValueProviderException** is thrown.
mediaFormat		If specified, this is the qualifier of the media format.

ProductKeywordsValueResolver

A resolver for product keywords.

Parameter	Default value	Description
optional	true	If false, indicates that the resolved values should not be null and not an empty string (for every qualifier). If these conditions are not met, an exception of type **FieldValueProviderException** is thrown.
split	true	If false, the collected keywords should be added as a single value to the index.
separator	" "	The separator to use when combining the keywords into a single value. Only used if split is false.

ProductPricesValueResolver

A resolver for product prices.

Parameter	Default value	Description
optional	true	If false, indicates that the resolved values should not be null and not an empty string (for every qualifier). If these conditions are not met, a **FieldValueProviderException** is thrown.

ProductUrlsValueResolver

A resolver for product urls.

Parameter	Default value	Description
optional	true	If false, indicates that the resolved values should not be null and not an empty string (for every qualifier). If these conditions are not met, an exception of type **FieldValueProviderException** is thrown.

ProductPromotionAttributesValueResolver

A resolver that gets the product promotion values from attributes on the promotion model. By default, if a parameter attribute is not specified, it tries to get the attribute with the same name as the one configured for the indexed property.

Parameter	Default value	Description
optional	true	If false, indicates that the resolved values should not be null and not an empty string (for every qualifier). If these conditions are not met, an exception of type **FieldValueProviderException** is thrown.
attribute		If specified, this is the name of the attribute.
evaluateExpression	false	If true the attribute name is assumed to be a spring expression language that need to be evaluated

Related Information

Value Provider API
Search & Navigation Module

Populating the In-Store Customers List with IoT Device Data

An Assisted Service agent working physically in a store can look up the customers who are currently in that particular store. At the moment, the information about which customers are present in a given store is mocked and imported via an ImpEx file, but this data may come from any source, including an IoT device.

Context

Note
The below steps present an example reference integration with a physical detection device. The procedure is meant to provide the reader with necessary insight into the technical details of this business case and should not be used as best practice. The actual integration with an IoT device is **not** supported.

To have the In-Store Customers list populated with data coming from an external device, perform the steps below:

Procedure

1. Disable HTTPS and use HTTP for OCC v2 by opening the `hybris/bin/modules/commerce-services/ycommercewebservices/web/webroot/WEB-INF/config/v2/security-v2-spring.xml` file for editing.
2. In the `security-v2-spring.xml` file, locate the `<intercept-url pattern="/**" requires-channel="https"/>` code.
    a. Change the `https` parameter to `http`.
3. In the `hybris/bin/modules/commerce-services/ycommercewebservices/web/webroot/WEB-INF/config/v2` directory, open the `jaxb-converters-spring.xml` file for editing.
    a. Replace the `<util:list id="messageConvertersV2">` bean with the following code:

```
<util:list id="messageConvertersV2">
        <ref bean="customJsonHttpMessageConverter"/>
        <ref bean="customXmlHttpMessageConverter"/>
                <bean class="org.springframework.http.converter.FormHttpMessageConverter" />
                <bean class="org.springframework.http.converter.StringHttpMessageConverter">
                <constructor-arg index="0" name="defaultCharset" value="UTF-8"/>
        </bean>
 </util:list>
```

4. Create a public `LoyaltyDao` interface in the `/hybris/bin/ext-commerce/commerceservices/src/de/hybris/platform/commerceservices/customer/dao/` directory.
    a. In the `LoyaltyDao` interface, define the `findUserByLoyaltyCardId()` method.

```
public interface LoyaltyDao
{
    UserModel findUserByLoyaltyCardId(String loyaltyCardId);
}
```

5. Create a public `DefaultLoyaltyDao` class in the `hybris/bin/ext-commerce/commerceservices/src/de/hybris/platform/commerceservices/customer/dao/impl` directory.
    a. In the `DefaultLoyaltyDao` class, implement the business logic.

```
public class DefaultLoyaltyDao extends DefaultGenericDao<CustomerModel>implements LoyaltyDao
{
    public DefaultLoyaltyDao()
    {
        super(CustomerModel._TYPECODE);
    }
    @Override
    public CustomerModel findUserByLoyaltyCardId(final String loyaltyCardId)
    {
        final List<CustomerModel> resList = find(Collections.singletonMap(CustomerModel.LOYALTYCARDID, loyaltyCardId));
        return resList.isEmpty() ? null : resList.get(0);
    }
}
```

6. Create a public `LoyaltyService` interface in the `hybris/bin/ext-commerce/commerceservices/src/de/hybris/platform/commerceservices/customer/` directory.
    a. In the `LoyaltyService` interface, define the `getUserForLoyaltyCardID()` method.

```
public interface LoyaltyService
{
    UserModel getUserForLoyaltyCardID(String loyaltyCardId);
}
```

7. Create a public `DefaultLoyaltyService` class in the `hybris/bin/ext-commerce/commerceservices/src/de/hybris/platform/commerceservices/customer/impl` directory.
    a. In the `DefaultoyaltyService` class, implement the business logic.

```
public class DefaultLoyaltyService implements LoyaltyService
{
    private LoyaltyDao LoyaltyDao;
    @Override
    public UserModel getUserForLoyaltyCardID(final String loyaltyCardId)
    {
        return getLoyaltyDao().findUserByLoyaltyCardId(loyaltyCardId);
    }
    protected LoyaltyDao getLoyaltyDao()
    {
        return LoyaltyDao;
    }
    @Required
    public void setLoyaltyDao(final LoyaltyDao LoyaltyDao)
```

```
        {
            this.LoyaltyDao = LoyaltyDao;
        }
    }
```

8. Configure the default implementations of your service and DAO in Spring by editing the `hybris/bin/modules/commerce-services/commerceservices/resources/commerceservices-spring.xml`file.

```
<alias name="defaultLoyalityDao" alias="loyalityDao"/>
<bean id="defaultLoyalityDao" class=" de.hybris.platform.commerceservices.customer.dao.impl.DefaultLoyalityDao"/>

<alias name="defaultLoyalityService" alias="loyalityService"/>
<bean id="defaultLoyalityService" class="de.hybris.platform.commerceservices.customer.impl.DefaultLoyalityService">
      <property name="loyalityDao" ref="loyalityDao"/>
</bean>
```

9. Navigate to the `hybris/bin/ext-template/ycommercewebservices/web/src/de/hybris/platform/ycommercewebservices/v2/controller/` directory and update the `CustomerGroupsController` class with the code below:

   - Properly annotate and declare a `loyaltyService` variable:

```
@Resource(name = "loyalityService")
private LoyalityService loyalityService;
```

   - Properly annotate and create the methods responsible for assigning and removing customers from groups:

```
@RequestMapping(value = "/{groupId}/geolocationRegister", method = RequestMethod.POST, consumes =
{ MediaType.APPLICATION_FORM_URLENCODED_VALUE })
@ResponseStatus(value = HttpStatus.OK)
public void assignUserToCustomerGroupByScanner(@PathVariable final String groupId, @RequestBody final MultiValueMap members
{
    String customerLoyaltyId = ((LinkedList) members.get("field_values")).get(0).toString();
    final Pattern pattern = Pattern.compile("/"([^/"]*)/"");
    final Matcher matcher = pattern.matcher(customerLoyaltyId);
    if (matcher.find())
    {
        customerLoyaltyId = matcher.group(1).toLowerCase();
    }
    final UserModel userModel = loyalityService.getUserForLoyaltyCardID(customerLoyaltyId);
    if (null != userModel)
    {
        checkMembers(Collections.singletonList(userModel.getUid()), "You cannot add user to group :" + sanitize(groupId) +
        customerGroupFacade.addUserToCustomerGroup(groupId, userModel.getUid());
    }
    else
    {
        throw new RequestParameterException("Can't find user with loyalty card[" + customerLoyaltyId + "]");
    }
}
/**
 * Removes user from a customer group.
 */
@RequestMapping(value = "/{groupId}/geolocationDeregister/{loyaltyCardId:.*}", method = RequestMethod.DELETE)
@ResponseStatus(value = HttpStatus.OK)
public void removeUsersFromCustomerGroupByScanner(@PathVariable final String groupId,
        @PathVariable(value = "loyaltyCardId") final String loyaltyCardId)
{
    final UserModel userModel = loyalityService.getUserForLoyaltyCardID(loyaltyCardId);
    checkMembers(Collections.singleton(userModel.getUid()), "You cannot remove user from group :" + sanitize(groupId) + "."
    customerGroupFacade.removeUserFromCustomerGroup(groupId, userModel.getUid());
}
```

ℹ Note

For more information about the Customer Lists Framework, see the **Customer Lists Framework** section of the commerceservices Extension document.

10. Navigate to the `hybris/bin/platform` directory, run the `ant clean all` command, and start the hybris Server.
11. Set up the device as per the manufacturer's instructions. You can use all sorts of devices to provide the data, such as cell phones and iBeacons.

    The device this procedure is based upon is Impinj Speedway Revolution. Note that every device sends the data in a specific way. Below, you can find the example REST calls the device makes to hybris:

Function	URL
Register a customer in store	**Note** The HTTP method is different depending on the device. Impinj Speedway Revolution uses POST for this call. `http://localhost:9001/rest/v2/electronics/customergroups/{groupId}/geolocationRegister`
Unregister a customer in store	The HTTP method is different depending on the device. Impinj Speedway Revolution uses DELETE for this call.`http://localhost:9001/rest/v2/electronics/customergroups/{groupId}/geolocationDeregister/{loyaltyGroupId}`

Commerce Quotes

Learn about the classes in `commerceservices` that are used for the Commerce Quotes feature.

The Commerce Quotes feature allows buyers to submit a quote request to their sales reps. The quote can be negotiated by both parties until an agreement is reached, at which point the buyer checks out the quote and places an order. The various quote-related items included in `commerceservices` are listed below.

CommerceComment, OrgUnit, and CommerceCartMetaData

These are the items related to CommerceComment, OrgUnit, and CommerceCartMetaData:

- `CommerceCommentService`: Offers a method to add a new comment to the item model, supplied with `CommerceCommentParameter`.
- `DefaultCommerceCommentService`: Default implementation of `CommerceCommentService` and extends `DefaultCommentService`.
- `CommerceCommentUtils`: Utility class that offers static methods to build the `CommerceCommentParameter` object using `CommerceCommentParameterBuilder`.
- `CommerceCommentParameterBuilder`: Builder class to build `CommerceCommentParameter`.
- `CommerceCommentParameter`: DTO/POJO used to pass along the data from the facade layer to the service layer.
- `OrgUnitQuoteService`: Offers an interface method to get quotes for an employee of an organization.
- `DefaultOrgUnitQuoteService`: Default implementation of `OrgUnitQuoteService`.
- `CommerceCartMetadataParameterUtils`: Instantiates `CommerceCartMetadataParameterBuilder`.
- `CommerceCartMetadataParameterBuilder`: Builder for `CommerceCartMetadataParameter`.
- `CommerceCartMetadataParameter`: DTO/POJO use to pass along the data from the facade layer to the service layer.

QuoteEvent Services

These are the services related to `QuoteEvent`:

- `AbstractQuoteSubmitEvent`: Superclass for quote-related events that offers common variables such as `QuoteModel`, `UserModel`, and `QuoteUserType`. Offers getter methods and default constructor to fetch to set these variables.
- `QuoteBuyerSubmitEvent`: Event indicating that the buyer submitted a quote.
- `QuoteCancelEvent`: Event to indicate the cancellation of a quote.
- `QuoteSalesRepSubmitEvent`: Event to indicate that sales rep has submitted a quote.
- `QuoteSellerApprovalSubmitEvent`: Event to indicate that the seller approver approved or rejected a quote submitted by the sales rep.
- `QuoteBuyerOrderPlacedEvent`: Event to indicate that the buyer placed an order from the vendor quote.

Quote Exceptions

These are the classes related to quote exceptions:

- `IllegalQuoteStateException`: An exception of type `RuntimeException` that is thrown when an action cannot be performed for a quote. Encapsulates basic quote information and offers several public constructors to construct the exception message.
- `IllegalQuoteSubmitException`: An exception of type `RuntimeException` that extends `IllegalQuoteStateException`. An exception is thrown when a submit action cannot be performed for a quote. Encapsulates basic quote information and offers several public constructors to construct the exception message.
- `CommerceQuoteAssignmentException`: An exception of type `RuntimeException` that extends `SystemException`. An exception is thrown if quote assignment fails.
- `CommerceQuoteExpirationTimeException`: An exception of type `RuntimeException` that extends `SystemException`. An exception is thrown if the quote has expired.

Quote Strategy Classes

These are the classes related to quote strategies:

- `QuoteUserTypeIdentificationStrategy`: Interface that provides a method to determine the `QuoteUserType` of the supplied `UserModel`.

- `DefaultQuoteUserTypeIdentificationStrategy`: Default implementation of `QuoteUserTypeIdentificationStrategy`.
- `QuoteStateSelectionStrategy`: Interface that offers methods for getting allowed states for an action, for getting allowed actions for a state, and for getting transition states for an action.
- `DefaultQuoteStateSelectionStrategy`: Uses `QuoteUserTypeIdentificationStrategy` for determining the `QuoteUserType` (BUYER,SELLER,SELLER_APPROVER). Makes use of pre-configured multilevel maps to determine actions or states for a particular `QuoteUserType`.
- `QuoteActionValidationStrategy`: Strategy that validates whether a user can perform an action on a quote. It offers two methods:
  - `void validate(QuoteAction quoteAction, QuoteModel quoteModel, UserModel userModel)`: Validated whether the supplied action can be performed on the quote; if invalid, it throws `IllegalQuoteStateException`.
  - `boolean isValidAction(QuoteAction quoteAction, QuoteModel quoteModel, UserModel userModel)`: Returns a boolean value to indicate whether the action is valid.
- `DefaultQuoteActionValidationStrategy`: Default implementation of `QuoteActionValidationStrategy`. Uses `QuoteStateSelectionStrategy` for getting allowed states for an action.
- `QuoteUpdateExpirationTimeStrategy`: Interface that provides a method to update expiration time of the supplied `QuoteModel`.
- `DefaultQuoteUpdateExpirationTimeStrategy`: Uses `QuoteUserTypeIdentificationStrategy` to determine the `QuoteUserType`. Based on the `QuoteUserType` and supplied action it updates the expiration time for the `QuoteModel`. For more implementation details, refer to the javadocs.
- `QuoteUpdateStateStrategy`: Interface that provides a method to update the state of a supplied `QuoteModel` for a particular action and user.
- `DefaultQuoteUpdateStateStrategy`: Default implementation of `QuoteUpdateStateStrategy`. Uses `QuoteStateSelectionStrategy` to find out the transition state for the supplied action and user. The transition state is used to update the state of the supplied `QuoteModel`.
- `QuoteUserIdentificationStrategy`: Interface that provides a method to determine the current session user.
- `DefaultQuoteUserIdentificationStrategy`: Default implementation for `QuoteUserIdentificationStrategy`.
- `QuoteMetadataValidationStrategy`: Interface that provides a method to validate quote metadata fields such as name, description, and expirationTime.
- `DefaultQuoteMetadataValidationStrategy`: Default implementation for `QuoteMetadataValidationStrategy`. If the name is empty, then it throws `IllegalArgumentException`. Uses `QuoteUserTypeIdentificationStrategy` to determine if the `QuoteUserType` is a SELLER_APPROVER and if the quote is being approved. If the expiration time is not set, then `IllegalStateException` is thrown.
- `QuoteAssignmentValidationStrategy`: Interface that provides methods to validate quote assignment and unassignment.
- `DefaultQuoteAssignmentValidationStrategy`: Default implementation of `QuoteAssignmentValidationStrategy`. The implementation methods throw `CommerceQuoteAssignmentException` for invalid assignment/unassignment operations.
- `QuoteCartValidationStrategy`: Interface that provides a method to validate the given two `AbstractOrderModels` based on certain criteria defined in the default implementation.
- `DefaultQuoteCartValidationStrategy`: Default implementation of `QuoteCartValidationStrategy`. This is primarily used during the quote checkout process on each checkout step to ensure the vendor quote is not modified until the order is placed.
- `QuoteExpirationTimeValidationStrategy`: Interface that provides a method to check whether the supplied quote has expired.
- `DefaultQuoteExpirationTimeValidationStrategy`: Default implementation of `QuoteExpirationTimeValidationStrategy`. This is primarily used when the buyer accepts the vendor quote to ensure the buyer cannot check out an expired quote offer. It also uses `TimeService` from the platform.
- `QuoteSellerApproverAutoApprovalStrategy`: Interface that provides a method to check whether the supplied quote requires manual approval from the SELLER_APPROVER.
- `DefaultQuoteSellerApproverAutoApprovalStrategy`: Default implementation of `QuoteSellerApproverAutoApprovalStrategy`. This is primarily used by the sales rep business process that is triggered when a sales rep submits a quote.

Cart and Order Classes

These are the classes related to cart and order strategies and hooks:

- `CommerceCreateCartFromQuoteStrategy`: Inherits `DefaultCreateCartFromQuoteStrategy` from platform to add commerce quote related business logic using the `postProcess` hook in `commerceservices`.
- `DefaultCommerceQuoteCartCalculationMethodHook`: Implements the `CommerceCartCalculationMethodHook` interface. It adds quote-related discounts to the global discounts and makes a call to calculate cart totals so that quote discounts are included in the global discounts.
- `CommercePlaceQuoteOrderMethodHook`: Implements the `CommercePlaceOrderMethodHook` interface. The primary purpose of this hook is to check if the order is a quote order, and then publish a `QuoteBuyerOrderPlacedEvent`.

CommerceCartMetadata

These are the classes related to `CommerceCartMetadata`:

- `CommerceCartMetadataUpdateStrategy`: Interface that provides a method to update cart metadata, such as name and description, using the `CommerceCartMetadataParameter` DTO.
- `DefaultCommerceCartMetadataUpdateStrategy`: Default implementation of `CommerceCartMetadataUpdateStrategy`. It updates the supplied cart with the supplied fields in `CommerceCartMetadataParameter`. It also provides `beforeUpdateMetadata` and `afterUpdateMetadata` hooks as extension points.
- `CommerceCartMetadataUpdateMethodHook`: Interface that provides `beforeUpdateMetadata` and `afterUpdateMetadata` hooks that can be used as extension points.
- `DefaultCommerceCartMetadataUpdateValidationHook`: Implements the `CommerceCartMetadataUpdateMethodHook` interface hook to validate the supplied name and description in `CommerceCartMetadataParameter`.
- `QuoteCommerceCartMetadataUpdateValidationHook`: Implements the `CommerceCartMetadataUpdateMethodHook` interface hook to validate the supplied `CommerceCartMetadataParameter` when a quote is saved. It uses `QuoteActionValidationStrategy`, `QuoteUserIdentificationStrategy`, `QuoteUserTypeIdentificationStrategy` and `TimeService`.

CommerceQuoteService

These are the classes related to `CommerceQuoteService`:

- `CommerceQuoteUtils`: Provides a helper method to remove existing quote discounts from `AbstractOrderModel`. It is used in `DefaultCommerceQuoteService` for applying quote discounts and for loading the quote as the session cart.
- `QuoteExpirationTimeUtils`: Offers several static utility helper methods that deal with `QuoteExpirationTime`.
- `CommerceQuoteDao`: Interface provides several quote retrieval methods based on different parameters such as customer, baseStore, and quote states.
- `DefaultCommerceQuoteDao`: Default implementation of `CommerceQuoteDao`, and is primarily used in `DefaultCommerceQuoteService`.
- `UpdateQuoteFromCartStrategy`: Interface that provides a method to update a quote with the contents of the supplied cart. The quote being updated is linked to the supplied cart.
- `DefaultUpdateQuoteFromCartStrategy`: Extends `GenericAbstractOrderCloningStrategy<QuoteModel, QuoteEntryModel, CartModel>`and is the default implementation of `UpdateQuoteFromCartStrategy`. It also provides the `postProcess` hook as an extension point.
- `OrderQuoteDiscountValuesAccessor`: Interface that provides methods to get and set the quote discount values of the supplied `AbstractOrderModel`.
- `DefaultOrderQuoteDiscountValuesAccessor`: Default implementation of `OrderQuoteDiscountValuesAccessor`. This accessor is primarily introduced to convert the list of discount values into string representation and vice-versa. The `getQuoteDiscountValues` is used in the `DefaultCommerceQuoteCartCalculationMethodHook.afterCalculate` method to apply quote discounts to `CartModel`. `getQuoteDiscountValues` is used in the `DefaultCommerceQuoteService.applyQuoteDiscount` method to apply quote discounts to `AbstractOrderModel`.
- `CommerceQuoteService`: Interface that provides methods for quote management operations such as retrieval, submitting, approving, rejecting, and editing.
- `DefaultCommerceQuoteService`: Default implementation of `CommerceQuoteService` that makes use of these services, DAOs, and utils classes:
  - `CommerceQuoteDao`
  - `CartService`
  - `ModelService`
  - `CommerceSaveCartService`
  - `SessionService`
  - `CommerceCartService`
  - `QuoteStateSelectionStrategy`
  - `QuoteActionValidationStrategy`
  - `QuoteUpdateStateStrategy`
  - `UpdateQuoteFromCartStrategy`
  - `quoteSnapshotStateTransitionMap`;
  - `QuoteService`
  - `CalculationService`
  - `QuoteUserTypeIdentificationStrategy`
  - `EventService`
  - `QuoteAssignmentValidationStrategy`
  - `QuoteSellerApproverAutoApprovalStrategy`
  - `QuoteCartValidationStrategy`
  - `QuoteExpirationTimeValidationStrategy`
  - `QuoteUpdateExpirationTimeStrategy`
  - `QuoteMetadataValidationStrategy`
  - `OrderQuoteDiscountValuesAccessor`
  - `UserService`
  - `CommerceQuoteUtils`
- `void assignQuoteToUser(QuoteModel quote, UserModel assignee, UserModel assigner)` and `void unassignQuote(QuoteModel quote, UserModel assigner)`: Methods used to assign and unassign users to a `QuoteModel`. For flexibility, the `UserModel` type is used; the assigner is passed to verify whether he is allowed to execute an operation.

Quote Validation Extension

The CommerceQuoteValidator interface allows you to provide custom validation logic, as well as a custom error message in case of a validation failure.

```
void validate(QuoteAction quoteAction, QuoteModel quoteModel, UserModel userModel);
```

The `defaultQuoteActionValidationStrategy` will call any registered validator during the quote validation, as well as when checking whether a certain quote transition is allowed. As soon as a validator indicates an issue by throwing an `IllegalQuoteStateException`, validation is interrupted and considered failed. No more validators will be invoked. Validation is considered a success only if all of the validators indicate no issues. To indicate why a validation failed, create an `IllegalQuoteStateException` with a custom message.

To implement your own custom validation logic, proceed as follows:

1. Create your custom validator class by implementing interface `de.hybris.platform.commerceservices.order.validator.CommerceQuoteValidator`.

2. Declare your custom validator as a spring bean.

```
<alias name="defaultCustomQuoteValidator" alias="customQuoteValidator" />
<bean id="defaultCustomQuoteValidator" class="com.custom.quote.validation.impl.MyCustomQuoteValidator"/>
```

3. Register it as a commerce quote validator by adding it to the list of validators via spring.

```
<bean id="customQuoteValidatorListMergeDirective" depends-on="commerceQuoteValidatorList" parent="listMergeDirective">
   <property name="add" ref="customQuoteValidator" />
</bean>
```

Related Information

Technical Overview
Quote Statuses
User Types
User Type Identification and State Selection Strategies
Commerce Quotes Items

Cart Entry Grouping Functionality

Entry Grouping functionality allows to group multiple cart entries into single package and operate with it as with one entity.

Platform provides types and basic APIs for handling entry groups, for more information, see the Grouping Cart Entries document. And `commerceservices` provide commerce functionality for adding, removing, or updating entry groups.

In order to provide entry group to the hooks, CommerceCartParameter has an attribute entryGroupNumbers. It is a set of unique numbers representing entry groups:

```
<bean class="de.hybris.platform.commerceservices.service.data.CommerceCartParameter"
                  template="resources/commerce-cart-parameter-template.vm">
        ...
        <property name="entryGroupNumbers" type="java.util.Set&lt;Integer>">
        <description>Entry group numbers</description>
    </property>
```

The `entryGroupNumbers` attribute can be found in commerce cart modification `de.hybris.platform.commerceservices.order.CommerceCartModification` and cart modification data `de.hybris.platform.commercefacades.order.data.CartModificationData`

`commercefacades/resources/commercefacades-beans.xml`

```
<bean class="de.hybris.platform.commercefacades.order.data.CartModificationData">
...
<property name="entryGroupNumbers" type="java.util.Set&lt;Integer>"/>
</bean>
```

`commerceservices/resources/commerceservices-beans.xml`

```
<bean class="de.hybris.platform.commerceservices.order.CommerceCartModification">
...
<property name="entryGroupNumbers" type="java.util.Set&lt;Integer>"/>
</bean>
```

Spring Configuration

A `de.hybris.platform.commerceservices.service.data.RemoveEntryGroupParameter` bean is added to `commerceservices/resources/commerceservices-beans.xml`.

```
<bean class="de.hybris.platform.commerceservices.service.data.RemoveEntryGroupParameter">
                <property name="cart" type="de.hybris.platform.core.model.order.CartModel">
                        <description>The CartModel</description>
                </property>
                <property name="enableHooks" type="boolean">
                        <description>Should the method hooks be executed</description>
                </property>
                <property name="entryGroupNumber" type="java.lang.Integer">
                        <description>Entry group number to be removed</description>
                </property>
        </bean>
```

`DefaultCartFacade` uses the standard workflow with service, strategy, and hook.

Two interfaces are created for strategy- `de.hybris.platform.commerceservices.order.CommerceRemoveEntryGroupStrategy` and hook - `de.hybris.platform.commerceservices.order.hook.CommerceRemoveEntryGroupMethodHook` to support removing entry group.

```
package de.hybris.platform.commerceservices.order;
import de.hybris.platform.commerceservices.service.data.RemoveEntryGroupParameter;
import javax.annotation.Nonnull;

/**
 * A strategy interface for adding products to cart cart.
 */
public interface CommerceRemoveEntryGroupStrategy
{
    /**
     * Removes from the (existing) {@link de.hybris.platform.core.model.order.CartModel} the (existing) {@link de.hybris.platform.core
     * If an entry with the given entry group already exists in the cart, it will be removed too.
     *
     * @param parameter - A parameter object containing all attributes needed for remove entry group
     *  <P>
     *      {@link RemoveEntryGroupParameter#cart} - The user's cart in session
     *      {@link RemoveEntryGroupParameter#entryGroupNumbers} - The number of the entry group to be removed from the cart
     *      {@link RemoveEntryGroupParameter#enableHooks} - Are hooks enabled
     *           </P>
     * @return the cart modification data that includes a statusCode and the actual entry group number removed from the cart
     * @throws CommerceCartModificationException
     *            if related cart entry wasn't removed
     */
    @Nonnull
    CommerceCartModification removeEntryGroup(@Nonnull final RemoveEntryGroupParameter parameter) throws CommerceCartModificationExcep
}
```

```
package de.hybris.platform.commerceservices.order.hook;

import de.hybris.platform.commerceservices.order.CommerceCartModification;
import de.hybris.platform.commerceservices.order.CommerceCartModificationException;
import de.hybris.platform.commerceservices.service.data.RemoveEntryGroupParameter;
import javax.annotation.Nonnull;
public interface CommerceRemoveEntryGroupMethodHook
{
    /**
     *  Executed after commerce remove entry group
     *
     * @param parameter
     * @param result
     */
    void afterRemoveEntryGroup(@Nonnull final RemoveEntryGroupParameter parameter, CommerceCartModification result);
    /**
     *
     * Executed before commerce remove entry gtoup
     *
     * @param parameter
     */
    void beforeRemoveEntryGroup(@Nonnull final RemoveEntryGroupParameter parameter) throws CommerceCartModificationException;
}
```

Default implementation of strategy

```
package de.hybris.platform.commerceservices.order.impl;
import de.hybris.platform.commerceservices.constants.CommerceServicesConstants;
import de.hybris.platform.commerceservices.order.CommerceCartModification;
import de.hybris.platform.commerceservices.order.CommerceCartModificationException;
import de.hybris.platform.commerceservices.order.CommerceCartModificationStatus;
import de.hybris.platform.commerceservices.order.CommerceEntryGroupService;
import de.hybris.platform.commerceservices.order.CommerceRemoveEntryGroupStrategy;
import de.hybris.platform.commerceservices.order.CommerceUpdateCartEntryStrategy;
import de.hybris.platform.commerceservices.order.hook.CommerceRemoveEntryGroupMethodHook;
import de.hybris.platform.commerceservices.service.data.CommerceCartParameter;
import de.hybris.platform.commerceservices.service.data.RemoveEntryGroupParameter;
import de.hybris.platform.core.model.order.AbstractOrderEntryModel;
import de.hybris.platform.core.model.order.CartModel;
import de.hybris.platform.core.order.EntryGroup;
import de.hybris.platform.servicelayer.config.ConfigurationService;
import de.hybris.platform.servicelayer.model.ModelService;
import org.springframework.beans.factory.annotation.Required;
import javax.annotation.Nonnull;
import java.util.List;
import java.util.stream.Collectors;
import static de.hybris.platform.servicelayer.util.ServicesUtil.validateParameterNotNull;
import static de.hybris.platform.servicelayer.util.ServicesUtil.validateParameterNotNullStandardMessage;
```

```
public class DefaultCommerceRemoveEntryGroupStrategy implements
        CommerceRemoveEntryGroupStrategy
{
    private List<CommerceRemoveEntryGroupMethodHook> commerceRemoveEntryGroupHooks;
    private ConfigurationService configurationService;
    private CommerceUpdateCartEntryStrategy updateCartEntryStrategy;
    private CommerceEntryGroupService entryGroupService;
    private ModelService modelService;
    /**
     * Removes from the cart an entry group with all nested groups and their cart entries
     *
     * @param parameter
     *           remove entry group parameters
     * @return Cart modification information
     * @throws CommerceCartModificationException if related cart entry wasn't removed
     *
     */
    @Override
    @Nonnull
    public CommerceCartModification removeEntryGroup(@Nonnull final RemoveEntryGroupParameter parameter) throws CommerceCartModificati
    {
        validateParameterNotNullStandardMessage("parameter", parameter);
        beforeRemoveEntryGroup(parameter);
        final CommerceCartModification modification = doRemoveEntryGroup(parameter);
        afterRemoveEntryGroup(parameter, modification);
        return modification;
    }
    /**
     * Do remove from the cart.
     *
     * @param parameter
     *           remove entry group parameter
     * @return the commerce cart modification
     * @throws CommerceCartModificationException
     *            the commerce cart modification exception
     */
    protected CommerceCartModification doRemoveEntryGroup(final RemoveEntryGroupParameter parameter) throws CommerceCartModificationEx
    {
        validateRemoveEntryGroupParameter(parameter);
        final CartModel cartModel = parameter.getCart();
        final Integer groupNumber = parameter.getentryGroupNumbers();
        try
        {
            final EntryGroup parentGroup = entryGroupService.getParent(cartModel, groupNumber);
            final EntryGroup group = getEntryGroupService().getGroup(cartModel, groupNumber);
            final List<Integer> groupNumbers = getAllSubsequentGroupNumbers(group);
            removeEntriesByGroupNumber(parameter, groupNumbers);
            if (parentGroup == null)
            {
                final List<EntryGroup> rootGroups = cartModel.getEntryGroups();
                cartModel.setEntryGroups(excludeEntryGroup(rootGroups, groupNumber));
            }
            else
            {
                parentGroup.setChildren(excludeEntryGroup(parentGroup.getChildren(), groupNumber));
            }
            getEntryGroupService().forceOrderSaving(cartModel);
            return createRemoveEntryGroupResp(parameter, CommerceCartModificationStatus.SUCCESSFULLY_REMOVED);

        }
        catch (IllegalArgumentException e)
        {
            return createRemoveEntryGroupResp(parameter, CommerceCartModificationStatus.INVALID_ENTRY_GROUP_NUMBER);
        }
    }
    protected List<EntryGroup> excludeEntryGroup(final List<EntryGroup> source, final Integer groupNumber)
    {
        return source.stream().filter(g -> !groupNumber.equals(g.getGroupNumber())).collect(Collectors.toList());
    }
    protected CommerceCartModification createRemoveEntryGroupResp(final RemoveEntryGroupParameter parameter, final String status)
    {
        final Integer entryGroupNumbers = parameter.getentryGroupNumbers();
        final CommerceCartModification modification = new CommerceCartModification();
        modification.setStatusCode(status);
        modification.setentryGroupNumbers(entryGroupNumbers);
        return modification;
    }
    protected void removeEntriesByGroupNumber(final RemoveEntryGroupParameter parameter, final List<Integer> groupNumbers)
            throws CommerceCartModificationException
    {
        final CartModel cartModel = parameter.getCart();
        if (cartModel.getEntries() == null)
        {
            return;
        }

        for (final AbstractOrderEntryModel entry : cartModel.getEntries())
        {
            if (groupNumbers.contains(entry.getentryGroupNumbers()))
            {
                final CommerceCartParameter updateParameter = new CommerceCartParameter();
                updateParameter.setCart(parameter.getCart());
                updateParameter.setEnableHooks(parameter.isEnableHooks());
                updateParameter.setQuantity(0L);
                updateParameter.setEntryNumber(entry.getEntryNumber());
                getUpdateCartEntryStrategy().updateQuantityForCartEntry(updateParameter);
            }
        }
    }
    protected List<Integer> getAllSubsequentGroupNumbers(final EntryGroup group)
```

```
    {
        return getEntryGroupService().getNestedGroups(group).stream()
                                     .map(EntryGroup::getGroupNumber)
                                     .collect(Collectors.toList());
    }
    protected void validateRemoveEntryGroupParameter(final RemoveEntryGroupParameter parameters)
    {
        final CartModel cartModel = parameters.getCart();
        final Integer entryGroupNumbers = parameters.getentryGroupNumbers();
        validateParameterNotNull(cartModel, "Cart model cannot be null");
        validateParameterNotNullStandardMessage("entryGroupNumbers", entryGroupNumbers);
    }
    protected void beforeRemoveEntryGroup(final RemoveEntryGroupParameter parameters) throws CommerceCartModificationException
    {
        if (getCommerceRemoveEntryGroupHooks() != null
                    && (parameters.isEnableHooks() && getConfigurationService().getConfiguration().getBoolean(CommerceServicesConstant
        {
            for (final CommerceRemoveEntryGroupMethodHook commerceAddToCartMethodHook : getCommerceRemoveEntryGroupHooks())
            {
                commerceAddToCartMethodHook.beforeRemoveEntryGroup(parameters);
            }
        }
    }
    protected void afterRemoveEntryGroup(final RemoveEntryGroupParameter parameters, final CommerceCartModification result)
            throws CommerceCartModificationException
    {
        if (getCommerceRemoveEntryGroupHooks() != null
                    && (parameters.isEnableHooks() && getConfigurationService().getConfiguration().getBoolean(CommerceServicesConstant
        {
            for (final CommerceRemoveEntryGroupMethodHook commerceAddToCartMethodHook : getCommerceRemoveEntryGroupHooks())
            {
                commerceAddToCartMethodHook.afterRemoveEntryGroup(parameters, result);
            }
        }
    }
    protected List<CommerceRemoveEntryGroupMethodHook> getCommerceRemoveEntryGroupHooks()
    {
        return commerceRemoveEntryGroupHooks;
    }
    @Required
    public void setCommerceRemoveEntryGroupHooks(final List<CommerceRemoveEntryGroupMethodHook> commerceRemoveEntryGroupHooks)
    {
        this.commerceRemoveEntryGroupHooks = commerceRemoveEntryGroupHooks;
    }
    protected ConfigurationService getConfigurationService()
    {
        return configurationService;
    }
    @Required
    public void setConfigurationService(final ConfigurationService configurationService)
    {
        this.configurationService = configurationService;
    }
    protected CommerceUpdateCartEntryStrategy getUpdateCartEntryStrategy()
    {
        return updateCartEntryStrategy;
    }
    @Required
    public void setUpdateCartEntryStrategy(final CommerceUpdateCartEntryStrategy updateCartEntryStrategy)
    {
        this.updateCartEntryStrategy = updateCartEntryStrategy;
    }
    protected CommerceEntryGroupService getEntryGroupService()
    {
        return entryGroupService;
    }
    @Required
    public void setEntryGroupService(final CommerceEntryGroupService entryGroupService)
    {
        this.entryGroupService = entryGroupService;
    }
    protected ModelService getModelService()
    {
        return modelService;
    }
    @Required
    public void setModelService(final ModelService modelService)
    {
        this.modelService = modelService;
    }
}
```

There is no default hook implementation. But in `commerceservices/resources/commerceservices-spring.xml` there is the following spring bean:

```
<util:list id="commerceRemoveEntryGroupMethodHooks" value-type="de.hybris.platform.commerceservices.order.hook.CommerceRemoveEntryGrou
```

The following statuses in `commerceservices/src/de/hybris/platform/commerceservices/order/CommerceCartModificationStatus.java` are available:

- `INVALID_ENTRY_GROUP_NUMBER`- if there is no entry group with such number in the cart.
- `SUCCESSFULLY_REMOVED` - if an entry group was successfully removed.

The `removeEntryGroup` can be found in `commerceservices/src/de/hybris/platform/commerceservices/order/CommerceCartService.java`. By default, the method calls the `CommerceRemoveEntryGroupStrategy` for removing process.

Facade Layer

`EntryGroupData` DTO is used to show entry groups on the storefront. It is defined in `commercefacades/resources/commercefacades-beans.xml`:

```
<bean class="de.hybris.platform.commercefacades.order.EntryGroupData">
		<import type="com.fasterxml.jackson.annotation.JsonManagedReference"/>
		<import type="com.fasterxml.jackson.annotation.JsonBackReference"/>
		<import type="com.fasterxml.jackson.annotation.JsonIdentityReference"/>
		<import type="com.fasterxml.jackson.annotation.JsonIdentityInfo"/>
		<import type="com.fasterxml.jackson.annotation.ObjectIdGenerators"/>
		<property name="groupNumber" type="java.lang.Integer"/>
		<property name="priority" type="java.lang.Integer"/>
		<property name="label" type="java.lang.String"/>
		<property name="groupType" type="de.hybris.platform.core.enums.GroupType"/>
		<property name="children" type="java.util.List&lt;de.hybris.platform.commercefacades.order.EntryGroupData&gt;">
			<annotations>@JsonManagedReference</annotations>
		</property>
		<property name="externalReferenceId" type="java.lang.String"/>
		<property name="erroneous" type="java.lang.Boolean"/>
		<property name="orderEntries" type="java.util.List&lt;de.hybris.platform.commercefacades.order.data.OrderEntryData&gt;
		<property name="rootGroup" type="de.hybris.platform.commercefacades.order.EntryGroupData">
			<annotations>@JsonIdentityInfo(generator=ObjectIdGenerators.PropertyGenerator.class, property="groupNumber")@J
		</property>
		<property name="parent" type="de.hybris.platform.commercefacades.order.EntryGroupData">
			<annotations>@JsonBackReference</annotations>
		</property>
	</bean>
```

AbstractOrderData has an additional attribute rootGroups which gets populated from `AbstractOrderModel.entryGroups`:

```
<bean class="de.hybris.platform.commercefacades.order.data.AbstractOrderData">
		...
		<property name="rootGroups" type="java.util.List&lt;de.hybris.platform.commercefacades.order.EntryGroupData&gt;"/>
</bean>
```

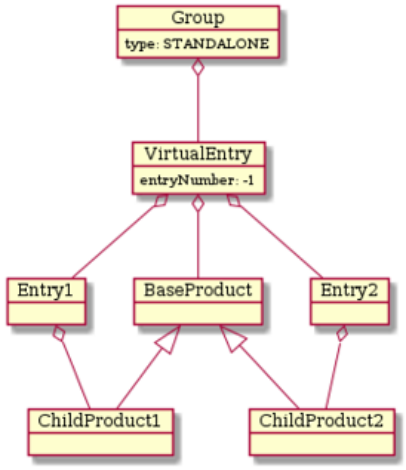
Jackson annotations are used to avoid Stackoverflow exceptions during the serialization because of two-directional parent-child references.

Standalone Entries

For each cart entry that doesn't belong to specific cart entry group, a virtual entry group is created, with `STANDALONE` group type . This behavior is implemented in `DefaultVirtualEntryGroupStrategy`. If there is a need for different handling of standalone entries, projects can implement their own `VirtualEntryGroupStrategy`.

Multidimensional Entries

With regular entries each entry is wrapped into a separate entry group during population from model to DTO and can be changed by redefining `VirtualEntryGroupStrategy`. Multidimensional entries (described by `GroupOrderEntryPopulator`) have entry group for parent entry only. The parent entry includes the subsequent entries as children.



Related Information

commerceservices Extension
Grouping Cart Entries
Cart Entry Grouping

Voucher Redemption, Validation, and Brute Force Attack Detection

Vouchers are redeemed upon placing an order. Previously this functionality was left up to the partner to implement.

Proper voucher redemption allows the system to register when vouchers are used, and to then enforce usage restrictions. Such restrictions include the total limit of available voucher redemptions, and the limit of redemptions per customer.

Multiple validations ensure that an order is not placed with an invalid voucher. These validations include the following:

- Upon adding a voucher to the cart, the voucher is validated.
- Upon starting the checkout process, the voucher is validated.
- Before placing the order at the end of the checkout process, the voucher is validated.

If a voucher is invalidated at the same time as an order is placed, an error occurs in the place order process, and the voucher is not redeemed.

Adding a voucher to the cart also contains a simple brute force attack detection that aims at preventing a brute force attack to guess voucher codes.

Customizing the Voucher Redemption Logic

The logic that redeems the vouchers is implemented using a hook. The default hook can be replaced by a custom hook. The following steps describe how to use a custom voucher redemption hook instead of the provided, default implementation.

1. Create a hook class that implements the interface `de.hybris.platform.commerceservices.order.hook.CommercePlaceOrderMethodHook`.
2. Register your class as a spring bean. The following is an example:

   ```
   <bean id="myCustomVoucherPlaceOrderMethodHook" class="com.example.MyCustomVoucherPlaceOrderMethodHook">
   ```

3. Redefine the alias `voucherRedemptionPlaceOrderMethodHook` so that it refers to the custom hook you created. The following is an example:

   ```
   <alias name="myCustomVoucherPlaceOrderMethodHook" alias="voucherRedemptionPlaceOrderMethodHook" />
   ```

   Note
   The redemption logic from the default implementation is in the `beforeSubmitOrder` method.

Customizing the Voucher Cart Validation Logic

The cart voucher validation logic that is triggered at the beginning and the end of the checkout process is implemented with a cart validation hook. The default hook can be replaced by a custom hook. The following steps describe how to use a custom voucher cart validation hook instead of the provided, default implementation.

1. Create a hook class that implements the interface `de.hybris.platform.commerceservices.strategies.hooks.CartValidationHook`.
2. Register your class as a spring bean. The following is an example:

   ```
   <bean id="myCustomVoucherCartValidationMethodHook" class="com.example.MyCustomVoucherCartValidationMethodHook" />
   ```

3. Redefine the alias `voucherRedemptionPlaceOrderMethodHook` so that it refers to the custom hook you created. The following is an example:

   ```
   <alias name="myCustomVoucherCartValidationMethodHook" alias="voucherRedeemableCartValidationMethodHook" />
   ```

   Note
   The validation logic from the default implementation is in the `afterValidateCart` method.

Configuring the Voucher Brute Force Attack Detection

A simple brute force detection mechanism is triggered when an attempt is made to add a voucher to the cart. The aim of the detection mechanism is to prevent brute force attacks trying to guess voucher codes.

By default, the brute force attack handler tracks attempts to add a voucher code on a per IP-address base. If a configurable amount of attempts in a configurable window of time have been made, any further attempts of that IP address will be blocked for a configurable amount of time. Additionally, the user is presented with a specific error message.

Another default setup is that any further attempts of an IP-address will be denied for one hour if more than 5 attempts have been made within 5 minutes.

The following parameters in the `local.properties` file configure the aforementioned behavior:

- `commerceservices.bruteForceAttackHandler.maxAttempts`: Defines how many attempts to enter a voucher code can be made before any further attempts will be denied.
- `commerceservices.bruteForceAttackHandler.timeFrame`: Defines the time window (in seconds) over which the the limit of attempts can not be exceeded. By default, this value is set to 300, i.e. 5 minutes.
- `commerceservices.bruteForceAttackHandler.waitTime`: Defines how long (in seconds) an IP-address is prevented from entering voucher codes after it is blocked. This value is set to 3600 by default. The 3600 seconds equate to one hour.

Note

You can set both the time frame and wait time property to 0, and restart your server to effectively disable the brute force attack detection.

Customizing Commerce Services

Guidelines on how to customize Commerce Services.

Overview

Commerce Services are configured to expose a part of the Commerce Facades. It is delivered as a `ycommercewebservices` extension, which is a template extension allowing you to create your own extension in your domain name space. You may customize it by adding new resources or by adapting serialization and deserialization. To achieve that, you need to make changes mainly to the Spring MVC Controllers that expose Commerce Services and to the DTOs provided by the Commerce Facades.

Creating Customized Extension Using extgen

Run the `ant extgen` command to create `ycommercewebservices`-based extension in your project's name space.

```
C:\workspace\bin\platform>ant extgen

extgen:
    [input]
    [input] Please choose the name of your extension. It has to start with a letter followed by letters and/or numbers.
    [input] Press [Enter] to use the default value [training]
```

Assume that your new extension name is `mysampleextension`.

Note

Do not use the standard template name within the new extension name that you create. This will lead to a Build error. For example, naming your new extension `mycommercewebservices` when extending it from `ycommercewebservices`, will show an error because the extension name contains the standard template name.

```
mysampleextension
    [input]
    [input] Please choose the package name of your extension. It has to fulfill java package name convention.
    [input] Press [Enter] to use the default value [org.training]
com.mycompany.mycommercewebservices
    [input]
    [input] Please choose a template for generation.
    [input] Press [Enter] to use the default value ([yempty], ycockpit, ybackoffice, yaddon, yacceleratorfulfilmentprocess, ycommercew
ycommercewebservices
     [echo] Using extension template source: C:\workspace\bin\modules\commerce-services\ycommercewebservices
   [delete] Deleting directory C:\workspace\temp\hybris\extgen_final
   [delete] Deleting directory C:\workspace\temp\hybris\extgen
    [mkdir] Created dir: C:\workspace\temp\hybris\extgen
     [echo] Copying template files from C:\workspace\bin\modules\commerce-services\ycommercewebservices to C:\workspace\temp\hybris\ex
     [copy] Copying 298 files to C:\workspace\temp\hybris\extgen
     [copy] Copying 298 files to C:\workspace\temp\hybris\extgen_final
    [mkdir] Created dir: C:\workspace\bin\custom\mysampleextension\lib
     [copy] Copying 215 files to C:\workspace\bin\custom\mysampleextension
     [echo]
     [echo]
     [echo]     Next steps:
     [echo]
     [echo] 1) Add your extension to your C:\workspace\config/localextensions.xml
     [echo]
     [echo]      <extension dir="C:\workspace\bin/custom/mysampleextension"/>
     [echo]
     [echo] 2) Make sure the applicationserver is stopped before you build the extension the first time.
     [echo]
     [echo] 3) Perform 'ant' in your hybris/platform directory.
     [echo]
     [echo] 4) Restart the applicationserver
     [echo]
     [echo]

BUILD SUCCESSFUL
```

Follow the **Next steps** hints to include your new extension in your hybris platform installation. Replace the `ycommercewebservices` with `mysampleextension` in the `localextensions.xml` file.

```
<!-- <extension dir="${HYBRIS_BIN_DIR}/ext-hybris/ycommercewebservices"/> -->
              <extension dir="${HYBRIS_BIN_DIR}/custom/mysampleextension"/>
```

Edit the `extensioninfo.xml` file for your customized Commerce Services extension to provide a `webroot` context that best fits your project, for example:

```
<extensioninfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="extensioninfo.xsd">

   <extension  name="mysampleextension"
               abstractclassprefix="Generated"
               classprefix="mysampleextension"
               >
```

```
                <requires-extension name="commercefacades"/>
                <requires-extension name="commerceservices"/>
                <webmodule jspcompile="false" webroot="/mysampleextension"/>

        </extension>

</extensioninfo>
```

Provide the base root path information for your web services extension in the `project.properties` file:

```
...
commercewebservices.rootcontext=/mysampleextension/v1/
...
```

More detailed information on creating an extension, see Creating a New Extension.

Adding New Resources

You can add new resources for the customized Commerce Services by defining new handler methods within the web services controllers. In order to make effective use of Commerce Services, you should become familiar with the following Spring MVC annotations:

- `@Controller`
- `@RequestMapping`, both used on a class and method level
- `@ResponseBody`
- `@RequestParam`
- `@PathVariable`

Let us create a controller that performs mapping to the `/v1/deals/**` path. The contained handler method should specifically map onto HTTP GET requests to this controller. Note that the `@RequestMapping` annotation on the class level automatically requires all incoming requests to have the `Accept` header with the value of `application/xml` or `application/json`. These are two representation formats supported by Commerce Services out of the box.

DealsController.java

```
@Controller
@RequestMapping(value = "/v1/deals", headers = "Accept=application/xml,application/json")
public class DealsController extends BaseController
{
        @RequestMapping(method = RequestMethod.GET)
        @ResponseBody
        public DealList doSomething(final HttpServletRequest request)
        {
                ...
        }
}
```

XML/JSON Serialization of Returned Objects

Note

Information included below pertains to the v1 of Commerce Services. For information dedicated to v2 see: HTTP Message Converters and HTTP Message Converters.

The serialization of returned object to XML or JSON does not require the use of annotations in the target classes. Commerce Services incorporate the Xstream library to perform serialization of returned objects. The customization of the serialization can be done in Spring.

Custom XML/JSON Serialization

We created a basic spring base configuration that provides custom spring mappings to customize the serialized output of the Commerce Services web frontend API.

This configuration consists of the:

- `AttributeOmitMapping`
- `FieldAliasMapping`
- `TypeAliasMapping`

TypeAliasMapping

`TypeAliasMapping` is used to provide alias to the serialized element, for example:

```
<bean class="de.hybris.platform.commercefacades.xstream.alias.TypeAliasMapping">
                <property name="alias" value="product" />
                <property name="aliasedClass"  value="de.hybris.platform.commercefacades.product.data.ProductData" />
        </bean>
```

The previous code renames all of the instances of serialized `ProductData` objects in the response with the appropriate `product` JSON object or `product` tag in the XML response.

FieldAliasMapping

Allows to re-alias attributes in the serialized objects, for example:

```
<bean class="de.hybris.platform.commercefacades.xstream.alias.FieldAliasMapping">
                <property name="alias" value="class" />
                <property name="fieldName" value="className" />
                <property name="aliasedClass" value="de.company.customwebservices.dto.ErrorData" />
        </bean>
```

The code would rename the attribute of `ErrorData` instances from original `className` to `class`.

AttributeOmitMapping

This mapping skips a field from a target object during serialization, for example:

```
<bean class="de.hybris.platform.commercefacades.xstream.alias.AttributeOmitMapping">
                <property name="attributeName" value="code" />
                <property name="aliasedClass" value="de.hybris.platform.commerceservices.search.facetdata.FacetData" />
        </bean>
```

The code removes the code attribute from both JSON and XML serialized `FacetData` objects.

Related Information

commercefacades Extension - Technical Guide

Integrating with Adobe Experience Manager

SAP Commerce can easily integrate with Adobe Experience Manager (AEM). This allows you to connect the SAP Commerce solution responsible for controlling product data, shopping carts, checkout and order fulfillment, while AEM controls the data display and marketing campaigns.

The implementation provided out-of-the-box can be extended to include the search functionality.

Installation Overview

You need to install both SAP Commerce and Adobe Experience Manager before you proceed with integration.

Installing SAP Commerce

ℹ Note

This section describes only the custom configuration and gives a basic view over the installation process.

For more details on installation process, see Installing and Upgrading SAP Commerce.

For more details on Express Update functionality, see Enabling the Product Express Update.

1. First extract the latest artifact into your destination directory. For the purpose of this example, name it *<DIR>*.
2. Next, set your working directory to the *<DIR>* `/hybris/bin/platform` and run the following commands.

   ```
   . ./setantenv.sh
   ant clean
   ```

3. At this point, all **config** files should be generated and present in the *<DIR>* `/hybris/config` directory.
4. The sample below shows the minimal extension set required for our integration. However, the real set will depend on the specific needs of customers using this integration in their environments.

   ```
   <hybrisconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="resources/schemas/extensions.
           <extensions>
                   <!--
                                   All extensions located in ${HYBRIS_BIN_DIR}/platform/ext are automatically loaded.
                                   More information about how to configure available extensions can be found here: https://help.sap
                   -->

                   <path dir="${HYBRIS_BIN_DIR}" />

                   <!-- ext-platform -->
                   <extension name="productcockpit" />
                   <extension name="yacceleratorstorefront" />
                   <extension name="yacceleratorinitialdata" />
                   <extension name="electronicsstore" />
                   <extension name="sampledata" />
   ```

```
                <extension name="ycommercewebservices" />
        </extensions>
</hybrisconfig>
```

5. Now, save this content to the `localextensions.xml` in the *<DIR>* `/hybris/config` directory.
6. By default, the platform uses HSQL for storing the data. In order to have better performance, especially on large catalogs, the MySQL database on the development server is recommended. To use mySQL, you have to provide MySQL server configuration in the `local.properties` file in the ***DIR*** `/hybris/config` directory.

```
db.url=jdbc:mysql://localhost/<database_name>?useConfigs=maxPerformance&characterEncoding=utf8
db.driver=com.mysql.jdbc.Driver

# DB PASSWORD SHOULD BE CHANGED TO MATCH YOUR MYSQL INSTALLATION
db.username=<username>
db.password=<password>
db.tableprefix=
mysql.optional.tabledefs=CHARSET=utf8 COLLATE=utf8_bin
mysql.tabletype=InnoDB
```

7. In order to enable **Express Update** functionality you have to declare for which catalog version it should be available. In case of the example below, only products from the **Online** catalog version can be added to the Express Update queue.

```
expressUpdateCatalogVersion=Online
```

8. When the configuration is ready, you can build and initialize the platform.

```
ant all initialize
```

Boosting Performance

ⓘ Note

If you use MySQL server in combination with InnoDB-based database tables you can experience huge performance impact. It is recommended to set the **innodb_flush_log_at_trx_commit** variable to **0**. Use MySQL console to set that option:

```
mysql> set global innodb_flush_log_at_trx_commit=0;
```

If your development server has more that one core, you can set extra options to speed up the initialization process, as well as importing and synchronizing catalogs later on.

```
# OPTIMIZATION
# WARNING! - number of impex.import and catalog.sync workers could have maximum value equal number of cores
impex.import.workers=8
catalog.sync.workers=8
build.parallel=true
```

You can also customize cache settings with help of the following options.

```
cache.main=120000
regioncache.entityregion.size=120000
```

Adobe AEM

1. Download the latest build of CQ5, for example **cq-quickstart-5.6.1.jar**
2. Rename the file to **cq5-author-p4502.jar** and run it with custom parameters:

```
java -XX:MaxPermSize=280M -Xmx1100M -jar cq5-author-p4502.jar
```

3. During the first run all packages should be installed, the process can take a few minutes. When the server is ready, you can access it in your browser: `http://localhost:4502`. You can use default credentials:
    - login: **admin**
    - password: use the password you defined for your account.
4. Use Adobe's Package Share to install the latest SAP Commerce-package builds. The packages you want to install are:
    - `cq-hybris-server`
    - `cq-hybris-content`
    - `cq-geometrixx-hybris-content`

Embedded SAP Commerce

If you have `cq-hybris-server` package installed, Platform will start automatically along with AEM. You do not have to do anything more as it is already pre-configured and works out of the box. Keep in mind that embedded solution includes HSQL database which has performance impact.

Standalone SAP Commerce

If you want to integrate AEM with the standalone SAP Commerce installation you have to configure URL to the right instance:

1. Go to `http://localhost:4502/system/console/configMgr`.
2. Search for **Day CQ Commerce Hybris Connection Handler** and click on it.

3. Now, change **Base URL** variable in the popup.
4. You can also change other values if required.
5. Test integration by importing the catalog from the Platform: `http://localhost:4502/etc/importers/hybris.html`.

Related Information

Enabling the Product Express Update
Removing Old Carts with Cronjob

Removing Old Carts with Cronjob

Carts are not deleted after the session is closed. This way it is still possible to restore the cart later when the user gets back online. However, to avoid storing large amount of old carts you can use the cronjob to remove them.

Cronjob

The `OldCartRemovalCronJob` item type is defined in the `ycommercewebservices-items.xml` file. There are three attributes defining which carts should be removed:

- `sites`: Collection of sites for which old carts will be removed
- `cartRemovalAge`: Carts older then a specified number of seconds will be removed
- `anonymousCartRemovalAge`: Anonymous carts older then a specified number of seconds will be removed

ycommercewebservices-items.xml

```
...
		<itemtype code="OldCartRemovalCronJob" autocreate="true"
			generate="true" extends="CronJob"
			jaloclass="de.hybris.platform.ycommercewebservices.jalo.OldCartRemovalCronJob">
		<attributes>
			<attribute type="BaseSiteCollection" qualifier="sites">
					<modifiers/>
					<persistence type="property"/>
					<description>BaseSites for which old carts should be removed</description>
			</attribute>
			<attribute type="java.lang.Integer" qualifier="cartRemovalAge">
				<modifiers/>
				<persistence type="property"/>
				<defaultvalue>Integer.valueOf(2419200)</defaultvalue>
				<description>After specified number of seconds carts will be cleaned up. Default is 28 days.</descript
			</attribute>
			<attribute type="java.lang.Integer" qualifier="anonymousCartRemovalAge">
					<modifiers/>
					<persistence type="property"/>
					<defaultvalue>Integer.valueOf(1209600)</defaultvalue>
					<description>After specified number of seconds carts will be cleaned up. Default is 14 days.</
			</attribute>
		</attributes>
	</itemtype>
...
```

Job Object

The `OldCartRemovalJob` class defines the logic responsible for removing old cards. The relevant bean is defined in the `ycommercewebservices-spring.xml` file. See the code sample for an example.

```
...
	<bean id="oldCartRemovalJob" class="de.hybris.platform.ycommercewebservices.cronjob.OldCartRemovalJob"  parent="abstractJobPer
		<property name="commerceCartDao" ref="commerceCartDao"/>
		<property name="timeService" ref="timeService"/>
		<property name="userService" ref="userService"/>
	</bean>
...
```

Trigger

The sample `OldCartRemovalCronJob` is created during the init/update process. It remains active and is triggered daily at 3:30 AM. The script located in the `ycommercewebservices/resource/` directory provides an example of adding a cronjob.

```
...
# Old Cart Cleanup CronJobs
# This is only sample cron job but to make it work correctly you need to set also sites property for it
INSERT_UPDATE OldCartRemovalCronJob;code[unique=true];job(code);cartRemovalAge;anonymousCartRemovalAge;sessionLanguage(isoCode)[defaul
;oldCartRemovalCronJob;oldCartRemovalJob;2419200;1209600

INSERT_UPDATE Trigger;cronJob(code)[unique=true];second;minute;hour;day;month;year;relative;active;maxAcceptableDelay
;oldCartRemovalCronJob;0;30;3;-1;-1;-1;false;true;-1
...
```

Note

The `ycommercewebservices` extension is not aware about the sites available in the system. As a result, the sample cronjob does not have the `sites` attribute set.

To make it work correctly you need to set the `sites` attribute. You can do it directly in the ImpEx script file or by using visual interface of the Backoffice.

Example for adding the `sites` attribute with help of the ImpEx script file.

```
#Setting Old Cart Cleanup CronJobs for wsTest site
INSERT_UPDATE OldCartRemovalCronJob;code[unique=true];job(code);sites(uid)
;oldCartRemovalCronJob;oldCartRemovalJob;wsTest
```

Adding Sites

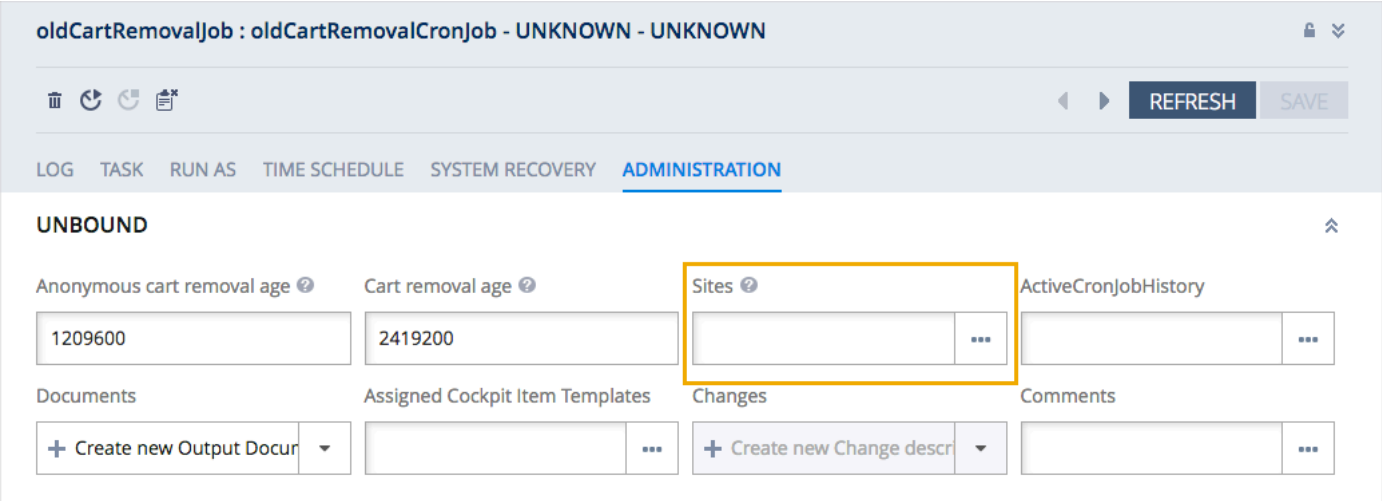
Context

Use the Backoffice Administration Cockpit to add the `sites` attribute.

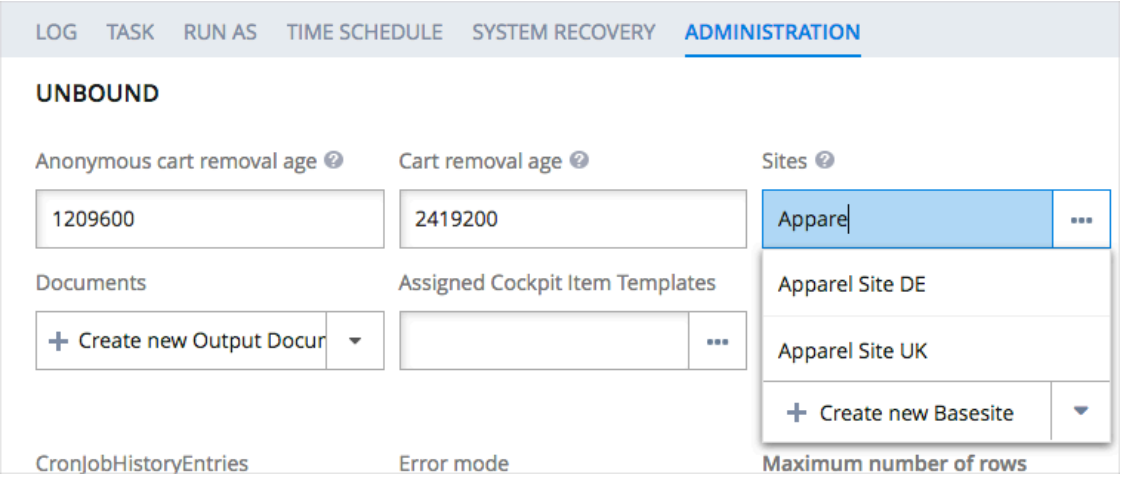
Procedure

1. Navigate to System > Background Processes > CronJobs
2. In the main editor, search for **oldCartRemovalCronJob**.
3. Click the entry.

   An editor opens.
4. Go to Administration > Sites



5. Start typing in the name of the site you would like to add or use the Reference Search window.



6. The list now includes the sites for which the old carts will be removed by the selected cronJob.
7. Save your configuration.

Extend Commerce Services

Commerce Services can be extended by creating an AddOn with one additional `Controller` class and using appropriate methods. However, in most cases some more complex changes are required to build an AddOn which adds the functionality to Commerce Services.

This document describes how to successfully extend Commerce Services.

Note

For details on how to create an AddOn for OCC Web Services refer to: Creating an AddOn for OCC Web Services.

Extending process is described based on the `occaddon`. It is a sample AddOn that extends Commerce Services. The `occaddon` adds new properties to the `Customer` item type:

- `nickname`
- `workOfficeAddress`

A REST call allows you to set these properties. In the `occaddon,` there is also an example of how to override an existing request mapping.

Extending Data Objects

You can extend the data model and data transfer object for Commerce Services using an AddOn.

Extending the Data Model

Procedure

1. In the `extensionname-items.xml` file, replace the `extensionname` part in the file with the name of the actual extension.

   The sample AddOn extends the `CustomerModel` class with two properties:

   - `nickname`
   - `workOfficeAddress`

   Note

   For more details, see items.xml.

2. Verify that you see the correct results in the `*-items.xml` file.

   The `*-items.xml` file looks like the following sample.

   `occaddon-items.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<items  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xsi:noNamespaceSchemaLocation="items.xsd">
        <itemtypes>
                <typegroup name="Customer">
                        <itemtype code="Customer" autocreate="false" generate="false">
                                <description>Extending Customer type with additional attributes.</description>
                                <attributes>
                                        <attribute autocreate="true" qualifier="nickname" type="java.lang.String">
                                                <modifiers read="true" write="true" optional="true" />
                                                <persistence type="property" />
                                                <description>Customer nickname</description>
                                        </attribute>
                            <attribute autocreate="true" qualifier="workOfficeAddress" type="Address" isSelectionOf="addresses">
                               <modifiers read="true" write="true" search="false" optional="true" />
                                   <persistence type="property" qualifier="WorkOfficeAddress"/>
                        </attribute>
                                </attributes>
                        </itemtype>
                </typegroup>
        </itemtypes>
</items>
```

Extending the Data Transfer Object

Procedure

1. In the `extensionname-beans.xml` file, replace the *<extensionname>* part with the name of the current extension.

   The sample AddOn extends the `CustomerData` with two properties:

   - `nickname`
   - `workOfficeAddress`

   Note

For details on generating custom Java Beans, see Generating Beans and Enums.

2. Verify that you see the correct results in the `*-beans.xml` file.

`occaddon-beans.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="beans.xsd">
        <!-- Additional nickname and workofficeAddress property for CustomerData -->
        <bean class="de.hybris.platform.commercefacades.user.data.CustomerData" extends="de.hybris.platform.commercefacades.user
                <property name="nickname" type="String" />
                <property name="workOfficeAddress" type="de.hybris.platform.commercefacades.user.data.AddressData"/>
        </bean>
</beans>
```

Populating Data

Context

The `Converter` objects in the *<commerce>-** extensions are configured in Spring only and have no separate concrete implementation classes. During the conversion process, they use a list of `Populator` objects. It is possible to add a `Populator` to an existing `Converter` without having to redeclare the `Converter`. Use the `modifyPopulatorList` method.

This procedure provides the steps to populate new attributes.

Procedure

1. Define the Populator Class in `OccaddonCustomerPopulator.java`.

```
public class OccaddonCustomerPopulator implements Populator<CustomerModel, CustomerData>
{
    private Converter<AddressModel, AddressData> addressConverter;
    @Override
    public void populate(final CustomerModel source, final CustomerData target) throws ConversionException
    {
        Assert.notNull(source, "Parameter source cannot be null.");
        Assert.notNull(target, "Parameter target cannot be null.");
        target.setNickname(source.getNickname());
        if (source.getWorkOfficeAddress() != null)
        {
            target.setWorkOfficeAddress(getAddressConverter().convert(source.getWorkOfficeAddress()));
        }
    }
    protected Converter<AddressModel, AddressData> getAddressConverter()
    {
        return addressConverter;
    }
    @Required
    public void setAddressConverter(final Converter<AddressModel, AddressData> addressConverter)
    {
        this.addressConverter = addressConverter;
    }
    }
```

2. Add the Populator to the Converter in `occaddon-spring.xml`.

```
...
        <alias name="defaultOccaddonCustomerPopulator" alias="occaddonCustomerPopulator"/>
        <bean id="defaultOccaddonCustomerPopulator" class="de.hybris.platform.occaddon.customer.converters.populator.OccaddonCus
                <property name="addressConverter" ref="addressConverter"/>
        </bean>

        <bean parent="modifyPopulatorList">
                <property name="list" ref="customerConverter"/>
                <property name="add" ref="occaddonCustomerPopulator"/>
        </bean>
...
```

Next Steps

The same actions must be performed to the reverse `Converter`.

ℹ Note

For more information on Converters and Populators, refer to Converters and Populators.

Localizing Attributes

Procedure

Add `<key>=<localized string>` entries in the following type system localization file: `occaddon/resources/localization/occaddon-locales_en.properties`.

`occaddon-locales_en.properties`

```
type.Customer.nickname.name=Nickname
type.Customer.nickname.description=Nickname description
type.Customer.workOfficeAddress.name=Work office address
type.Customer.workOfficeAddress.description=Work office address
```

Extending the Data Transfer Object (DTO) for v2

Context

For **v2** of Commerce Services, there is an additional DTO layer. It was created to improve the stability and configurability of the response data.

To use the new DTO and configure responses, perform the following steps:

ℹ Note

For more information, refer to WsDTO Concept.

Procedure

1. Extend the DTO in `occaddon-beans.xml`.

```
...
    <!-- Additional nickname and workofficeAddress property for UserWsDTO -->
    <bean class="de.hybris.platform.commercewebservicescommons.dto.user.UserWsDTO"
          extends="de.hybris.platform.commercewebservicescommons.dto.user.PrincipalWsDTO">
        <property name="nickname" type="String" />
        <property name="workOfficeAddress" type="de.hybris.platform.commercewebservicescommons.dto.user.AddressWsDTO"/>
    </bean>
    ...
```

2. Add new fields to the predefined field level configurations in `occaddon-web-spring.xml`.

   Remember that you need to do this in the *<addonname>*`-web-spring.xml` file localized in the resource directory, which is added to the Commerce Services context.

```
...
        <bean parent="fieldSetLevelMapping">
        <property name="dtoClass"
            value="de.hybris.platform.commercewebservicescommons.dto.user.UserWsDTO" />
        <property name="levelMapping">
            <map>
                <entry key="BASIC" value="nickname" />
                <entry key="DEFAULT" value="nickname" />
                <entry key="FULL" value="nickname,workOfficeAddress(FULL)" />
            </map>
        </property>
    </bean>
    ...
```

   Additional Resources

   - For more information on the web spring context refer to Extend Commerce Services.

3. Populate data between commerce data and web services DTO. Populating data from the commerce layer to web services DTO is done with the help of `Orika` - a popular Java Bean mapper framework. Fields with the same names are populated automatically by methods from version v2 with the help of the DataMapper object.

   UsersController.java

```
...
    public UserWsDTO getUser(@RequestParam(defaultValue = "BASIC") final String fields)
    {
        final CustomerData customerData = customerFacade.getCurrentCustomer();
        final UserWsDTO dto = dataMapper.map(customerData, UserWsDTO.class, fields);
        return dto;
    }
    ...
```

   You can also use the DataMapper in the AddOn controllers. The default implementation of DataMapper is defined in the `commercewebservicescommons` extension. To use it in an AddOn, define the dependency for this extension and add the proper resource in the controller:

   ...Controller.java

```
...
    @Resource(name = "dataMapper")
    protected DataMapper dataMapper;
    ...
```

Extending the REST API

You can extend the REST API for Commerce Services using an AddOn.

Defining a Controller

Context

To expose new calls, define a `Controller` class with the appropriate methods.

Procedure

Create the `Controller` in the `/acceleratoraddon/web/src/de/hybris/platform/acceleratorwebservicesaddon/controllers` directory.

`ExtendedCustomersController.java`

```
/**
 * Controller which extends Customer Resources
 */
@Controller("sampleExtendedCustomerController")
@RequestMapping(value = "/{baseSiteId}/customers")
public class ExtendedCustomersController
{
...
	@Secured("ROLE_CUSTOMERGROUP")
	@RequestMapping(value = "/current/nickname", method = RequestMethod.GET)
	@ResponseBody
	public String getCustomerNickname()
	{
		final String name = getCustomerFacade().getCurrentCustomer().getNickname();
		return name;
	}

	@Secured("ROLE_CUSTOMERGROUP")
	@RequestMapping(value = "/current/nickname", method = RequestMethod.PUT)
	@ResponseBody
	public CustomerData setCustomerNickname(@RequestParam final String nickname) throws DuplicateUidException
	{
		final CustomerData customer = customerFacade.getCurrentCustomer();
		customer.setNickname(nickname);
		customerFacade.updateFullProfile(customer);
		return customerFacade.getCurrentCustomer();
	}
...
}
```

i Note

When you create a `controller` in the AddOn, you should not use the class defined in `ycommercewebservices`. The package name for such a class changes once the `extgen` process is completed.

Creating the Web Spring Context

Context

Since the sample controller is annotated as `@Controller`, Spring must be told where it should look for it.

Procedure

In the resource directory, update the *<addonname>*`-web-spring.xml` file.

The file extends Commerce Services context. This file **is not** the standard web context file from the `web\webroot\WEB-INF` directory.

i Note

If your AddOn was generated from the `yoccaddon` template, component scan configuration for `<your_addon_package>.controllers` is set.

resource/occaddon/web/spring/occaddon-web-spring.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
           http://www.springframework.org/schema/context
                   http://www.springframework.org/schema/context/spring-context-3.1.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">

        <context:component-scan base-package="de.hybris.platform.occaddon.controllers"/>
</beans>
```

Adding the Web Context to Commerce Services

Context

Extend Commerce Services context with the context from the newly created AddOn. You can accomplish this by using the `additionalWebSpringConfigs` mechanism. The `project.properties` file must contain the following line where `Spring Config Classpath` points to any classpath resource from the `<addonExtension>`:

```
<targetExtension>.additionalWebSpringConfigs.<addonExtension>=[Spring Config Classpath]
```

Procedure

Add the following line to the `project.properties.template` file.

`project.properties.template`

```
ycommercewebservices.additionalWebSpringConfigs.<addOnName>=classpath:/occaddon/web/spring/<addOnName>-web-spring.xml
```

This file is a template for the `project.properties` file, which is generated during the installation process.

ℹ Note

If your AddOn was generated from `yoccaddon` template, `project.properties.template` should already have the proper content.

Verify the New Endpoints Defined in the Controller

ℹ Note

If you use the sample code in the Extending Data Objects, you need to rebuild and initialize/update the Commerce System to apply the new defined attributes in the `Customer` type and new beans. For the details, see Installing and Upgrading SAP Commerce.

Request Flow:

Method	URL	Header	Body Parameter
POST	https://localhost9002/authorizationserver/oauth/token	Content-Type: application/x-www-form-urlencoded	`client_id=$CLIENT_ID$&client_secret=$CLIENT_SECRET$&grant_type=passwor`
PUT	https:/localhost9002/occ/v2/electronics/users/8716fdb8-6cea-4e28-8396-1258e819f758/nickname • 8716fdb8-6cea-4e28-8396-1258e819f758 is userId.	Content-Type: application/x-www-form-urlencoded	nickname=myNickName
GET	https://localhost9002/occ/v2/electronics/users/8716fdb8-6cea-4e28-8396-1258e819f758/nickname • 8716fdb8-6cea-4e28-8396-1258e819f758 is userId.		

Overriding the REST API

Context

In an **AddOn** for Commerce Services, you can override the existing calls. The only difference between extending and overriding the REST API is using the `@RequestMappingOverride` annotation.

Procedure

Use the `@RequestMappingOverride` annotation to set priorities for methods with identical `@RequestMapping`.

`RequestMappingOverride`

```
@Target(
{ ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface RequestMappingOverride
{
    /**
     * Name for property, which stores priority value
     */
    String priorityProperty() default "";
}
```

The method with the **highest** priority is used.

⚠ Caution

`@RequestMappingOverride` should be used to override an identical @RequestMapping annotation.

It will however **not** work correctly in a situation where the original request mapping supports **two HTTP methods** and you will try to override only one of them:

- request mapping for original method = @RequestMapping(value = "/current/addresses/default/{id}", method = { RequestMethod.PUT, RequestMethod.POST })

- request mapping for method, which should override original = @RequestMapping(value = "/current/addresses/default/{id}", method = RequestMethod.PUT)

Even example below is considered as different mapping :

- request mapping for original method = @RequestMapping(value = "/{productCode}", method = RequestMethod.GET)

- request mapping for method, which should override original = @RequestMapping(value = "/{productId}", method = RequestMethod.GET)

In such cases methods will not be overridden and error "Ambiguous handler methods..." will appear during request (not during platform start up)

The `Priority value` is read from the properties file (`project.properties`, `local.properties` files) based on:

- `priorityProperty` given in the annotation or
- property name **requestMappingOverride.<className>.<methodName>.priority**

Example:

`requestMappingOverride.de.hybris.platform.occaddon.controllers.ExtendedCustomersController.updateDefaultAddress.priority`

If there is no property defined in the properties file, the priority value is set to zero.

Example:

- `@RequestMappingOverride(priorityProperty="occaddon.updateDefaultAddress.priority")` - here the priority value is read from the **occaddon.updateDefaultAddress.priority** property.
- `@RequestMappingOverride` - here the priority value is read from `requestMappingOverride.<className>.<methodName>.priority` property (e.g. `requestMappingOverride.de.hybris.platform.occaddon.controllers.ExtendedCustomersController.updateDefaultAddress.priority`).

ⓘ Note

The `priority value` is read from the properties file to resolve a situation when more than one method overrides the original call. In this case, you can select the preferred method by setting the highest priority value for it in the `local.properties` file.

Overriding the Request Mapping Example

`ExtendedCustomerController.java`

```
/**
 * Controller which extends Customer Resources
 */
@Controller("sampleExtendedCustomerController")
@RequestMapping(value = "/{baseSiteId}/customers")
public class ExtendedCustomersController
{
...
    /**
     * This is example of overriding existing request mapping. Annotation {@link RequestMappingOverride} allows override
     * existing request mapping, defined by {@link RequestMapping} annotation.
     */
    @Secured("ROLE_CUSTOMERGROUP")
    @RequestMapping(value = "/current/addresses/default/{id}", method = RequestMethod.PUT)
    @RequestMappingOverride
    @ResponseBody
    public String updateDefaultAddress(@PathVariable final String id) throws DuplicateUidException
    {
        final AddressData address = userFacade.getAddressForCode(id);
        userFacade.setDefaultAddress(address);
        return "Address was updated successfully by method from sampleExtendedCustomerController";
    }
...
}
```

Assigning New Calls to a Specific API Version

Context

There are two versions available for Commerce Services (**v1** and **v2**). Both versions are available simultaneously. As a result, you might need to specify the version to call from the AddOn. To do this, you can use the `ApiVersion` annotation.

```
/**
 * Annotation can be used for controllers. It allows restrict visibility of methods annotated with
 * {@code @RequestMapping} only to selected version of commerce web services (e.g. v1 or v2).
 */
@Target(
{ ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface ApiVersion
{
    /**
    * Returns API version for which methods from controller should be registered (e.g. v1).
    */
    String value();
}
```

This annotation can be used at the `controller` level.

```
@Controller("sampleExtendedCustomerController")
@RequestMapping(value = "/{baseSiteId}/customers")
@ApiVersion("v1")
public class ExtendedCustomersController
{
...
}
```

ℹ Note

If there is no `@ApiVersion` annotation assigned to the controller in the AddOn, methods from this controller are added for all available versions.

For example - method annotated by `@RequestMapping(value = "/{baseSiteId}/customers/current/nickname")` are available from `URI` `/rest/v1/{baseSiteId}/customers/current/nickname` and `/rest/v2/{baseSiteId}/customers/current/nickname`.

Extending Server-Side Caching

Server-side caching is available for AddOns. In order to add your own cache to the existing configuration, use the `wsCacheManagerList` bean to store all cache managers (implementations of `org.springframework.cache.CacheManager` interface) that are in use.

Procedure

Use the **List Merge Directive** to extend the list from the AddOn.

occaddon-web-spring.xml

```
<!-- Cache manager for occaddon -->

        <alias name="defaultOccAddonCacheManager" alias="occAddonCacheManager"/>
        <bean id="defaultOccAddonCacheManager" class="org.springframework.cache.ehcache.EhCacheCacheManager">
                <property name="cacheManager" ref="occAddonEhcache"/>
        </bean>

        <alias name="defaultOccAddonEhcache" alias="occAddonEhcache"/>
        <bean id="defaultOccAddonEhcache" class="de.hybris.platform.commercewebservicescommons.cache.TenantAwareEhCacheManagerFactoryB
                <property name="cacheNamePrefix" value="occAddonCache_"/>
                <property name="configLocation" value="/WEB-INF/cache/addons/occaddon/ehcache.xml"/>
        </bean>

        <bean depends-on="wsCacheManagerList" parent="listMergeDirective">
                <property name="add" ref="occAddonCacheManager" />
        </bean>
```

This example is available in the `occaddon`.

i Note

Remember that all beans described here must be placed in the `ADDON_NAME/resources/ADDON_NAME/web/spring/ADDON_NAME-web-spring.xml` file.

In the example, the Ehcache library is used by default. You can replace the library with any other cache library that is supported by the Spring Framework (i.e. Guava, GemFire, JSR-107).

The last thing you have to do (when using default configuration generated by `yoccaddon` template) is modify the `ehcache.xml` file located in the `ADDON_NAME/acceleratoraddon/web/webroot/WEB-INF/cache` directory and customize it to your needs.

Related Information

Caching

Extending Message Bundles

You can use an AddOn to define a message bundle that is available in Commerce Services.

Procedure

Define the message bundle in the `/acceleratoraddon/web/webroot/WEB-INF/messages` directory.

ExtendedUsersController.java

```
@Controller("sampleExtendedUserController")
@RequestMapping(value = "/{baseSiteId}/users")
@ApiVersion("v2")
public class ExtendedUsersController
{
...
    @Resource(name = "messageSource")
    protected MessageSource messageSource;
...

    @ResponseStatus(value = HttpStatus.PAYMENT_REQUIRED)
    @ResponseBody
    @ExceptionHandler({ WebserviceValidationException.class })
    public ErrorListWsDTO handleWebserviceValidationException(final WebserviceValidationException ex)
    {
        final ErrorListWsDTO errorListDto = handleErrorInternal(ex);
        return errorListDto;
    }
    protected ErrorListWsDTO handleErrorInternal(final WebserviceValidationException ex)
    {
        final ErrorListWsDTO errorListDto = new ErrorListWsDTO();
        final Locale locale = i18nService.getLocaleForLanguage(i18nService.getCurrentLanguage());
        final Errors errors = (Errors) ex.getValidationObject();
        final List<ErrorWsDTO> errorsList = errors.getAllErrors().stream().map(eo -> mapError(eo, locale))
                .collect(Collectors.toList());
```

```
        errorListDto.setErrors(errorsList);
        return errorListDto;
    }
    protected ErrorWsDTO mapError(final ObjectError error, final Locale locale)
    {
        final ErrorWsDTO result = new ErrorWsDTO();
        final String message = messageSource.getMessage(error.getCode(), error.getArguments(), locale);
        result.setMessage(message);
        result.setReason("Validation failed");
        result.setType("Error");
        return result;
    }
}
```

All files from this location are copied during the build phase into the `/web/webroot/WEB-INF/messages/addons/addonName/` directory of Commerce Services and automatically loaded by the `MessageSource` implementation.

The message bundle defined in the AddOn is visible in class from Commerce Services, which uses the `messageSource` bean.

A common `messageSource` bean is used by AddOn controllers. The example uses `messageSource` for exception handling.

`AddonAwareMessageSource` class is a custom implementation of `MessageSource` interface delivered by Commerce Services. It is configured to scan target directory `/WEB-INF/messages/addons/` for any `xml` and `properties` files that can be used as message bundles. Scanning and indexing files is done once during bean initialization.

`AddonAwareMessageSource` provides the following extra properties for customization:

- baseAddonDir - usually `/WEB-INF/messages/addons/`, it scans for all files in this directory and its subdirectories.
- fileFilter - filter for files that should be loaded. By default, the filter loads all `xml` and `properties` files.
- dirFilter - filter for subdirectories. By default, all subdirectories are scanned.

Related Information

Generating Beans and Enums