Commerce

Generated on: 2024-12-03 14:02:18 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.

Base Commerce Module

The Base Commerce module provides a wide range of functions within SAP Commerce, including payment, customer reviews, returns, refunds, store locator, stock service, and more.

Features

Base Store

Customer Reviews

Customer Services

Deep Linking

Fraud Detection

Replenishment and Order Scheduling

Order History and Order Versioning

Order Splitting

Stock Service

Store Locator Implementation

Warehouse Integration

Promotions (Legacy)

Vouchers (Legacy)

Architecture

basecommerce Extension

basecommercebackoffice Extension

customerreview Extension

payment Extension

promotions Extension

promotionsbackoffice Extension

voucherbackoffice Extension

voucher Extension

Implementation

Store Locator Implementation

Payment Integration

Base Commerce Module Features

The features offered by the Base Commerce module include payment services, customer reviews, return service, refund service, order history and versioning, warehouse integration, replenishment and order scheduling, store locator, and stock service.

Base Store

The Base Store forms the basis for your e-commerce setup and is used to model your online store. You can create and manage base stores in the Backoffice.

Customer Reviews

Customer Reviews allow you to collect and manage customer ratings and reviews on your storefront.

Customer Services

Customer Services includes an order cancel service, return service, and refund service.

Deep Linking

The Deep link URL feature provides an easy way of mapping URLs to corresponding destination URLs.

Fraud Detection

Fraud Detection provides extensible services for performing fraud checks internally as well as interfaces to integrate external fraud detection providers.

Replenishment and Order Scheduling

Replenishment and Order Scheduling are technically realized by the B2B Commerce Module and the `basecommerce` extension. Functionally, these features are a part of the Order Management Module and are designed mainly for the B2B market. It provides several possible ways to schedule the creation of new orders or recurring orders. For more information about order replenishment, see B2B Orders.

Order History and Order Versioning

Order History and Order Versioning provide a simple way of storing information about order-related actions. Order Versioning makes persistent snapshots of an order, while Order History tracks changes made to the order itself.

Order Splitting

Order Splitting allows orders to be broken down into several consignments and warehouse interfaces, which permits partial shipments and effective warehousing. Order Splitting is a part of the `basecommerce` extension.

Payments

Payments enables you to integrate external payment service providers to handle electronic payments, offering a flexible approach to multi-channel online payment methods.

Stock Service

Stock Service is part of the `basecommerce` extension. It offers functionality to manage and query product stock level and product availability information, aggregated or for a specific warehouse.

Store Locator

The SAP Commerce Store Locator helps customers find brick-and-mortar retail locations in the proximity of a postcode, or using Global Positioning System (GPS) information. Store Locator is an addon for Cart & Checkout, and works with both desktop and mobile solutions.

Warehouse Integration

The Warehouse Integration interface supports communication between a warehouse and SAP Commerce during the Order Management process.

Promotions (Legacy)

The promotions module has been replaced by Promotion Engine.

Vouchers (Legacy)

Vouchers enable you to create and manage vouchers redeemable by your customers.

Base Store

The Base Store forms the basis for your e-commerce setup and is used to model your online store. You can create and manage base stores in the Backoffice.

The base store forms the core foundation of your e-commerce instance, as it is the key entity that models a store. The base store defines many of the key aspects of your ecommerce presence, such as languages, currencies, products, tax handling, and more. Carts and orders created via the storefront are associated with a base store.

Features

Essential Store Management

The base store is where you manage the essential features and properties of your e-commerce store. The base store is where you define currencies, languages, and permitted shipping and billing countries. You associate a catalog with the base store to determine which products are sold on that store. In addition, you also use the base store to define, for example, tax handling.

Data Model Flexibility

Each base store is associated with one or more websites (also called base sites). Many customers set up their data model with a single base store representing their business, with potentially multiple websites (base sites) underneath. For example, you can have one website for the English web shop and the other website for the German web shop, both of which use the same base store.

Alternatively, you can also have one website that contains multiple base stores.

You can use multiple base stores to represent different channels within a multichannel system. For example, you might have one base store for the website, and another base store for an integration with external sites also selling their products.

Related Information

basecommerce Extension

Setting Up a Basic Web Site in the Backoffice

Base Store Management

Manage base store settings in the Backoffice.

Base Store Management

You can create new base stores and manage base store settings in the Backoffice. To access this area, navigate to **Base Commerce** > **Base Store**, then select the base store that you want to edit.

Properties

On the **Properties** tab, create the following settings:

Field / Setting	Description
ID	Enter a unique ID for the base store. The ID is used in Backoffice to manage the base store.
Name	Enter a descriptive name for the base store. This field can be localized.
Net	Select whether prices on the storefront are displayed as gross or net prices. • **True:** The storefront displays the net price of products. The net price is the price before any taxes are added. • **False:** The storefront displays the gross price of products. The gross price is the price including tax.
Default Currency	Select the default currency of the store. In the absence of other information about the currency of the customer, the store displays the default currency.
Default Language	Select the default language of the store. In the absence of other information about the language of the customer, the store displays the default language.
Catalogs	Select the catalogs associated with this base store. This selection determines which products are sold on this store.
Currencies	Select the currencies available on this store. The selected currencies appear in the dropdown currency selection menu on the storefront.
Websites	Select the websites (base sites) that use this base store. You can select multiple websites. For more information, see Base Store.

Field / Setting	Description
Warehouses	Select the warehouses associated with this base store. Products purchased on this base store are sourced from the selected warehouses. Product availability displayed on the storefront reflects availability only in the warehouses stored here.
Customer Allowed to Ignore Suggestions	Select whether customers are allowed to ignore suggestions from an integrated AVS (Address Verification System). • **True**: Customers may ship to an address that cannot be verified by an integrated AVS. • **False**: Customers must use an address confirmed by the AVS.
Delivery Countries	Select the countries that products can be shipped to. These countries appear in the delivery country selection menu on the storefront.
Billing Countries	Select possible billing countries for this base store. These countries appear in the billing address selection menu on the storefront.
External Tax Enabled	Select whether an external tax calculation service (if integrated) is enabled or disabled for this base store. • **True**: The integrated third-party tax calculation service is enabled for this base store. • **False**: The integrated third-party tax calculation service is disabled for this base store.
Payment Provider	Enter the name of the payment provider as it appears in the Spring-configured list of payment providers. Note that if you change the payment provider, the new provider only comes into effect for future orders. Previous orders are not affected. For more information about payment providers, see payment Extension .
Pickup in Store Mode	Out of the box, only **Buy and Collect** is supported. The option **Reservce and Collect** could be used to enable online reservation and in-store payment, but requires customization.
Submit Order Process Code	Enter an order process code. The order process determines the sequence of events that take place when an order is placed on this base store. For more information, see Ordering Process.
Tax Group	Select the tax group associated with this base store. For more information, see Tax Calculation.
Storelocator Distance Unit	Select whether the storefront store locator displays distance to the nearest stores in km or miles.
Enable Express Checkout	Select whether customers can use express checkout on this base store. Express checkout allows customers to place an order using stored address and payment details. For more information, see Express Checkout.

Locations

On the Locations tab, create the following settings:

Field / Setting	Description
Locations	Select the points of service that are associated with this base store. These stores appear in the store locator on the website. For this field, you should only select points of service with the type Store.
Default Delivery Origin	Select the default warehouse that products sold on this base store are shipped from. For this field, you should only select points of service with the type Warehouse.

Administration

On the Administration tab, create the following settings:

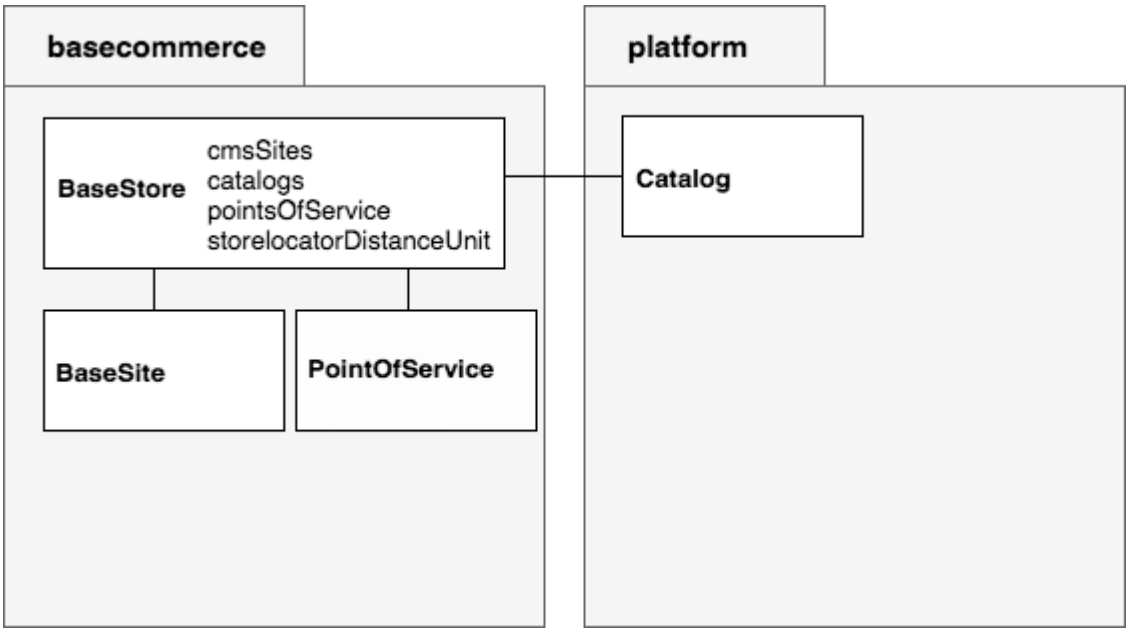
Field / Setting	Description
Checkout Flow Group	Enter the checkout group for this base store. For more information, see Configurable Checkout.
Delivery Modes	Select delivery modes available on this base store.
Languages	Select all of the languages on this base store. These languages appear in the dropdown menu on the storefront.
Max Radius for POS Search (kms)	Limits the maximum radius searched by the store locator.

Base Store Data Model

Learn more about the relationship between the base store, base site, and points of service.

The `BaseStore` type is a part of the `basecommerce` extension. The `BaseStore` type is used by the other extensions including the `CMS2` extension and the `promotions` extension.

The following image displays the link between the `basecommerce` extension and the SAP Commerce Platform:



A BaseStore contains the following attributes:

Attribute Name	Used For
`uid`	Unique identifier of a store.
`name`	Name of a store (localized).
`catalogs`	Product and classification catalogs which are available in the store. **Note** Do not assign a content catalog to a store. A content catalog is only assigned to a CMSSite.
`cmsSites`	Provide the look and feel for a product set displayed on a certain site.
`pointsOfService`	The list of local stores that will appear in the storefront's store locator.
`storelocatorDistanceUnit`	A selection from the DistanceUnit emum, either "km" or "miles".

Creating a Base Store

Create a new base store. The base store forms the basis for your e-commerce setup and is used to model your online store.

Procedure

1. Log on to the Backoffice and navigate to Base Commerce > Base Store.
2. Click the plus sign to create a new base store.
3. In the window, enter the settings.

Setting	Description
Captcha Widget Enabled	The captchaaddon AddOn introduces a reCAPTCHA widget to the customer registration form of the Accelerator storefronts.

Setting	Description
	This technology is used to block spammers and bots that try to automatically harvest e-mail addresses, or that try to automatically sign up for (or make use of) websites, blogs, or forums. For more information, see captchaaddon AddOn.
Enable Express Checkout	Select whether customers can use express checkout on this base store. Express checkout allows customers to place an order using stored address and payment details. For more information, see Base Store Management.
Time Created	Enter a creation date and time for the base store.
ID	Enter a unique ID for the base store. The ID is used in Backoffice to manage the base store.
Net	Select whether prices on the storefront are displayed as gross or net prices. ◦ **True**: The storefront displays the net price of products. The net price is the price before any taxes are added. ◦ **False**: The storefront displays the gross price of products. The gross price is the price including tax.
Tax Estimation Enabled	Select whether tax estimation is enabled for this store. This allows the customer to view estimated taxes before completing the checkout process. Tax estimation requires the customer's country and zipcode, which are stored in the session.
External Tax Calculation	Select whether an external tax calculation service (if integrated) is enabled or disabled for this base store. ◦ **True**: The integrated third-party tax calculation service is enabled for this base store. ◦ **False**: The integrated third-party tax calculation service is disabled for this base store.

4. Click **Done** to create the new base store.
5. Select the base store to continue editing the store and determining additional settings. For more information, see Base Store Management.

Customer Reviews

Customer Reviews allow you to collect and manage customer ratings and reviews on your storefront.

The `customerreviews` extension enables customer rating and review forms on your storefront.

Use Case

Customers are able to provide product feedback and product ratings for products they have purchased in your online shop.

As the owner of an online shop, you are able to approve or reject comments to control which ratings and reviews are displayed on the shop.

Features

Manage Customer Reviews in the Backoffice

You can view, edit, approve, and reject customer reviews in the Backoffice. For more information, see Managing Customer Reviews or Managing Customer Reviews per Product.

Related Information

customerreview Extension

Managing Customer Reviews

View, edit, approve, and reject customer reviews in the Backoffice.

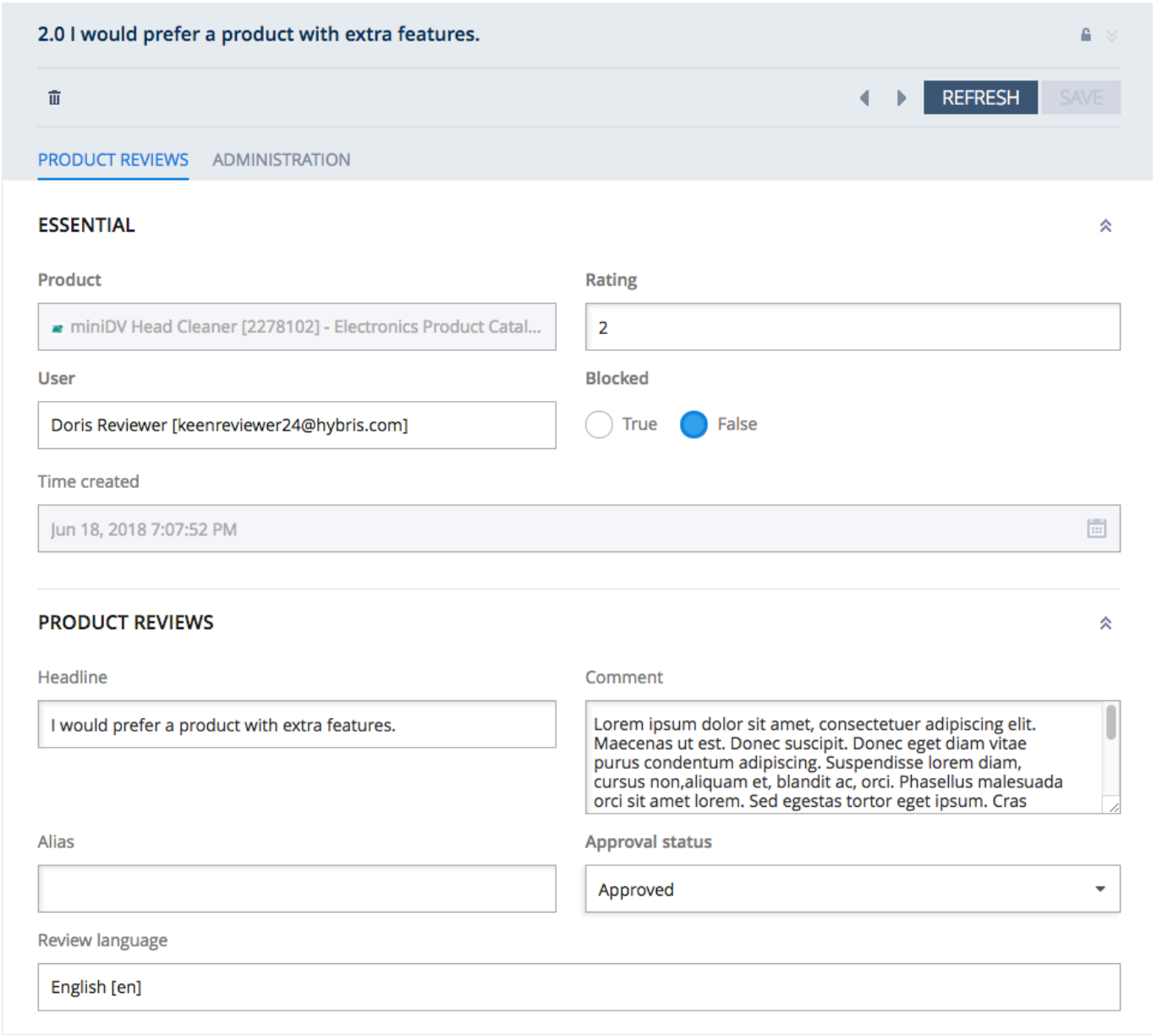
Procedure

1. Log on to the Backoffice and navigate to **Marketing** > **Product Reviews**.

   A list of current product reviews appears.

2. Select the customer review that you want to approve, reject, or edit.

   The review opens to the **Product Reviews**.



3. In the Approval Status field, change the status to Approved, Pending, or Rejected. You can also edit the review.
4. Click Save to save your changes.

Managing Customer Reviews per Product

In the Backoffice you can view all customer reviews available for a product and edit, delete, approve, and reject reviews.

Context

Every product has a Reviews tab that displays the total number of reviews, the average rating, and a complete list of reviews. You can approve or reject reviews, edit the individual review details, or delete a review.

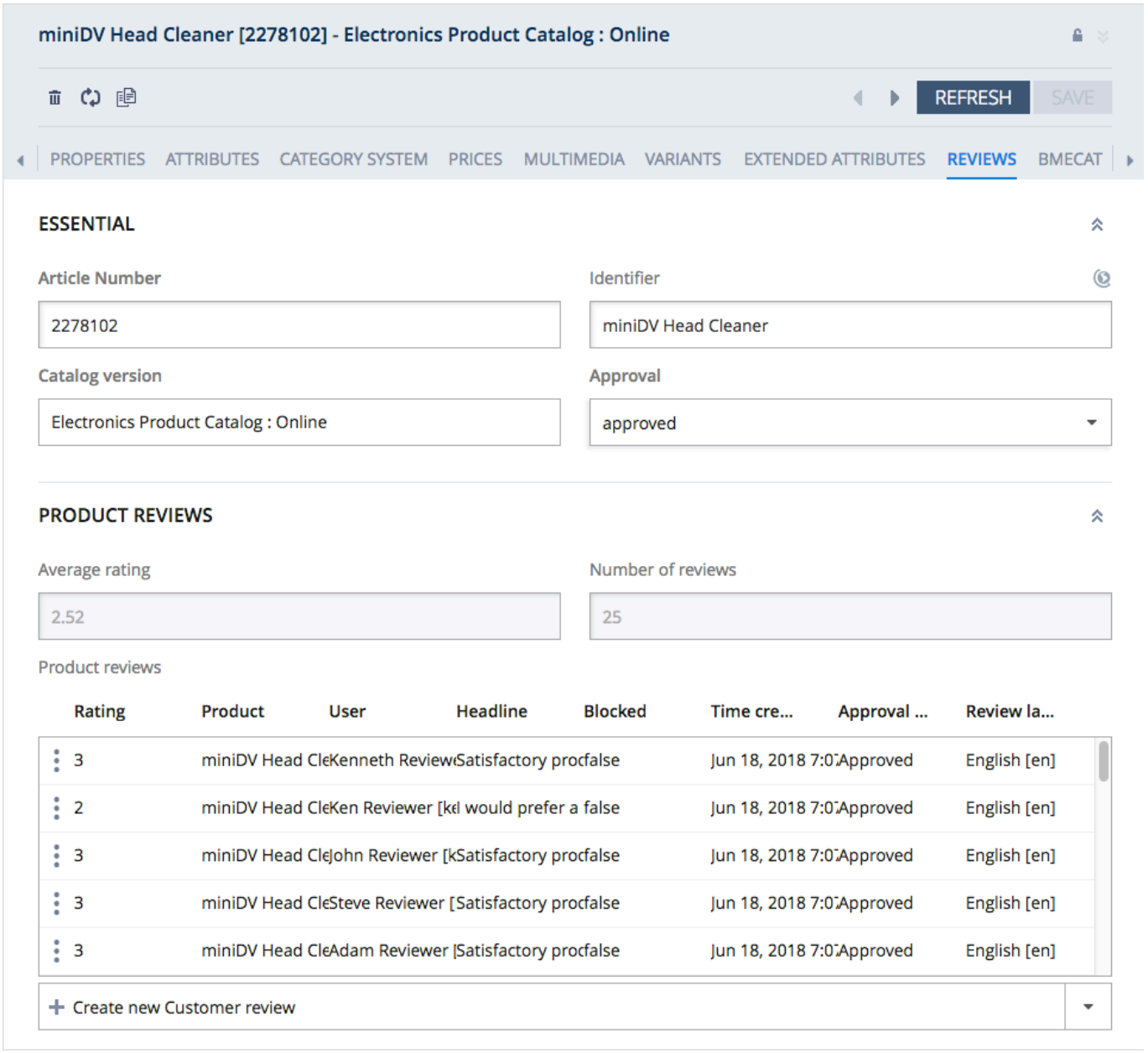
Procedure

1. In the Backoffice, navigate to Catalog > Products.

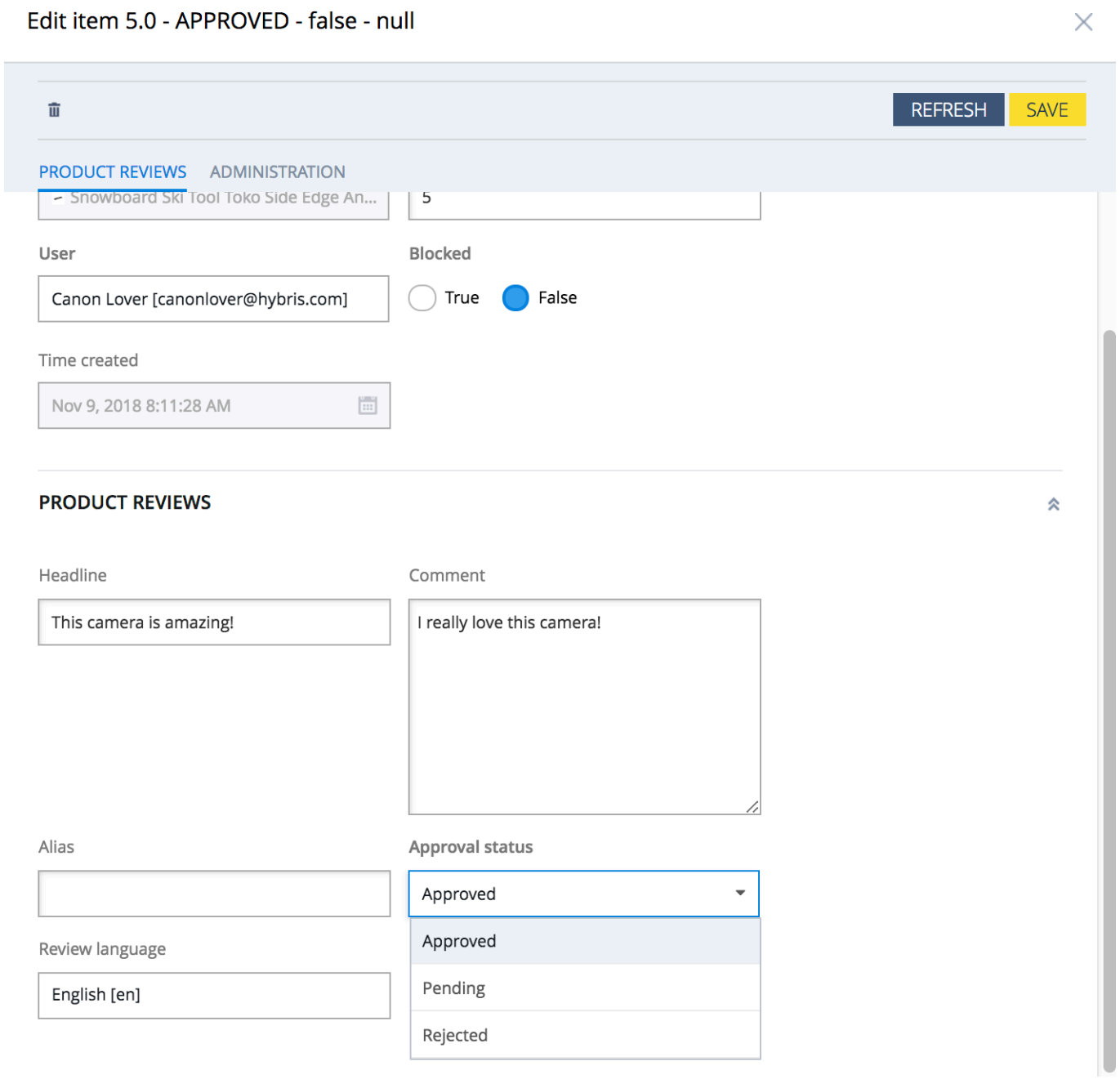
   A list of products appears.

2. Select the products for which you want to approve, reject, or edit reviews.
3. Open the Reviews tab.

The Reviews tab shows all available reviews for this product.



4. Click Remove to delete the review. Click Edit Details to open a window where you can edit, approve, or reject the review. Note that the field Approval in the Essentials area does not refer to the product review, but to the product itself. To approve or reject the review, click Edit Details.



Edit, Approve, or Reject a Product Review

5. Click Save to save your changes.

Customer Services

Customer Services includes an order cancel service, return service, and refund service.

Use Case

Customers in your ecommerce shop expect to be able to cancel orders, return items, and obtain a refund for returned items. These services are implemented as a part of the `basecommerce` extension.

Features

Order Cancel Service

Allows customers to cancel orders, or partially cancel orders. Orders can be cancelled only if they have not yet been fulfilled. Order cancellation is not available for digital products, which are fulfilled immediately.

As a shop owner, you can customize the rules for order cancellation - for example, to not permit customers to cancel orders at all.

Return Service

Allows customers to return goods to the seller.

Refund Service

Enables a customer to get a refund after returning goods to the seller.

Related Information

basecommerce Extension
Order Cancel Service
Return Service
Refund Service

Order Cancel Service

This service allows customer service agens to cancel orders. Order cancellation supports the Order Management module, but is technically realized in the `basecommerce` extension.

Configuring the Order Cancellation Process

You can configure order cancellation in the Backoffice. See Interfaces and Methods.

> **Note**
> The Order Cancel service only allows customer service agents to cancel orders on behalf of customers. Functionality to allow customers to cancel orders on the storefront is provided with the `orderselfserviceaddon`, which is a part of the Order Management module. For more information, see orderselfserviceaddon.

How Cancellation Works

A customer service agent can cancel an order completely as long as it has not yet been fulfilled.

Depending on the status of the order, order cancellation works as follows:

- **Order has been placed in the system, but not yet sent to the warehouse.**

  If an order has not been sent to the warehouse yet, the cancellation process removes the order from the queue of orders which have to be sent to the warehouse. The order is marked as canceled. The system also removes any amount reservation on the payment means used for the transaction.

- **Order has been sent to the warehouse, but is not fulfilled (or only partly fulfilled).**

  If the order has been sent to the warehouse already but no update about fulfillment has been received, a cancellation message is sent to the warehouse immediately. In case the cancel request overlaps with the fulfillment notification from warehouse, the cancel request is canceled or only applies for those items which have not yet been fulfilled. As a result, full cancellation becomes a partial cancellation. After receiving a confirmation from the warehouse, any reservation on the credit card is removed.

Digital Products and Subscriptions

Digital products cannot be canceled since their fulfillment is immediate. Therefore digital products are excluded from any cancellation process.

If an order contains a subscription, then the subscription can be canceled according to the subscription cancellation rules.

Partial Cancellation

Depending on the settings, you may allow cancellation of parts of an order.

If the order has already been sent to the warehouse, then the system sends an update for this order.

If the order total is higher then the original order total, then the customer must provide the payment details again in order for the system to continue with reauthorization and reservation.

After any partial cancellations, the system recalculates the value of the order. If needed, the system sends updates to the warehouse.

ℹ Note

After a partial order cancellation, the order is recalculated completely, using any promotions or other parameters currently valid in the system. As a result, the order may lose some previous promotional discounts, or new promotions may apply.

Related Information

basecommerce Extension

Order Cancellation Configuration

You can configure order cancellation in the Backoffice.

Context

These settings control what forms of order cancellation are possible.

Procedure

1. Log on to the and navigate to **Base Commerce** > **Order Cancellation Configuration**.
2. Create the settings as follows:

Setting	Description
Enable customer order cancel	Determines whether order cancelling is possible at all.
Enable partial order cancel	Determines whether it is possible to cancel parts of an order (the customer can cancel a complete order entry).
Enable partial order entry cancel	Determines whether it is possible to cancel part of an order entry. This parameter is only evaluated if **Enable partial order cancel** is set to true.
Enable complete cancel after shipping	Determines whether a cancel request is possible after shipping of part of the order has started. Only the the part of the order that has not been shipped yet can be cancelled. This parameter is evaluated only if **Enable warehouse order cancel** is set to true.

Setting	Description
Enable warehouse order cancel	Determines whether order cancellation is still possible after the order has been sent to the warehouse.
Order queue waiting time	Determines how long an order waits in queue before it is started to be fulfilled. This allows customers more or less time to cancel the order.

3. Click Save to save your changes.

Interfaces and Methods

Order Cancel Service is defined by interfaces that enable access to order cancel functionality and support communication to and from the Warehouse. Several methods allow access to data model and configuration, and perform order cancel execution logic. State-based processing is based on predefined set of states. Default service configuration can be modified in order to adapt it to the particular requirements.

Introduction

Order Cancel service is defined using two interfaces:

- **OrderCancelService**
- **OrderCancelCallbackService**

OrderCancelService interface is the main service interface of Order Cancel service. It is designed to be used by "business user" which technically is expected to be another service or a web shop implementation.

The role of this interface is to enable access to:

- Order Cancel related functionality (e.g. performing order cancels)
- Order Cancel data model and configuration

The main class to be used with **OrderCancelService** interface is **OrderCancelRequest** class. This class allows to construct cancel requests and may represent either complete cancel request or partial cancel requests.

OrderCancelCallbackService interface is a callback interface to be used by a Warehouse adapter. Warehouse adapter is an interface between hybris platform and physical Warehouse. The role of **OrderCancelCallbackService** interface is to receive asynchronous messages from a Warehouse in response to a previously-issued cancel requests. Key class used with **OrderCancelCallbackService** interface is **OrderCancelResponse** class. This class represents a response sent by the Warehouse. The response contains order cancel results information, for example whether the cancel was successful or not and why.

Methods in OrderCancelService Interface

Methods for Data Model and Configuration Access

- **OrderCancelConfigModel getConfiguration()**: Returns current configuration.
- **OrderCancelRecordModel getCancelRecordForOrder(OrderModel order)**: Returns OrderCancelRecordModel for given Order. This class is a main class of the Order Cancel Data Model.
- **OrderCancelRecordEntryModel getPendingCancelRecordEntry(OrderModel order) throws OrderCancelException**: Returns OrderCancelRecordEntryModel instance for currently pending cancel operation. This **OrderCancelRecordEntryModel** class represents data model for a single order cancel operation.

Methods Allowing Performance of the Order Cancel Execution Logic

- **CancelDecision isCancelPossible(OrderModel order, PrincipalModel requestor, boolean partialCancel, boolean partialEntryCancel)**: Verifies if order cancel is possible for given conditions. This method should always be called first, before requestOrderCancel to check for order cancel availability!
- **OrderCancelRecordEntryModel requestOrderCancel(OrderCancelRequest orderCancelRequest, PrincipalModel requestor)**: Issues order cancel request. You can call this method only if isCancelPossible allows the cancel, otherwise OrderCancelDeniedException will be thrown.
- **Map<AbstractOrderEntryModel, Long> getAllCancelableEntries(OrderModel order, PrincipalModel requestor)**: This method provides fine-grained information about which order entries can be canceled.

Order Cancel Service Implementation Details

State-based Processing

Order Cancel Service logic is based on predefined set of states, defined by **OrderCancelState** enum type. These states are derived from the state diagrams that provide conceptual model for processing order cancels.

There are several states defined by **de.hybris.platform.basecommerce.enums.OrderCancelState**. The list contains available states, the names in parentheses denote corresponding state from a state diagram:

- **PENDINGORHOLDINGAREA** (QueueAwait)
- **SENTTOWAREHOUSE** (SentToWarehouse)
- **SHIPPING** (Shipping)
- **PARTIALLYSHIPPED** (PartiallyShipped)
- **CANCELIMPOSSIBLE** (CancelImpossible)

Additionally a new state has been defined for enum **OrderStatus**: CANCELLING, which is used to model "Cancelling" state from the diagrams.

Order Cancel Service uses a strategy: **OrderCancelStateMappingStrategy**, that is responsible for determining current **OrderCancelState** for a given order.

The default mapping strategy is described in the OrderCancel - Default Mapping Strategy document. This strategy is defined as a Spring bean, and it can be easily changed.

Default Service Implementation Configuration

Default **OrderCancelService** implementation: **DefaultOrderCancelService** uses a set of strategies to implement the required functionality. Most important objects are:

- **OrderCancelStateMappingStrategy**: Provides information about current OrderCancelState for the Order.
- **OrderCancelDenialStrategy**: Enables to perform checks if cancel is possible for given conditions.
- **OrderCancelRequestExecutor**: An object used to actually execute the logic involved with canceling order.
- **OrderCancelResponseExecutor**: An object used to process messages received from a Warehouse in response to a cancel request.
- **OrderCancelCancelableEntriesStrategy**: A strategy used to get a list of the given order entries that can be canceled.

All these objects are defined as Spring beans. As a result altering the Spring configuration can easily influence **DefaultOrderCancelService** and change the way it works.

Executor Mappings

DefaultOrderCancelService does not have "fixed" execution logic. All actions are performed by so-called executors. There are two types of executors:

- **OrderCancelRequestExecutor**
- **OrderCancelResponseExecutor**.

Request executors are used to process cancel requests received via **OrderCancelService** interface. Response executors are used to process cancel responses received via **OrderCancelCallbackService**.

DefaultOrderCancelService has two maps of executors: requestExecutorsMap and responseExecutorsMap. Each map provides a mapping between current OrderCancelState and an executor that handles order cancel logic in this state. Both maps are configured by Spring, so this gives a very flexible way to configure overall logic of **DefaultOrderCancelService**.

Verifying Possibility of the Order Cancel: OrderCancelDenialStrategy

DefaultOrderCancelService class uses a List of **OrderCancelDenialStrategy** instances to verify if cancel is possible for a given Order. There is a couple of predefined **OrderCancelDenialStrategy** classes. Also, new classes may be added if needed.

The Spring configuration defines which **OrderCancelDenialStrategy** classes should be used by **DefaultOrderCancelService**. If **OrderCancelDenialStrategy** finds that order cancel cannot be performed, then it returns non-null **OrderCancelDenialReason**. Otherwise it returns null.

Because **DefaultOrderCancelService** processes a list of **OrderCancelDenialStrategy**, cancel operation is allowed if all strategies return null result. If any strategy returns non-null **OrderCancelDenialReason**, then cancel operation is not possible, and the list of received **OrderCancelDenialReason** is returned to the user.

In hybris implementation every concrete **OrderCancelDenialStrategy** can have its **OrderCancelDenialReason** configured by Spring. Users can do the mapping between **OrderCancelDenialStrategy** and **OrderCancelDenialReason** as they require. By default we provide **DefaultOrderCancelDenialReason** class that is a reasonable implementation of the **OrderCancelDenialReason** interface. However, it is possible to easily plug in another implementation, if needed, for example when **OrderCancelDenialReason** instance should convey additional information or be able to execute some logic.

Other Collaborating Objects

OrderCancelNotificationServiceAdapter

This interface is used to broadcast notifications about cancel operations. It is used by classes: **ImmediateCancelRequestExecutor**, **WarehouseProcessingCancelRequestExecutor**, **WarehouseResponseExecutor**. Order Cancel Service does not provide default implementation of this interface, because no reasonable default can be defined. Users should supply their own implementation and plug it in using Spring configuration for the aforementioned classes. If not provided, notifcations are not sent.

OrderCancelPaymentServiceAdapter

This interface is used to recalculate order after cancel operation has been finished. It is used by classes: **ImmediateCancelRequestExecutor**, **WarehouseResponseExecutor**. Order Cancel Service does not provide default implementation of this interface. Users should supply their own implementation and plug it in using Spring configuration for the aforementioned classes. If not provided, orders are not automatically recalculated by Order Cancel Service after cancel operation is finished (order recalculation can be then performed externally to Order Cancel Service)

OrderCancelWarehouseAdapter

This interface is used to forward cancel requests to a warehouse. It is used by class: **WarehouseProcessingCancelRequestExecutor**. A mock implementation (**DefaultWarehouseAdapterMock**) is provided by default.

Users should supply their own implementation and plug it in using Spring configuration for the aforementioned class.

Order Cancel Service Configuration

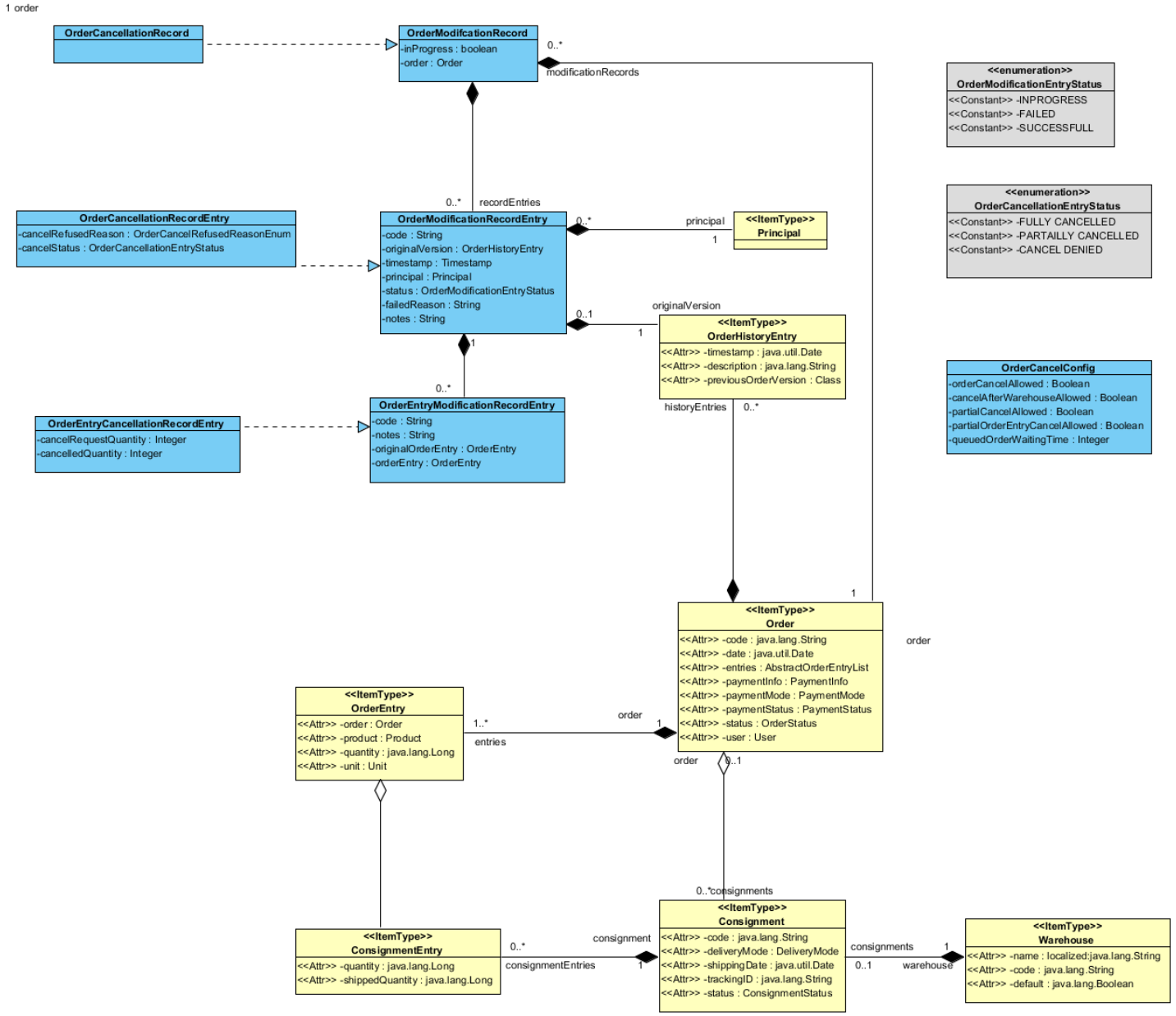
The following configuration entries are defined for Order Cancel Service using **OrderCancelConfigModel** class:

- **orderCancelAllowed**: Determines if the order canceling is possible.
- **cancelAfterWarehouseAllowed**: Determines if the order canceling is still possible after sending it to warehouse.
- **completeCancelAfterShippingStartedAllowed**: Determines if complete cancel request will be accepted after shipping has started. Such a request means: Cancel the part of the order that has not been shipped yet (in other words, cancel everything that can still be canceled). This parameter is evaluated only if **cancelAfterWarehouseAllowed** is set to true.
- **partialCancelAllowed**: Determines if the partial order canceling (discarding whole order entries) is possible.
- **partialOrderEntryCancelAllowed**: Determines if the partial order entry canceling (discarding parts of order entries) is possible. This parameter is evaluated only if **partialCancelAllowed** is set to true.
- **queuedOrderWaitingTime**: Determines how long an order should wait in queue before it is started to be fulfilled.

Almost all described entries are of type boolean. Only the **queuedOrderWaitingTime** is of type int. **queuedOrderWaitingTime** is defined as a part of Order Cancel configuration, but the queue itself must be implemented externally to Order Cancel Service. The reason for is that this Service does not provide any implementation for orders queue.

Data Model

The diagrams presents the data model for Order Cancel.



Related Information

- Order Cancel Complete State Diagram
- Order Cancel Partial State Diagram
- OrderCancel - Default Mapping Strategy

Order Cancel Complete State Diagram

The conceptual state diagram for handling complete Cancel Order requests captures essential logical states of the system and provides you with insight in its activities.

Order Cancel Complete State Diagram

The image presents conceptual state diagram for handling Complete Cancel Order requests. These states are only informal. The purpose of this diagram is to capture essential logical states of the system, not to define any concrete code-level constructs.

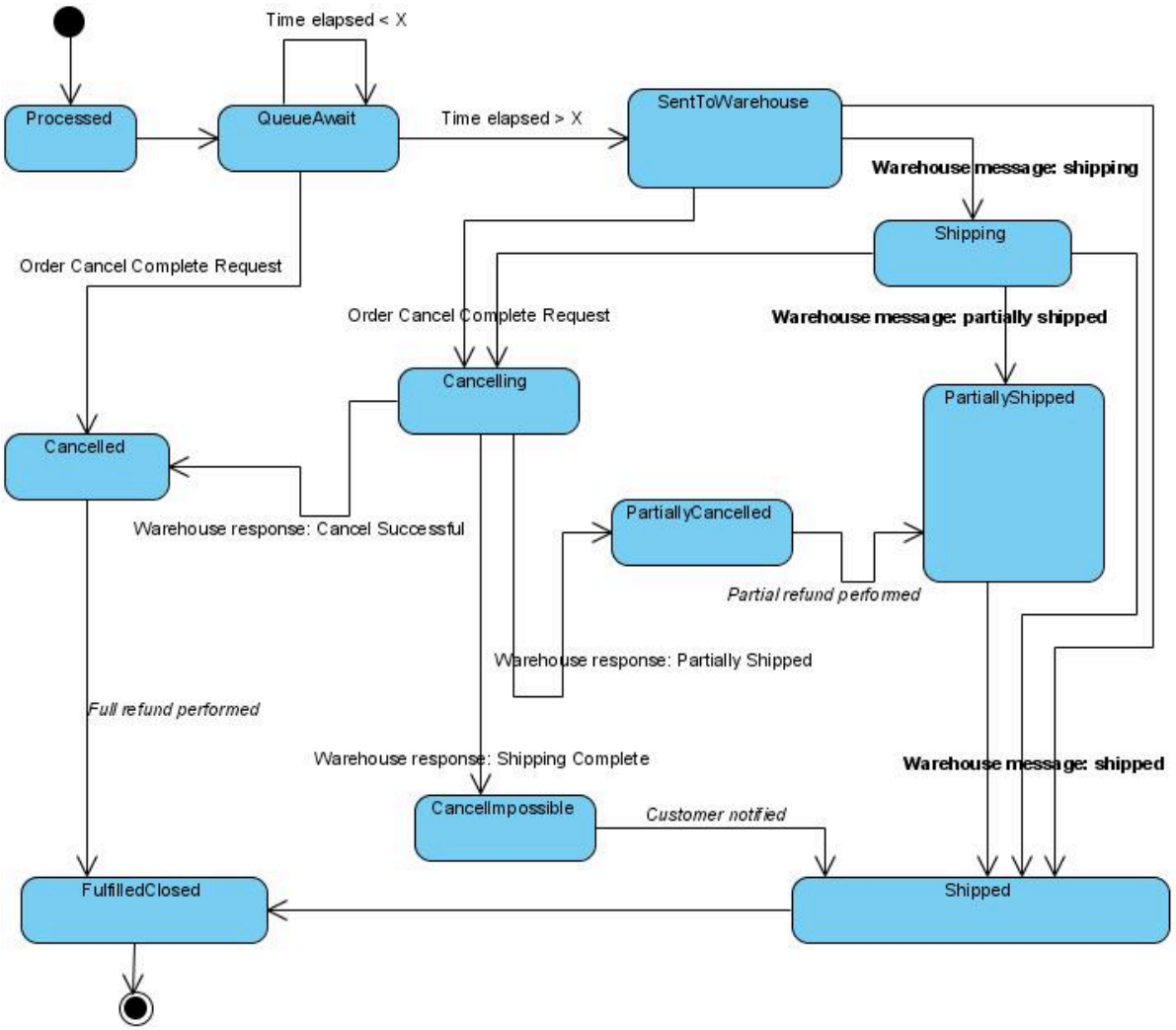


Diagram Elements Explanation

Actions

Some of Actions come from Customer/CSA, and some of them come from the Warehouse. In general, Warehouse can:

- Send responses to previously issued requests
- Send **messages** (indicated in **bold** font on the diagram) which are Warehouse-initiated asynchronous events, for example. messages sent after the Order has been shipped.

The assumption that messages are asynchronous is necessary to cover all possible use-cases. Those messages can, in fact, be implemented by warehouse polling and thus be synchronous. However, from the Cancel Order Service point of view this is irrelevant and messages should be considered asynchronous.

States

- **QueueAwait**: In this state Order is placed in an internal Order Queue and is awaiting for being sent to a Warehouse.
- **SentToWarehouse**: The state when order has left an internal Order Queue and is sent to Warehouse for further handling (shipment).
- **Shipping**: The state when order shipping process has been started by the Warehouse. This state does not imply that any order items has been physically shipped, but it does not exclude this possibility either.
- **PartiallyShipped**: The state when some (or all) order items are shipped.
- **Shipped**: The state when all order items are shipped.
- **Cancelling**: Order Cancel Service received **Order Cancel Complete Request** from the Client/CSA, but the Order has already been sent to the Warehouse for shipment. **Order Cancel Complete Request** is immediately forwarded to the

Warehouse and then Order Cancel Service waits for response from the Warehouse. Expected response time: several seconds to several hours.

- **PartiallyCancelled**: This state encapsulates functionality regarding partial shipment. In this case Order recalculation is necessary. This might be in general very complicated when connected with promotions. The steps to do in this state:
    - Modify Order according to data from the Warehouse (remove entries that were canceled, leave entries already shipped to the Customer)
    - Recalculate Order using some external service. Recalculation may involve promotions. The result of this step is the new money amount for the Order
    - Modify pending payment transaction with the new Order money amount.
    - Notify customer about Order status.
- **Cancelled**: An Order has been successfully canceled. No other cancel requests are available on such Order.
- **CancelImpossible**: An Order cannot be canceled. No other cancel requests are available on such Order.

Request Processing

Immediate Order Cancel

Order Cancel Complete Request can be always processed successfully in **QueueAwait** state.

Warehouse-dependent Cancel

SentToWarehouse and **Shipping** states require Warehouse interaction to perform the cancel. The difference between those two states is subtle. In most scenarios they can be treated as a single state. However, having two states for modeling shop-warehouse integration allows for grater flexibility. **SentToWarehouse** state means: warehouse has been notified to ship the Order. **Shipping** state means: warehouse acknowledges the order and starts shipping procedure.

For some time it may still be possible to stop the shipping procedure at the Warehouse after entering **Shipping** state (it depends on the Warehouse integration model). This is a reason why issuing **Order Cancel Complete Request** at this state is allowed.

In the **SentToWarehouse** and **Shipping** state Order Cancel Service forwards received **Order Cancel Complete Request** to the Warehouse and waits for Warehouse response in **Cancelling** state. The success of the cancel operation depends then on whether order item's shipment has not yet begun, as explained below.

In the **Cancelling** state Order Cancel Service waits for the Warehouse response. No further **Order Cancel Complete Request** are allowed until Warehouse response is received (i.e. only single cancel request at a time is allowed). The further processing of the Order is determined by the response. There can be three possible outcomes:

- The Order is completely canceled because it has not been shipped yet. This results with a transition to **Cancelled** state.
- The Order cannot be canceled, because it is too late. The Order has already been shipped (completely) to the Customer. This results with a transition to **CancelImpossible** state. In this state customer should be notified that the **Order Cancel Complete Request** has failed.
- The Order was only partially canceled, because a part of the Order has already been shipped. The remaining part of the Order that has not yet been shipped is canceled. This results with a transition to **PartiallyCancelled** state.

It is the warehouse integration model that decides if partial cancel in response to **Order Cancel Complete Request** is possible. The outcome described above assumes "fine-grained" warehouse implementation, but it is possible to get **Shipping Complete** instead of **Partially Shipped** response from the warehouse when Order is already partially shipped.

When is Order Cancel Complete Request not Allowed

- In **PartiallyShipped**, **Shipped**, **Cancelled**, **Fulfilled/Closed** states **Order Cancel Complete Request** cannot be satisfied, so it is not allowed.
- In the **Cancelling** state **Order Cancel Complete Request** is not allowed, because Order Cancel Service waits for a response to a previously issued request and only single request at the moment is supported.

- States: **CancelImpossible** and **PartiallyCancelled** are transitional states and should be handled in the same as way as **Cancelling** state

Other Issues

Concurrency

Because there are two (possibly more) sources of events for Cancel Order Service: Customer Front End/CSA and Warehouse, care must be taken to avoid race condition. For example in **SentToWarehouse** state the **Order Cancel Complete Request** and **Warehouse message: shipping** events may occur simultaneously. Proper concurrency handling must be implemented to avoid race condition. This is, however, outside of the scope of the Cancel Order Service.

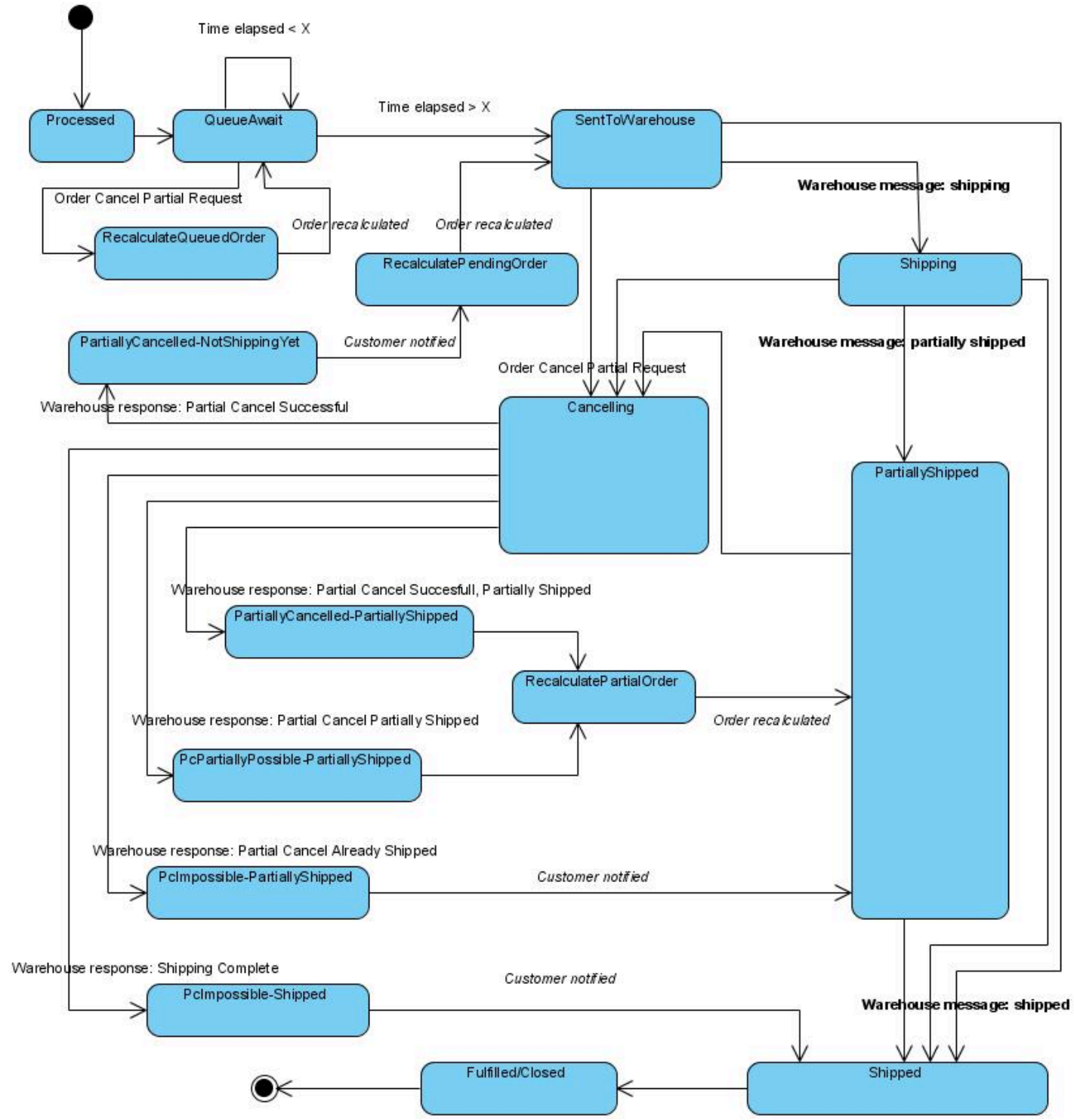
If the warehouse message: **shipping** or **partially shipped** or **shipped** is received while the Order is in the **Cancelling** state, the message should be ignored by the Cancel Order Service and the Service should wait for the proper response from the Warehouse.

Order Cancel Partial State Diagram

During the Partial Cancel process there can be several outcomes. This makes the case more complex than the Complete Cancel process.

Order Cancel Partial State Diagram

The diagram presents conceptual state diagram for handling Order Cancel Partial requests. These states are only informal. The purpose of this diagram is to capture essential logical states of the system, not to define any concrete code-level constructs.



Explanations

The case here is similar to the Order Cancel Complete case. However, due to the several possible outcomes the flow is also more complicated than with Complete Order Cancel. There are 6 possible **Order Cancel Partial Request** outcomes:

1. **"Success 1"**: **Order Cancel Partial Request** is received when Order is waiting in holding area before being sent to the Warehouse. Order entries are removed/modified according to the request, Order is recalculated and returned to holding area. Customer is notified. There is a question whether Order holding time should be reset after Order recalculation.
2. **"Success 2"**: **Order Cancel Partial Request** is received when Order is already sent to the Warehouse, but shipment has not yet begun. Partial Cancel request is forwarded immediately to the Warehouse and Order enters **Cancelling** state, where it waits for the response from Warehouse. When **Warehouse response: Partial Cancel Successful** is received, Order entries are removed/modified according to the request, Order is recalculated and the customer is notified.
3. **"Success 3"**: **Order Cancel Partial Request** is received when Order is already sent to the Warehouse, and part of it has already been shipped. However, the part of the Order that is subject to cancel has not been shipped yet.

Partial Cancel request is forwarded to the Warehouse and Order enters **Cancelling** state, where it waits for the response from Warehouse. When **Warehouse response: Partial Cancel Successfull, Partially Shipped** is received, Order entries are removed/modified according to the request, Order is recalculated and customer is notified.

4. **"Failure 1"**: **Order Cancel Partial Request** is received when Order is already completely shipped by the Warehouse but the Warehouse has not yet provided this information back to the shop. Partial Cancel request is forwarded to the Warehouse and Order enters **Cancelling** state, where it waits for the response from Warehouse. When **Warehouse response: Shipping Complete** is received, Customer is notified that Partial Cancel cannot be done. Order remains unchanged.
5. **"Failure 2"**: **Order Cancel Partial Request** is received when Order is already partially shipped by the Warehouse. Items that are requested to be canceled are already shipped and as a result cannot be canceled. Partial Cancel request is forwarded to the Warehouse and Order enters **Cancelling** state, where it waits for the response from Warehouse. When **Warehouse response: Partial Cancel Already Shipped** is received, Customer is notified that Partial Cancel cannot be done. Order remains unchanged.
6. "Partial Success": **Order Cancel Partial Request** is received when Order is already partially shipped by the Warehouse. Some cancel-requested items have already been shipped, so those items cannot be canceled. Remaining cancel-requested items are canceled. Partial Cancel request is forwarded to the Warehouse and Order enters **Cancelling** state, where it waits for the response from Warehouse. When **Warehouse response: Partial Cancel Partially Shipped** is received, Order is recalculated and Customer is notified that Partial Cancel was only partially completed. This particular case assumes fine-grained cancellation logic at the Warehouse. However, it is also likely that in the scenario described above, Warehouse will respond with a **Warehouse response: Partial Cancel Already Shipped** message, that indicates failure of the whole operation. Order Cancel Service is capable of handling both situations.

OrderCancel - Default Mapping Strategy

The **OrderCancelStateMappingStrategy** interface has a default implementation that impacts its behavior. This component takes the order information, status and consignments status, and qualifies it to a specific **OrderCancelState**. Further order cancel process is determined by the **OrderCancelState**.

Implementing custom **OrderCancelStateMappingStrategy** allows you to define your own, preferable states mappings. It allows you to decide which specific order states can be canceled.

Order Cancel States

In the current implementation there are four distinct OrderCancelStates:

- **CANCELIMPOSSIBLE**: Order will not be canceled
- **PARTIALLYSHIPPED**: Order cancel attempt will be done by component that involves warehouse integration
- **PENDINGORHOLDINGAREA**: Order cancel will be done internally
- **SENTTOWAREHOUSE**: Order cancel attempt will be done by component that involves warehouse integration
- **SHIPPING**: Order cancel attempt will be done by component that involves warehouse integration

Check the documents in the **Related Information** section to see the state diagrams showing how the order cancel service uses the information from the OrderCancelStates.

Behavior of the Default Implementation

The table below presents possible status cases returned by mapping strategy depending on the:

- **Order status**: Listed in the OrderStatus column
- **Consignment status**: Presented in the Consignments column

Mapping Strategy Returns	OrderStatus	Consignments
CANCELIMPOSSIBLE	If order status is one of: **CANCELLED**, **CANCELLING** or **COMPLETED**	or if statuses of **all** consigmnets are set to **SHIPPED**
PENDINGORHOLDINGAREA	If order status is other than **CANCELLED**, **CANCELLING** or **COMPLETED**	**and** order was not yet split into consignments
PARTIALLYSHIPPED	If order status is other than **CANCELLED**, **CANCELLING** or **COMPLETED**	**and** there are is at least one **SHIPPED** consignment and there are consignments having status different from **SHIPPED**
SENTTOWAREHOUSE	If order status is other than **CANCELLED**, **CANCELLING** or **COMPLETED**	**and** all consignments for the order have status different shipped **SHIPPED**
SHIPPING	Not returned from the default implementation	

Related Information

- Order Cancel Complete State Diagram
- Order Cancel Partial State Diagram

Return Service

Return Service enhances the Order Management Module by allowing the customer to return ordered goods to the seller.

To accomplish this goal, the service supports the following processes:

- A call center agent or returns administrator can use the information received from the customer to find order information.
- If the order is found, then the returns administrator can decide whether the goods are accepted.
- If the goods are accepted, then a **Return Request** is created, which contains the items from the original order. The refund method is determined.
- After the refund amount/replacement has been determined, the return request must be approved. The system can automatically approve the return, or it can be manually approved (depending on business policy). Once approved, the system generates an **RMA number** (RMA = Return Merchandise Authorization).
- The received products are logged and sent for dispositioning. The products are then returned to inventory.

i Note

The Return Service requires customization. To learn how to configure the Return Service, see Return Service Configuration.

Related Information

basecommerce Extension

Return Service Business Layer

Return Service business layer consists of methods that technically support and implement business functionality provided by this service.

Return Service Business Layer Overview

There are a few scenarios that may occur during the process of returning goods from the customer to the seller. The seller must first determine if the customer is eligible to return delivered products. If the customer is eligible, then the seller may proceed with handling the product return process and offer refund or replacement to the customer. All these actions are supported by the methods implemented in the basecommerce extension. Which of the scenarios, refund or replacement, is applied in reality depends on the particular shop owner business policy.

DefaultReturnService Business Layer Methods

The table below presents the business functionality methods that together are part of **DefaultReturnService**.

Method	Description
createReturnRequest	Creates an **return request** object for the order to be returned.
getReturnRequests	Returns the **return request** for the specified order.
getRMA	Returns a **Return Merchandise Authorization** also known as **Return Material Authorization (RMA)**.
createRMA	Creates **Return Merchandise Authorization** also known as **Return Material Authorization (RMA)** and assigns it to the corresponding **ReturnRequest**.
getReplacementOrder	Returns the **ReplacementOrder** based on the corresponding **RMA value**.
createReplacementOrder	Creates a **ReplacementOrder**.
addReplacementOrderEntries	Adds **ReplacementOrderEntry** entries from the corresponding **ReplacementEntryModel**. We only accept **ReplacementEntries**, which are not on HOLD (***ReturnAction***).
createReplacement	Creates a **Replacement** based on the assigned **OrderEntry** instance.
createRefund	Creates a **Refund** based on the assigned **OrderEntry** instance.
getReturnEntries	Returns the **ReturnEntries** for the specified product.
getReturnEntry	Returns the **ReturnEntries** for the specified order entry.
getReplacements	Returns all **Replacements** for the specified return request.
isReturnable	Determines if the product is returnable by using the injected **ReturnableCheck** strategy.
processReturnEntries	Here you have the chance to inject your final Returns Entry processing. For example for handling consignment creation
processReplacementOrder	Here you have the chance to inject your final **ReplacementOrder** processing. For example for handling consignment creation.
processRefundOrder	Here you have the chance to inject your final **RefundOrder** processing. For example for handling consignment creation or initiating the final payment transaction.
getAllReturnableEntries	Returns all returnable **OrderEntries**.

Related Information

payment Extension

Return Service Configuration

Configure the Return Service for a particular implementation environment.

Spring-Based Configuration

Return Service can support:

- Returning goods to the seller
- Replacement order handling
- Refund handling
- Configurable check of 'is returnable' status

You can configure Return Service by overriding the beans in the `basecommerce-spring.xml` file.

The first part of the configuration file contains a bean where, besides existing code, you can add additional logic to enhance the Return Service. For example, you could add Consignment (Warehouse)-related business logic. For details, see:

- de.hybris.platform.returns.processor.ReturnEntryProcessor
- de.hybris.platform.returns.processor.ReplacementOrderProcessor
- de.hybris.platform.returns.processor.RefundOrderProcessor

```
<alias alias="returnService" name="defaultReturnService"/>
                <bean id="defaultReturnService" class="de.hybris.platform.returns.impl.Defaul
                <property name="generator" ref="defaultRMAGenerator"/>
                <property name="refundService" ref="refundService"/>
                <property name="modificationHandler" ref="defaultOrderReturnRecordsHandler"/>
                <property name="returnableChecks">
                <list>
                <ref bean="defaultChecksExistingReturnEntry"/>
                <ref bean="defaultConsignmentBasedReturnableCheck"/>
                </list>
                </property>
                <!--
                <property name="returnEntryProcessor" ref="YOUR_RETUNRS_ENTRY_PROCESSOR"/>
                <property name="replacementOrderProcessor" ref="YOUR_REPLACEMEMNT_ORDER_PROCE
                <property name="refundOrderProcessor" ref="YOUR_REFUND_ORDER_PROCESSOR"/>
                -->
                <property name="replacementOrderDao" ref="replacementOrderDao"/>
                <property name="returnRequestDao" ref="returnRequestDao"/>
                </bean>
```

Configuration of the service responsible for handling refunds.

```
<alias alias="refundService" name="defaultRefundService"/>
                <bean id="defaultRefundService" class="de.hybris.platform.refund.impl.DefaultRefunc
                <property name="modificationHandler" ref="defaultOrderReturnRecordsHandler"/>
                <property name="refundDao" ref="refundDao"/>
                </bean>
```

Configuration code of the Return Merchandise Authorization (RMA) generator that uses the injected 'keygenerator'.

```
<bean id="defaultRMAGenerator" class="de.hybris.platform.returns.impl.DefaultRMAGenerator">
                <property name="keyGenerator" ref="defaultRMACodeGenerator"/>
                </bean>
```

Configuration code of the keygenerator used for RMA generation.

```
<bean id="defaultRMACodeGenerator" class="de.hybris.platform.servicelayer.keygenerator.impl.Persist
                <property name="key" value="RMA"/>
                <property name="digits" value="8"/>
                <property name="start" value="00000000"/>
                <property name="numeric" value="true"/>
                </bean>
```

Configuration code of the keygenerator used for 'Return Request Code' generation.

```
<bean id="defaultReturnRequestCodeGenerator" class="de.hybris.platform.servicelayer.keygenerator.in
                <property name="key" value="AUTH"/>
                <property name="digits" value="8"/>
                <property name="start" value="00000000"/>
                <property name="numeric" value="true"/>
                </bean>
```

Configuration code of the keygenerator used for 'Replacement Order Code' generation.

```
<bean id="defaultReplacementOrderCodeGenerator" class="de.hybris.platform.servicelayer.keygenerator
                <property name="key" value="REPORD"/>
                <property name="digits" value="8"/>
                <property name="start" value="00000000"/>
                <property name="numeric" value="true"/>
                </bean>
```

Configuration code responsible for adding a 'generated' code, if there is not a defined one.

```
<bean id="PrepareReturnRequestInterceptor" class="de.hybris.platform.returns.impl.PrepareReturnRequ
                <property name="keyGenerator" ref="defaultReturnRequestCodeGenerator"/>
                </bean>
```

Configuration code responsible for adding a 'generated' code, if there is no defined one.

```
<bean id="PrepareReplacementOrderInterceptor" class="de.hybris.platform.returns.impl.PrepareReplace
                                                                <property name="keyGenerator" ref="
                                                                </bean>
```

The configuration snippet for the class used by Return service to provide information, if the specified product is returnable.

```
<bean id="defaultConsignmentBasedReturnableCheck" class="de.hybris.platform.returns.strategy.impl.D
```

This class is used by **ReturnService** to provide information, if specified product is returnable. The currently implemented algorithm determines if the product is 'returnable' on base of existing 'returns entries'.

'True' will be returned if there are no related 'returns entries' or if the quantity of the related 'returns entries' plus the specified 'return quantity' is less than or equal to to the total quantity of the ordered items so far.

```
<bean id="defaultChecksExistingReturnEntry" class="de.hybris.platform.returns.strategy.impl.Default
```

Related Information

payment Extension

Refund Service

The Refund Service enables a customer to get a refund of a certain amount, when some kind of order goods return occurs. The default implementation does not adequately account for promotion lifecycles.

With help of the Return Service, a manager or call center agent can decide whether:

- Product can be returned to the seller
- Product can be replaced by another product
- Refund can be granted to the customer

The return process ends with an order refund or partial order refund. In a simple use case, the refund amount is equal to the sum of order entry prices which are canceled or returned. However, when discounts and promotions are involved, a more sophisticated approach is required. As a result, the default refund calculation in SAP Commerce has some limitations.

Prices and Promotions Lifecycle

When an order is created, a snapshot is taken of products, prices, and promotions. From the customer's point of view, this offer is recorded in the order. In SAP Commerce, the product code, price, and discount are recorded directly in the order. From that point in time, the products on offer in your catalog can evolve (such as changing prices) without affecting existing orders. In other words, even if the product price is changed after an order is created, this will not change original product price stored in the order, and will not affect recalculation.

However, when an order contains promotions, this is not the case. Only the promotion outcome, and not the specific promotion details, are reflected in the order. If a promotion is changed, this can affect refund calculations for orders that were placed using the older version of the promotion rule.

One custom solution would be to pre-calculate each order entry by including the weight of the promotion. One could then use this value, which contains the promotion ventilated on each order entry, to get the right refund amount.

Default SAP Commerce Refund Calculation

In SAP Commerce, the default refund calculation doesn't take into account the customer group or the promotion lifecycle. The default refund strategy assumes that when a promotion is published, it will never be changed. If this is the case, then SAP Commerce should calculate an accurate refund amount.

Note that calculation is executed on demand when we receive a refund request, for example from CS Cockpit.

Advantages of the default strategy include:

- Simple algorithm
- Easily understandable for the end user

Limitations of the default strategy include:

- If a promotion is changed, the new promotion conditions are not reflected in refund amounts
- Not relevant anymore when a customer has been upgraded or downgraded regarding customer group advantage (e.g. added to a "VIP" group)

- Not relevant anymore when other criteria have been changed (period, stock value, and so on)

For technical details about the default algorithm, see Refund Service Business Layer.

Related Information

basecommerce Extension

Refund Service Business Layer

Refund Service is a simple service that can enhance Order Management process while dealing with promotions, vouchers and similar actions.

> **Note**
> Be aware that the customization is necessary while implementing custom projects. As a result you need to implement the required refund logic by yourself.
>
> Standard refund logic shipped out-of-the-box may prove unsatisfactory because it looks at standard prices only and may cause incorrect calculations. It should only be used as an example for your own customized work.

RefundService Business Methods

The scenario supported by RefundService deals with complicated situations related to promotions and vouchers that are affected by the earlier price calculation. For dealing with such situations, RefundService offers the following business methods:

Method	Description
createRefundOrderPreview	Creates a refund order, which is a clone of the original one. This new instance is used for applying the refund.
apply	Based on the assigned refund entries the order is recalculated.
getRefunds	Returns the refunds for the specified order.

RefundService Scenario

The business methods implementing RefundService realize refund scenario in the following way:

- Clone the original order the customer wants to have a refund for by using createRefundOrderPreview.
- Create Refund instances for the products for which a refund should be calculated.
- Calculate the refund amount based on the Preview Order and the Refunds by using apply(List, Order).
- Based on the calculated Preview Order the customers decide if they want to accept the offered refund.
- Eventually, in case the customer agrees to the offered refund, the call center agent can now recalculate the original order by calling apply(Order) method.

The 'preview order' is created by cloning the original order. The recalculation of the 'preview order' respects all existing promotions and vouchers of the original order. As a result no special handling of the 'promotion scenario' is needed.

Related Information

payment Extension

Deep Linking

The Deep link URL feature provides an easy way of mapping URLs to corresponding destination URLs.

Use Case

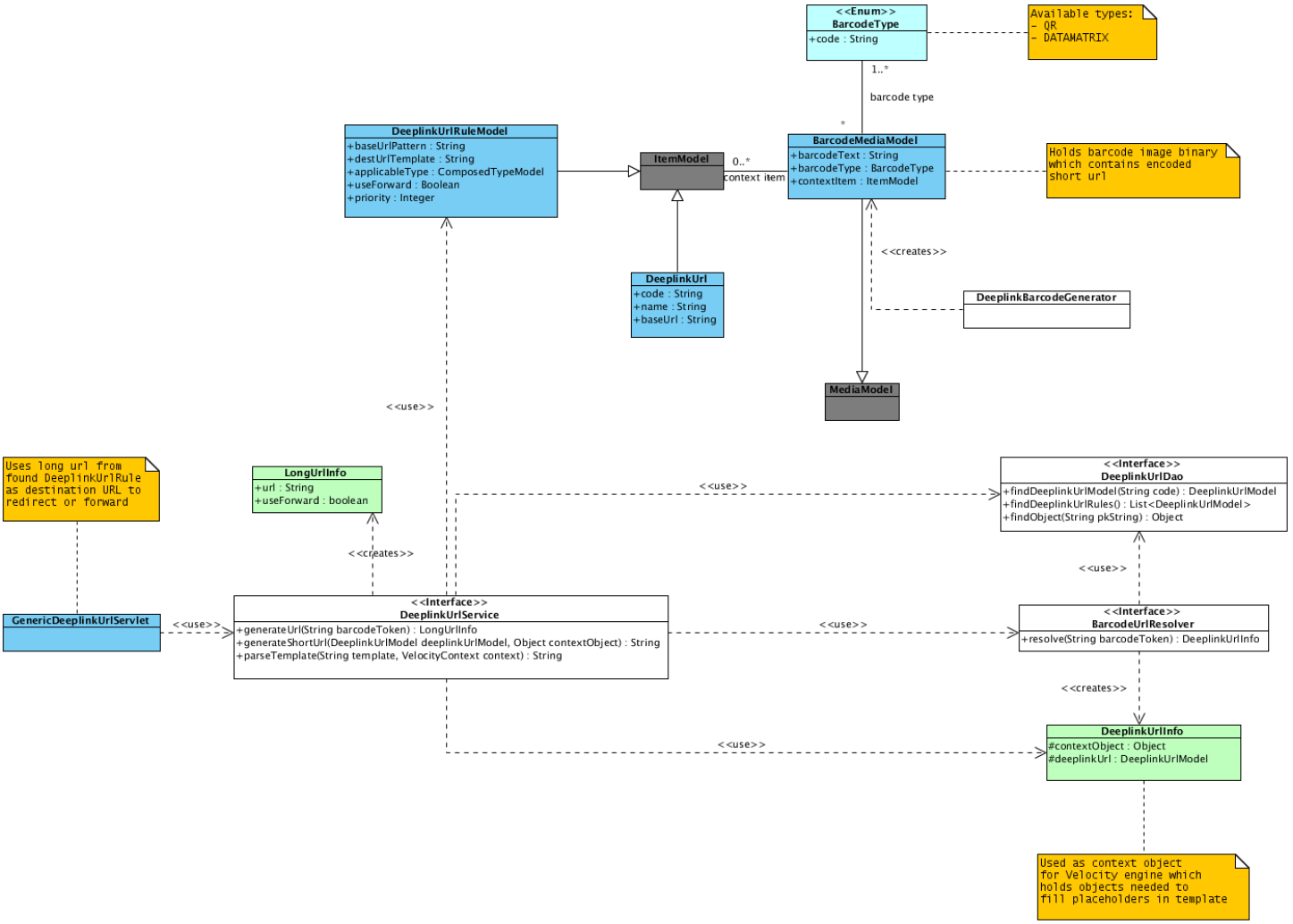
Deep linking allows you to create a short URL for some context items (for instance, products) and a long destination URL for the same product. It is based on the information extracted from a token, which is a part of the short URL with the usage of a special template provided in the `DeeplinkUrlRule` object. You can also use this short URL in product barcodes.

The `basecommerce` extension provides a generic servlet that you can use as a short URL, translate it and redirect the user to the destination URL.

Related Information

Deep Linking Architecture
Creating New Deeplink URL
Creating New Deeplink URL Rule
Generating URLs
Customizing Deeplink URLs

Deep Linking Architecture



Models

DeeplinkUrlModel

The deep link URL model is a business item containing the following attributes:

Attribute	Description
Code	Unique string identifier of the deeplink URL
Name	Localized name
BaseUrl	Base URL pointing to the deep link URL Servlet, for example, `http://www.conrad.de/mobile/link`

DeeplinkUrlRuleModel

The Deeplink URL rule is a technical mapping from the deeplink URL string to the destination URL. It contains the following attributes:

Attribute	Description
baseUrlPattern	Regular expression to match desired base URL. For example: `.*www\.conrad\.de.*`.
destUrlTemplate	Destination URL pattern. It can contain variables referencing the context item (for example product) and the deep link U item itself. For example:

Attribute	Description
	`/mobile/view/product/$ctx.contextObject.catalogVersion.catalog.id/$ctx.contextObject.c campaign=$ctx.deeplinkUrl.code&version=$ctx.contextObject.catalogVersion.version`.
applicableType	Composed type to which this entry can be applied to.
useForward	Deep link URL Servlet uses forward if this is set to true, it redirects otherwise.
priority	Priority used for searching for Deep link URL rules. The lower value set the higher priority.

Each instance of the the deeplink URL mapping represents one type of deeplink. If the deeplink has to be bound to an item (for example, a product), then the destination URL template includes a variable for that item. For example, you can get the `$ctx.item.code`, which stands for the product code. The specific item is then added (as PK) to the base URL with the deeplink URL code. See the following example for the final URL: `http://www.conrad.de/mobile/view/product/clothescatalog/30124?campaign=mobile&version=Online`.

Services

DeeplinkUrlService

The following lists the actions that the DeeplinkUrlService can do:

- Generates the short URL from the deeplink URL object and the context item. This URL can be later used in the barcode.
- Generates the long destination URL extracted from the token that is a part of the short URL. It uses the deeplink URL rules internally.
- Parses through the velocity engine, which provides the templates.

GenericDeeplinkUrlServlet

The Deeplink URL Servlet is used for getting the short URL (for instance obtained from barcodes by mobile device), transforming it, and next redirecting or forwarding to the destination URL.

BarcodeUrlDao

The DAO object is used by the DeeplinkUrlService for finding and getting DeeplinkUrlModel, DeeplinkUrlRule objects as well as context objects (for instance, products).

BarcodeUrlResolver

BarcodeUrlResolver is used for resolving the short URL tokens during the generation of the long URL (method DeeplinkUrlService#generateUrl(final String barcodeToken)).

DeeplinkUrlInfo

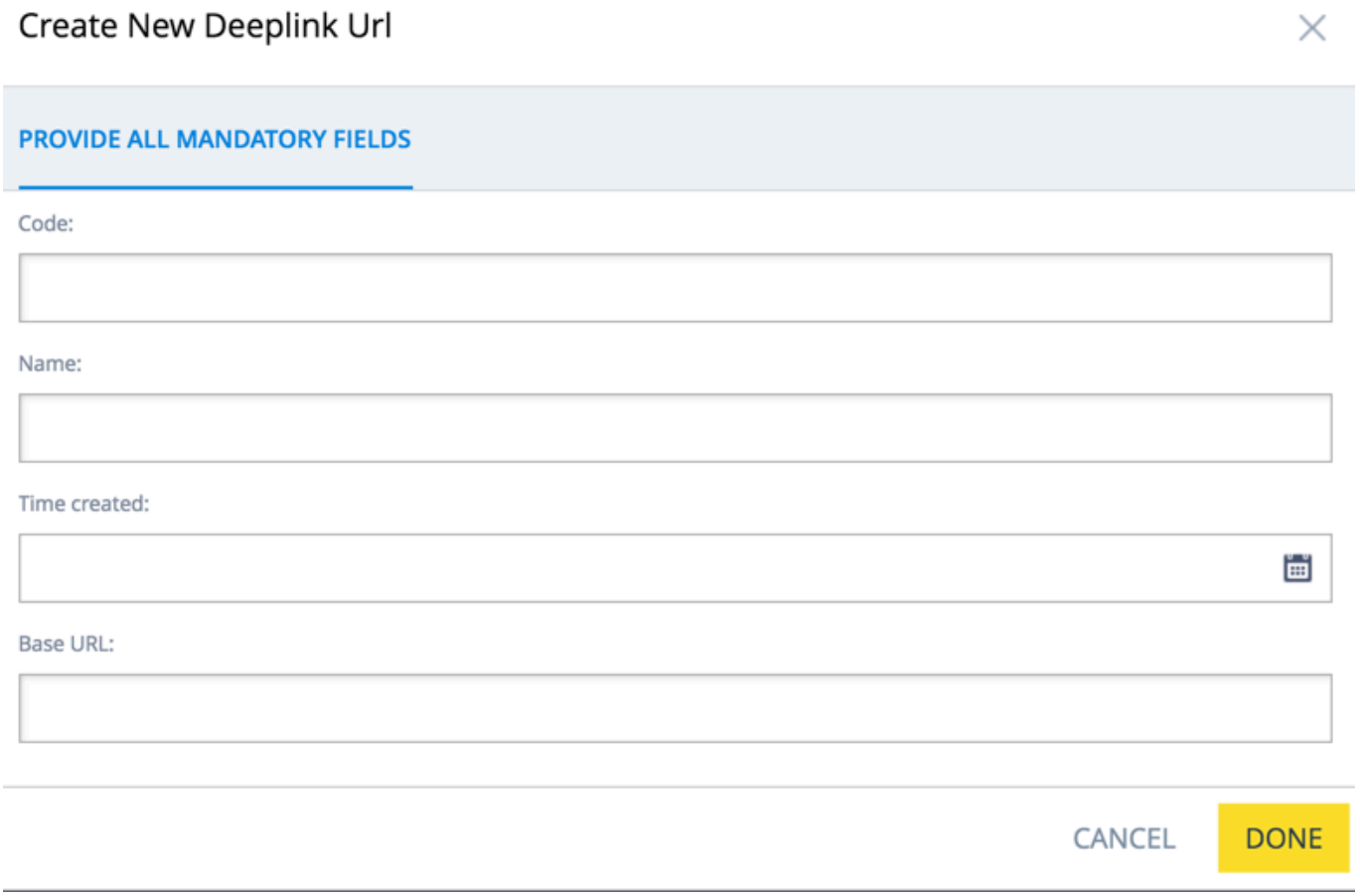
The DeeplinkUrlInfo is a data object holding the necesary data for the velocity engine.

Creating New Deeplink URL

You can create a deeplink URL through the Backoffice Administration Cockpit.

Procedure

1. Log on to the Backoffice Administration Cockpit.
2. Navigate to **Deeplink Urls** > **Deeplink Url**.
3. Click the **Add** button to display the Create New Deeplink Url window.



4. Enter the following details in the mandatory fields:
   - **Code**
   - **Name**
   - **Base URL**

   The system automatically populates the **Time Created** field with the information on when the deeplink URL is created.

   Note

   You can use the BaseURL with a specific context object (for instance, a product) for barcode generation.

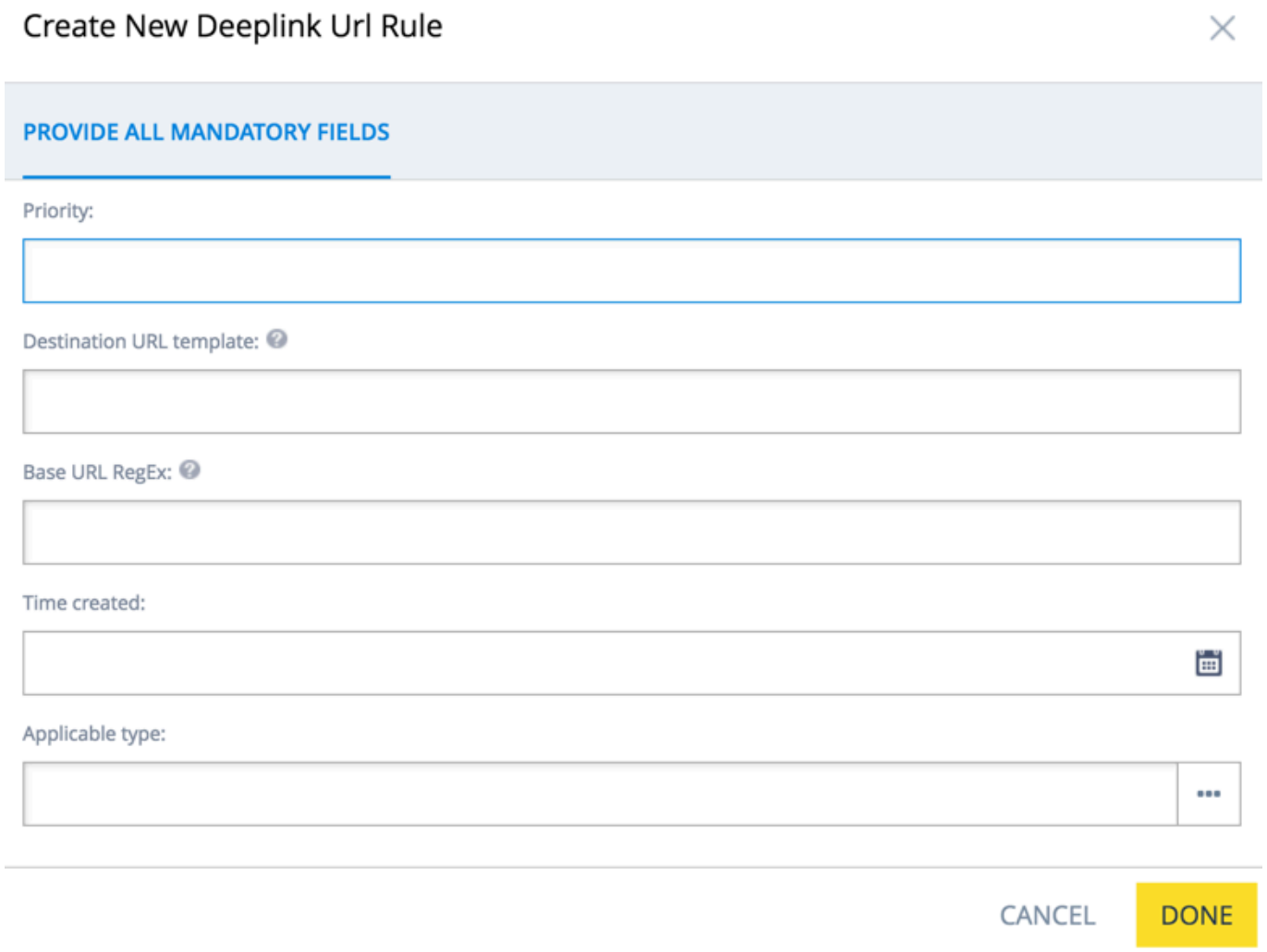
5. Click **Done** to create the deeplink URL. Otherwise, click **Cancel**.

Creating New Deeplink URL Rule

You can create a new deeplink URL Rule through the Backoffice Administration Cockpit.

Procedure

1. Log on to the Backoffice Administration Cockpit.
2. Navigate to **Deeplink Urls** > **Deeplink Url Rule**.
3. Click the **Add** button to display the Create New Deeplink Url window.



4. Enter the following details in the mandatory fields:
    - **Priority**
    - **Destination URL template**
    - **Base URL RegEx**
    - **Applicable type**

    The system automatically populates the **Time Created** field with the information on when the deeplink URL is created.

    Note

    By default, the `DeeplinkURLService` uses the velocity template engine to parse the Destination URL template. You can possibly change the parser in your own service. For more information, refer to Customizing Deeplink URLs.

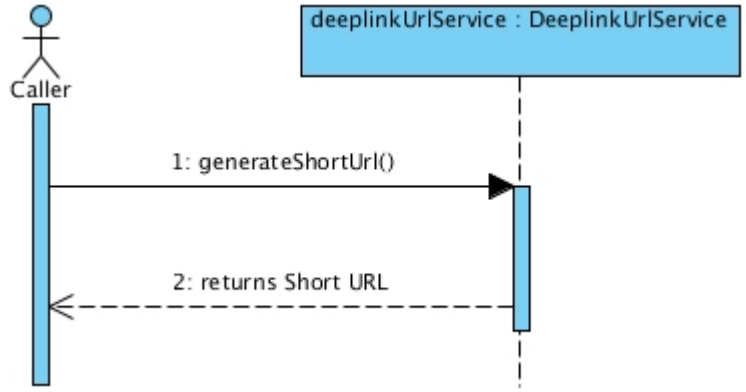
5. Click **Done** to continue action. Otherwise, click **Cancel**.

Generating URLs

You can generate short and long URLs and write a template for the URL.

Generating Short URLs

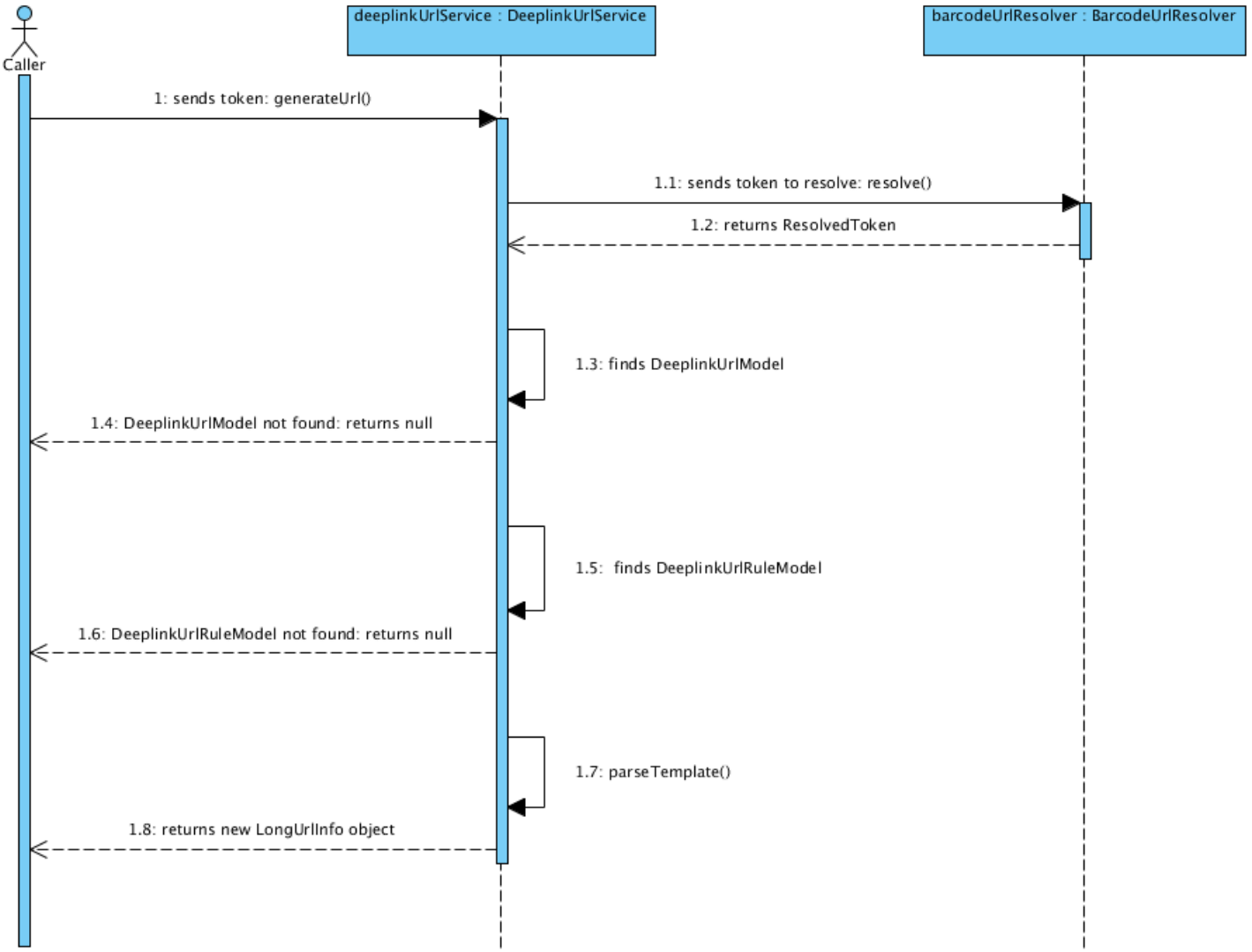
The following diagram presents the process of generating a short URL from the `DeeplinkUrlModel` object and any object that is considered as context object:



If the context object is an instance of the `ItemModel`, then its PK is used in the short URL.

Generating Long URLs

The following diagram presents the process of generating a long URL from passed tokens:



Writing a Template for the URL

Write all templates following the velocity standard where placeholders are preceded by the $ sign. The *<ctx>* variable in the template can access an object passed to the velocity engine in the `DeeplinkUrlService`. See the following example of a template:

```
/mobile/product/$ctx.contextObject.code?campaign=$ctx.deeplinkUrl.code&catalog=$ctx.contextObject.c
```

Customizing Deeplink URLs

All services and helper classes are controlled through Spring configuration.

The following table lists the bean IDs that are used in `basecommerce-spring.xml`:

Bean ID	Interface	Default Implementation
`deeplinUrlDao`	`de.hybris.platform.deeplink.dao.DeeplinkUrlDao`	`de.hybris.platform.de`
`barcodeUrlResolver`	`de.hybris.platform.deeplink.resolvers.BarcodeUrlResolver`	`de.hybris.platform.de`
`deeplinkUrlService`	`de.hybris.platform.deeplink.services.DeeplinkUrlService`	`de.hybris.platform.de`

Changing Default Token Parser

You may want to parse different templates than the default ones that allow you to only use the context object and the Deeplink URL object. To do so, change the parser class to another one, which returns a different data object. The velocity engine uses the data object as a context object during the template parsing. The following shows an example of the implementation:

```
public class FooBarResolver extends DefaultBarcodeUrlResolver
{
  @Override
  public ResolvedToken resolve(final String token)
  {
    // Split token
    final String[] splitted = token.split(":");
    // get Deeplink Url object
    final DeeplinkUrlModel deeplinkModel = getDeeplinkUrlDao().findDeeplinkUrlModel(splitted[0]);
    // Get context object (for instance product)
    final Object contextObject = getDeeplinkUrlDao().findObject(splitted[1]);

    // Return ctx object for Velocity
    return new FooBarUrlInfo(deeplinkModel, contextObject, splitted[2], splitted[3]);
  }
}
```

To give you a better idea, the following example displays how the code is cleared from any required null checks. Check `FooBarUrlInfo` in the following example:

```
public class FooBarUrlInfo extends DeeplinkUrlInfo
{
  private String someInfo;
  private String anotherOne;

  public FooBarUrlInfo(final DeeplinkUrlModel deeplinkUrl, final Object contextObject, final String
  {
    super(deeplinkUrl, contextObject);
    this.someInfo = someInfo;
    this.anotherOne = anotherOne;
  }

  public String getSomeInfo()
  {
    return someInfo;
  }

  public String getAnotherOne()
  {
    return anotherOne;
  }
}
```

The following shows you the Spring configuration:

```
<bean id="barcodeUrlResolver" class="de.hybris.platform.deeplink.resolvers.impl.FooBarResolver">
    <property name="deeplinkUrlDao" ref="deeplinUrlDao" />
  </bean>
```

In the template, you can use any getters from the `FooBarUrlInfo` data object, which is passed to the velocity engine:

```
/foo/bar/$ctx.contextObject/$ctx.deeplinkUrl/$ctx.someInfo/$ctx.anotherOne
```

Because the `FooBarUrlInfo#getDeeplinkUrl` (available through inheritance from `DeeplinkUrlInfo` class) method returns the `DeeplinkUrlModel` object, the placeholder, such as `$ctx.deeplinkUrl.code`, works. The `DeeplinkUrlModel` contains the attribute code and the appropriate getter, so the sample template can be expanded to the following Java code:

```
ctx.getDeeplinkUrl().getCode();
```

Assume that from `FooBarUrlInfo#getContextObject`, you can get the `ProductModel`, then this template also works: $ctx.contextObject.catalogVersion.catalog.id and is expanded to the following Java code:

```
ctx.getContextObject().getCatalogVersion().getCatalog().getId();
```

Fraud Detection

Fraud Detection provides extensible services for performing fraud checks internally as well as interfaces to integrate external fraud detection providers.

Use Case

Commerce systems have to cope with fraudulent orders. Fraud detection offers both a standard implementation and a common interface for integration of fraud detection services. Technically, it is implemented with the `basecommerce` extension.

Features

Symptom-based Fraud Detection

SAP Commerce includes a **standard provider** implementation, which checks for a number of fraud symptoms. You can customize how each symptom affects the fraud score.

Note that this standard provider should only be used as a first-level check. Used in combination with a third-party service provider,however, it may help to reduce the number of paid requests to the third-party provider.

Integrate Third-Party Service Providers

Integrate third-party fraud detection services with SAP Commerce to protect your website from fraudulent orders.

Extensibility

Extend the standard implementation to add a new symptom check and add it to the standard service configuration.

Related Information

yacceleratorfulfilmentprocess Extension

The SAP Commerce processengine

Fraud Detection Implementation

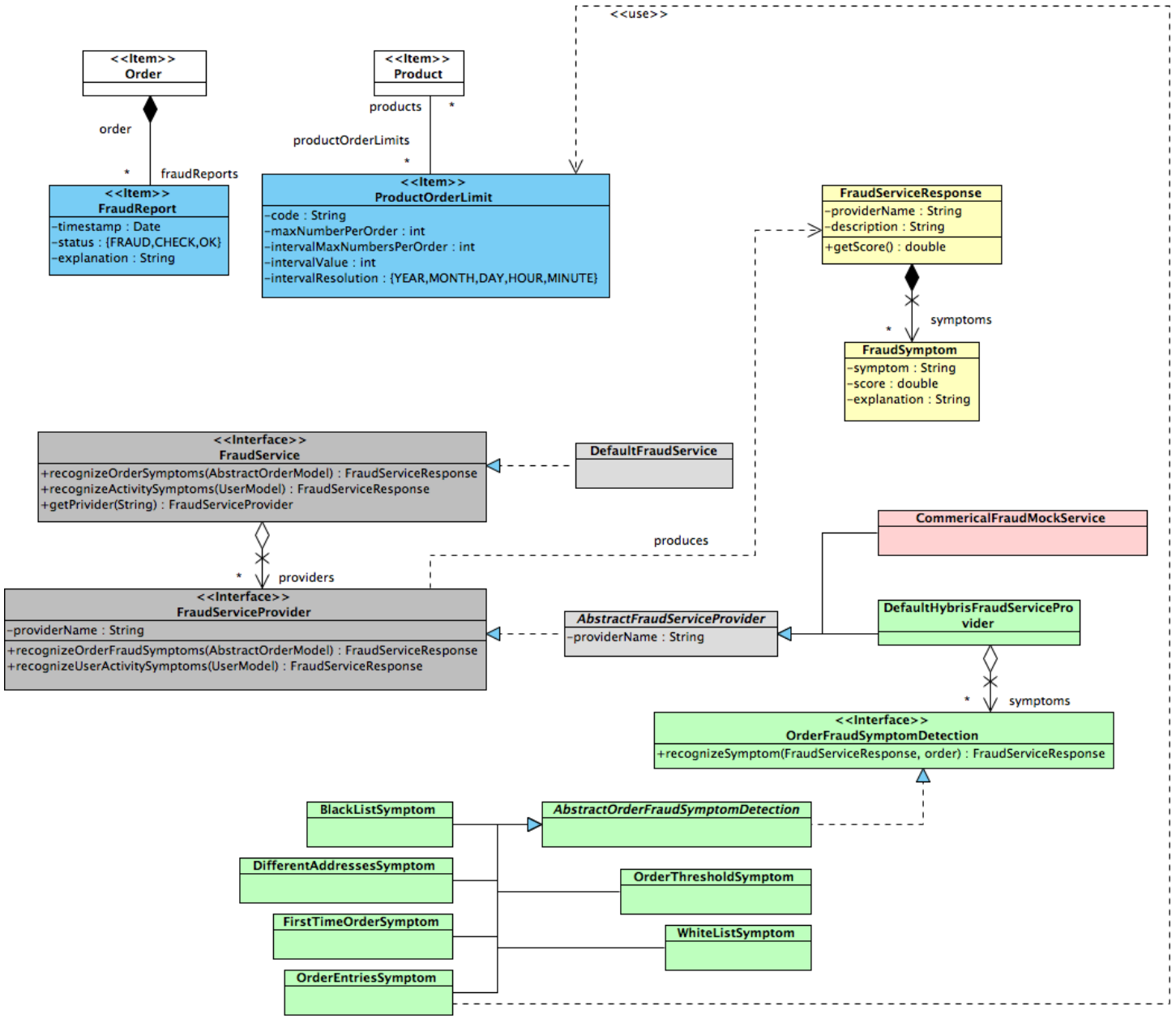
The fraud detection model offers a common interface for integration of arbitrary fraud detection services, as well as a standard integration included with SAP Commerce.

SAP Commerce includes a **standard provider** that checks for a number of fraud symptoms without the need for an external system. However, this standard service should be used in combination with a third-party service provider as a first-level check in order to reduce the number of paid requests to the third party provider. You can extend this service by adding custom symptoms.

Implementation

The following diagram shows the **fraud service** and all related items, result objects and included provider implementations.

The central service used for business logic is **FraudService**, which provides access to all available **FraudServiceProvider** services.



Perform Fraud Checking

This section illustrates how to perform fraud checking, how the obtained results should be handled, and how fraud detection status can be persisted. It also explains how the standard SAP Commerce fraud service provider works and how to extend it.

The fraud detection model functions by testing for several symptoms. These symptoms are usually weighted differently according to their importance. Therefore the result of a fraud check produces a list of tested, respectively positive symptoms, each of which has a numerical **score** showing its importance. The overall score is the sum of all these symptom values.

ℹ Note

The result of the fraud check is a numerical score, not a positive or negative decision regarding whether an order is fraudulent. The result thus only serves as the basis on which a decision can be made. The service caller usually interprets the overall score to determine whether the order is a fraud.

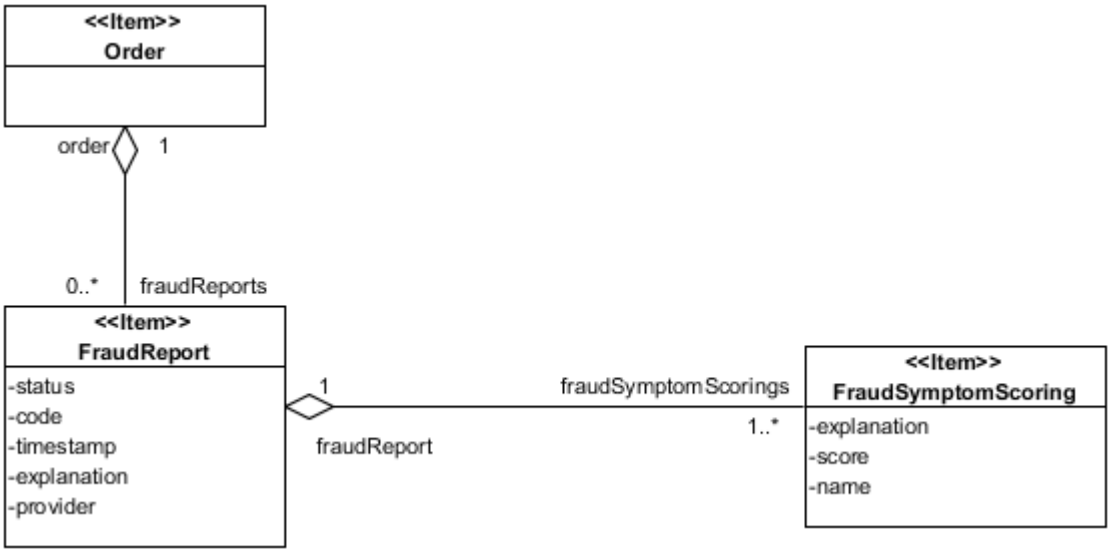
```
FraudService fraudService = ...
Order placedOrder = ...

// using standard provider named 'Hybris'
FraudServiceResponse result = fraudService.recognizeOrderFraudSymptoms("Hybris",placedOrder);

// this is the overall score
double score = result.getScore();

// we're also able to check all result symptoms
for( FraudSymptom symptom : result.getSymptoms() )
{
   double singleScore = symptom.getScore();
   String symptomCode = symptom.getSymptom();
   String explanationText = symptom.getExplanation();
}
```

The **FraudServiceResponse** is a plain Java object. The service caller is responsible to store its information elsewhere.

Storing Fraud Reports

Once a fraud check result was obtained and the caller decided whether or not the order is fraudulent, this information should be persisted for retrieval in further order processing.

The item type **FraudReport** is designed to store fraud check results for a particular order.



It enables to store the overall score result and evaluation statuses of particular symptoms configured in the fraud service.

The following snippet shows how to store the fraud evaluation response data in the default model:

```
ModelService modelService = ...

        Order placedOrder = ...
        FraudServiceResponse result = ...

        final FraudReportModel fraudReport = modelService.create(FraudReportModel.class);
        fraudReport.setOrder(placedOrder);
        fraudReport.setStatus(status);
        fraudReport.setProvider(providerName);
        fraudReport.setTimestamp(new Date());
        fraudReport.setCode(" .. "); //your own report code generation
        List<FraudSymptomScoringModel> symptoms = new ArrayList<FraudSymptomScoringModel>();
        // store info for each symptom
        for (final FraudSymptom symptom : result.getSymptoms())
        {
                final FraudSymptomScoringModel symptomScoring = modelService.create(FraudSymptomSc
                symptomScoring.setFraudReport(fraudReport);
                symptomScoring.setName(symptom.getSymptom());
                symptomScoring.setExplanation(symptom.getExplanation());
                symptomScoring.setScore(symptom.getScore());
                symptoms.add(symptomScoring);
        }
        fraudReport.setFraudSymptomScorings(symptoms);

        modelService.save(fraudReport)
```

Standard Fraud Detection Provider

The fraud functionality contains a standard fraud detection service that does not rely on third-party service providers. This service should therefore be seen as a first-level check, which should be supplemented with an integrated third-party service.

Each supported symptom is implemented as **OrderFraudSymptomDetection** and can be added or removed via the spring configuration. This approach also enables you to add new symptom check implementations when required.

Supported Symptoms

Currently the following sample implementations are available:

Class Name de.hybris.platform.fraud.symptom.impl	Description
BlackListSymptom	Sample - based on a static email/userId list.
WhiteListSymptom	Sample - based on a static email/userId list.
FirstTimeOrderSymptom	Checks if user is a first timer.
DifferentAddressesSymptom	Sample - checks if address rules apply.
OrderThresholdSymptom	Sample - checks if an order value threshold was reached.
OrderEntriesSymptom	Sample - checks if product order limit applies (in the mockup: maxNumberPerOrder only).

The actual limit per product is defined in **ProductOrderLimit** and is responsible to specify limit settings for one or more products.

Specify the **ProductOrderLimit** attributes to define a product's ordering limitations:

Attribute Name	Type	Description
code	String	Human recognizable identifier of the limit.
maxNumberPerOrder	Integer	Maximum allowed amount of a given product in the cart. Used by the **OrderEntriesSymptom** detection.
intervalValue	Integer	Quantifier of interval definition.
intervalResolution	Enumeration of time units (DAY, WEEK, MONTH,..etc.)	unit of interval definition. Together with **intervalValue** fully describes the time interval, for example (2,DAY), (1, WEEK).
intervalMaxOrdersNumber	Integer	Maximum allowed amount of given orders in the defined time interval.

How To Extend

To extend, you can implement a new symptom check and add it to the standard service configuration.

```
public class MySymptom implements OrderFraudSymptomDetection
{
   public FraudServiceResponse recognizeSymptom(FraudServiceResponse fraudResponse, AbstractOrderMo
   {
      if (... positive ... )
      {
         fraudResponse.addSymptom(
            new FraudSymptom(
               "...explanation...",
               100d, // score
               "MySymptom" // symptom code
            )
         );
      }
      else // optionally we may even add a symptom to result if test was negative
      {
         fraudResponse.addSymptom( new FraudSymptom( 0d,"MySymptom") );
      }
      return fraudResponse;
   }
}
```

The Spring configuration should look like this:

```
<bean id="mySymptom" class="MySymptom">

<!-- replace definition of default hybris fraud provider -->
<bean id="defaultHybrisFraudServiceProvider" class="de.hybris.platform.fraud.impl.DefaultHybrisFrau
   <property name="providerName" value="Hybris"/>
   <property name="symptomList">
      <list>
         <ref bean="blackListSymptom"/>
         ...
         <ref bean="mySymptom"/>
      </list>
   </property>
</bean>
```

Defining the Importance of Fraud Symptoms

You may want to recognize some of the fraud symptoms as very important (for example a user from a black list) or less important. To do so, you can set the symptom fraud score increment in the spring configuration. This value, when the given symptom occurs, influences the order overall fraud evaluation.

All symptoms extend from the **AbstractOrderFraudSymptomDetection** class:

```
public abstract class AbstractOrderFraudSymptomDetection implements OrderFraudSymptomDetection
{

	public static final double DEFAULT_INCREMENT = 50;

	private double increment = DEFAULT_INCREMENT;
	private String symptomName;

	protected FraudSymptom createSymptom(final boolean positive)
	{
		return createSymptom(null, positive);
	}

	protected FraudSymptom createSymptom(final String explanation, final boolean positive)
	{
		return new FraudSymptom(explanation, positive ? getIncrement() : 0, getSymptomName(
	}

	public String getStrategyName()
	{
		return getSymptomName() + STRATEGY_SUFFIX;
	}

	public String getSymptomName()
	{
		return symptomName;
	}

	@Required
	public void setSymptomName(final String name)
	{
		this.symptomName = name;
	}

	/**
	 * @return the value to be used as score in case the symptom has been tested positively.
	 */
	public double getIncrement()
	{
		return increment;
	}

	public void setIncrement(final double increment)
	{
		this.increment = increment;
	}
}
```

If you do not specify any score increment for your particular symptom, the default increment (+50) is taken from the abstract class.

As an example, the **WhiteListSympton** redefines the increment to -500, in order to favor orders from trusted users:

```
<bean id="whiteListSymptom" class="de.hybris.platform.fraud.symptom.impl.WhiteListSymptom">
		<property name="symptomName" value="White list" />
		<property name="increment" value="-500" />
		<property name="favoredEmails">
			<list>
				<value>goldMember@gmail.com</value>
				<value>silverMember@gmail.com</value>
			</list>
		</property>
		<property name="favoredUserIDs">
```

```
                        <list>
                                <value>goldMember</value>
                                <value>silverMember</value>
                        </list>
                </property>
        </bean>
```

Integrating an External Fraud Detection Provider

Integrating an external provider requires an implementation of **FraudServiceProvider**. Its responsibility is to produce **FraudServiceResponse** objects. There are no further constraints.

A response can be produced according to order criteria, such as suspicious addresses, cart content, or according to user activity criteria like frequently changed registration data. The former is offered by the **standard provider** behind the **recognizeOrderFraudSymptoms** method, but the latter is designed for future development purposes. In the existing implementation, it is not implemented, hence the **recognizeUserActivitySymptoms** method should not be called.

Response objects may contain symptom information or simply the overall score. Providing explanation texts is also optional.

```
public class MyDummyProvider implements FraudServiceProvider
{
   public String getProviderName()
   {
      return "Dummy";
   }

   public FraudServiceResponse recognizeOrderFraudSymptoms(AbstractOrderModel order)
   {
      return new FraudServiceResponse(getProviderName())
      {
         public double getScore()
         {
            return 0;
         }
      }
   }

   public FraudServiceResponse recognizeUserActivitySymptoms(UserModel user)
   {
      throw new RuntimeException("no implemented");
   }
}
```

Then replace the fraud service definition to add this provider.

```
<bean id="myDummyProvider" class="MyDummyProvider">

<bean id="defaultFraudService" class="de.hybris.platform.fraud.impl.DefaultFraudService">
   <property name="providers">
     <list>
        <ref bean="internalFraudServiceProvider"/>
        <ref bean="commercialFraudServiceProvider"/>
        <ref bean="myDummyProvider"/>
     </list>
   </property>
</bean>
```

Fraud Detection as Part of Fulfillment Process

The fraud service can be used in the order fulfillment process, where the order evaluation can take place in the process node that is executed before the order is sent for fulfillment.

Check the related example process of the order management. Note that the **fraudCheck** node either redirects the process to manual order check by the customer service if there is fraud risk, and otherwise continues with normal fulfillment.

The **fraudCheck** node should be implemented in a node action that extends from the **AbstractSimpleOrderDecisionAction** as the output decision follows from fraud evaluation: FRAUD, or NO FRAUD (which refers to the natural decision-action transitions: OK, NOK).

Here is a sample fraud check node as a part of fulfillment process:

```
public class MyFraudCheckAction extends AbstractSimpleOrderDecisionAction
{

        private FraudService fraudService;

        private String providerName;

        private int scoreLimit;

        @Override
        public Transition executeAction(final OrderProcessModel process)
        {

                final OrderModel order = process.getOrder();
                final FraudServiceResponse response = fraudService.recognizeOrderSymptoms(providerN

                final double score = response.getScore();
                if (score < scoreLimit)
                {
                        final FraudReportModel fraudReport = createFraudReport(providerName, respor
                        order.setFraudulent(Boolean.FALSE);
                        order.setPotentiallyFraudulent(Boolean.FALSE);
                        order.setStatus(OrderStatus.FRAUD_CHECKED);
                        modelService.save(fraudReport);
                        modelService.save(order);
                        return Transition.OK;
                }
                else
                {
                        final FraudReportModel fraudReport = createFraudReport(providerName, respor
                        order.setFraudulent(Boolean.TRUE);
                        order.setPotentiallyFraudulent(Boolean.FALSE);
                        order.setStatus(OrderStatus.FRAUD_CHECKED);
                        modelService.save(fraudReport);
                        modelService.save(order);
                        return Transition.NOK;
                }
        }

        private FraudReportModel createFraudReport(final String providerName, final FraudServiceRes
                        final FraudStatus satus)
        {
                final FraudReportModel report;
                report = ...
                //save the fraud evaluation report in data base for later use.
                return report;
        }

        @Required
        public void setFraudService(final FraudService fraudService)
        {
                this.fraudService = fraudService;
        }

        public void setProviderName(final String providerName)
        {
                this.providerName = providerName;
        }

        public void setScoreLimit(final int scoreLimit)
        {
                this.scoreLimit = scoreLimit;
```

```
        }

}
```

```
<bean id="fraudCheckOrderInternal" class="de.hybris.platform.fulfilment.actions.MyFraudCheckAction'
                <property name="fraudService" ref="fraudService"/>
                <property name="providerName" value="hybris"/>
        </bean>
```

Such a sample fraud checking node should make use of the **fraudService** to evaluate a processed order and use simple comparison with a defined threshold value in order to distinguish fraudulent from non-fraudulent orders. Besides setting proper flags on the order: **order.FRAUDULENT**, **order.STATUS**, the node should make proper transition to your next process nodes.

In the example above, the threshold value is injected via Spring. You may want to provide the threshold value differently, in a way that best suits your business scenarios.

The sample node does not show how to save the fraud check reports. Such reports may be used later to manually evaluate orders later. You can use the default, out-of-the-box data model to save fraud reports. Have in mind that in your business scenario, you can save the fraud evaluation results differently, for example in files.

Replenishment and Order Scheduling

Replenishment and Order Scheduling are technically realized by the B2B Commerce Module and the `basecommerce` extension. Functionally, these features are a part of the Order Management Module and are designed mainly for the B2B market. It provides several possible ways to schedule the creation of new orders or recurring orders. For more information about order replenishment, see B2B Orders.

Use Case

Replenishment and Order Scheduling is a functionality of the Order Management Module. Its main purpose is to enable customers (on-line, call-centers, mobile) to define schedules for future orders or for recurring orders.

Features

Order Scheduling

Order scheduling allows customers to add products to the shopping cart, then schedule a later date when the order should be executed. On the scheduled date, the order is executed automatically - without any additional customer involvement.

Example: On Monday, a customer sets up a grocery delivery order, then schedules the order for Friday. Order execution and fulfillment only take place on Friday.

- The customer can set up a schedule for an existing order or basket.
- The customer can decide whether the order is executed only on one schedule instance, or on a recurring schedule.
- The customer can use an existing order or the current shopping cart as a template for recurring orders.

Replenishment

Replenishment allows customers to execute orders on a regularly scheduled basis.

Example: A customer creates a recurring order with an online grocery store to have the same groceries delivered every Friday at 5:00.

About Replenishment and Order Scheduling

Order Scheduling and Replenishment complements the functionality of the Order Management module, but is technically a part of the `basecommerce` extension.

Scheduled Orders

Scheduled orders can be created from carts or based on the previously placed orders.

Orders from Template (Previous Order)

For creating an order based on a template, you can just reuse the previously placed order that serves now as a template.

```
public OrderModel createOrderFromOrderTemplate(final OrderModel template)
                                        {
                                        final OrderModel order = modelService.clone(template);
                                        modelService.save(order);
                                        runScheduledOrder(order);

                                        return order;
                                        }
```

Orders from Cart

For creating an order from the cart, you need to provide several arguments such as delivery address, payment address, and payment information.

```
public OrderModel createOrderFromCart(final CartModel cart, final AddressModel deliveryAddress,
                                        final AddressModel paymentAddress,
                                        final PaymentInfoModel paymentInfo) throws InvalidCartExcep
                                        ...
```

If the cart is empty, then no order is created.

```
{
                                        if (cart.getEntries().isEmpty())
                                        {
                                        return null;
                                        }
```

Running Scheduled Order

Both methods (orders from templates and orders from cart) return the OrderModel object that is used to run scheduled orders.

```
public OrderModel runScheduledOrder(final OrderModel order)
                                        {
                                        orderService.calculateOrder(order);
                                        modelService.save(order);

                                        runOrder(order);

                                        return order;
                                        }
```

Replenishing and Scheduling with Cron Jobs

The scheduling of orders is technically realized with cron jobs. **ScheduleOrderService.java** class provides the interface for creating specific cron jobs responsible for all implemented types of order scheduling.

CronJob Creation Service

Replenishment and Order Scheduling is realized in the form of cron job services. The **ScheduleOrderService.java** interface defines several methods responsible for creating particular a cron job responsible for order scheduling.

```
public interface ScheduleOrderService
                                {

                                /**
                                * Creates the order from order template cron job.
                                *
                                * @param template
                                *           the template
                                * @param trigger
                                *           the trigger
                                * @return OrderTemplateToOrderCronJobModel
                                */
                                OrderTemplateToOrderCronJobModel createOrderFromOrderTemplateCronJ

                                /**
                                * Creates the order from cart cron job.
                                *
                                * @param cart
                                *           the cart
                                * @param trigger
                                *           the trigger
                                * @return CartToOrderCronJobModel
                                */
                                CartToOrderCronJobModel createOrderFromCartCronJob(final CartModel
                                final AddressModel paymentAddress,
                                final PaymentInfoModel paymentInfo, List<TriggerModel> trigger);

                                /**
                                * Creates the scheduled order cron job.
                                *
                                * @param order
                                *           the order
                                * @param trigger
                                *           the trigger
                                * @return OrderScheduleCronJobModel
                                */
                                OrderScheduleCronJobModel createScheduledOrderCronJob(OrderModel or
                                }
```

CronJob Service Methods

The **ScheduleOrderService** interface defines methods for creating cron jobs responsible for specific functions. Provided trigger fires the cron job and the service method **ScheduledOrderJob.perform** is performed. The action performed by each function is dependent on the type of the cron job. Method checks what type of the cron job has been triggered and following the check results performs appropriate action and returns information about the cron job result and cron job status.

Method	Description
`createOrderFromOrderTemplateCronJob`	This method creates an order cron job based on existing order cron job template. Order created this way contains the same products as the template.
`createOrderFromCartCronJob`	This method creates cron job for the order that uses the cart data. It takes the cart item and if the cart is not empty, it automatically creates new order.

Method	Description
`createScheduledOrderCronJob`	This method recalculculates the order and creates scheduled order cron job.

Order History and Order Versioning

Order History and Order Versioning provide a simple way of storing information about order-related actions. Order Versioning makes persistent snapshots of an order, while Order History tracks changes made to the order itself.

Use Case

An order placed in SAP Commerce may proceed through a number of processing steps, each one adding information or changing the order itself. Order History and Order Versioning allow you to track these changes.

Features

Order History

Order history tracks these actions - for example, to allow customer service agents to see what happened to a particular order. See Order History.

Order Versioning

Order versioning makes persistent snapshots of an order to make it possible to track different versions of an order. See Order Versioning.

Related Information

The SAP Commerce processengine

basecommerce Extension

Order History

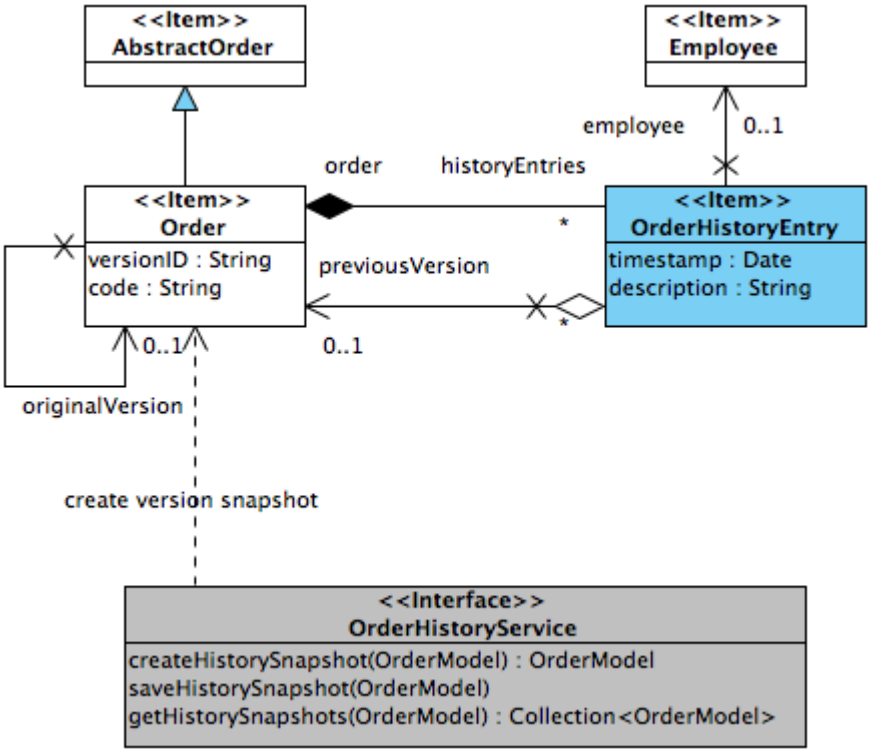
An order placed in SAP Commerce may proceed through a number of processing steps, each one adding information or changing the order itself. Order history tracks these actions - for example, to allow customer service agents to see what happened to a particular order.

Creating order history information consists of two steps:

- Creating an `OrderHistoryEntry` item model.
- Creating and attaching a snapshot (if an order state snapshot is required).

Managing Order History

The item type `OrderHistoryEntry` stores historical information about each order processing action. It does not create a new version of the original order. These entries belong to the original order; if the original order is removed, the entries are removed as well.



Each entry contains the following:

- Description
- Timestamp
- Reference to an employee (optional)
- Reference to an order state snapshot

This example uses the session user to create an `employee` reference.

```
ModelService modelService = ...
UserService userService = ...
OrderHistoryService historyService = ...

OrderModel processedOrder = ...

OrderHistoryEntryModel entry = modelService.create(OrderHistoryEntryModel.class);
entry.setTimestamp(new Date());
entry.setOrder(processedOrder);
entry.setDescription("fraud check manually passed");
entry.setEmployee( (EmployeeModel)userService.getCurrentUser() );

modelService.save(entry);
```

Fetch History Records for an Order

It is possible to fetch all history records for an original order, as in this example:

```
OrderModel processedOrder = ...

for( OrderHistoryEntryModel e : processedOrder.getHistoryEntries() )
{
   // ...
}
```

The `OrderHistoryService` also offers a range of fetching methods, which allow filtering of entries by date range or assigned employee:

```
OrderHistoryService historyService = ...
UserService userService = ...

OrderModel processedOrder = ...
Date from = ...
Date to = ...

Collection<OrderHistoryEntryModel> dateEntries =
   historyService.getHistoryEntries(processedOrder, from, to);

EmployeeModel user = (EmployeeModel)userService.getCurrentUser();

Collection<OrderHistoryEntryModel> userEntries =
   historyService.getHistoryEntries(processedOrder, user );
```

Order Versioning

Some order processing steps may change the original order. Order Versioning creates a snapshot of the order before the order is changed. The changed order becomes a new version of the order.

Order Versioning creates a deep copy of each order before it is changed. The service clones the entire order and marks the clone as a new version of the original. Order entries, addresses, and payment information are clones as well, since this information is part of the original order.

```
ModelService modelService = ...
OrderHistoryService historyService = ...

OrderModel processedOrder = ...

// create snapshot - not persisted yet !
OrderModel snapshot = historyService.createHistorySnapshot(processedOrder);

// make order adjustments
processedOrder.setDeliveryAddress(null);

// create history entry
OrderHistoryEntryModel entry = modelService.create(OrderHistoryEntryModel.class);
entry.setTimestamp(new Date());
entry.setOrder(processedOrder);
entry.setDescription("removed delivery address");
entry.setPreviousOrderVersion(snapshot);

// persist snapshot manually - this is necessary due to historical reasons
historyService.saveHistorySnapshot(snapshot);
// persist all other models as usual
modelService.saveAll( processedOrder, entry );
```

ℹ Note

Please note that after creating an order state snapshot, the model is not yet persisted. Changes can be made to it. Due to the historical behavior of the Jalo layer, it is necessary to persist the snapshot model separately by calling `OrderHistoryService.saveHistorySnapshot(..)` before actually persisting the history entry.

Since order state snapshots are generally items of the same type as their original versions, they reside in the same database table. Therefore, running queries upon any type of order now may require limiting the result to non-versioned orders.

```
SELECT {PK} FROM {Order} WHERE ...condition... AND {versionID} IS NULL
```

Since all versioned orders hold a version ID, filtering by `{versionID} IS NULL` returns only original orders.

Order Splitting

Order Splitting allows orders to be broken down into several consignments and warehouse interfaces, which permits partial shipments and effective warehousing. Order Splitting is a part of the `basecommerce` extension.

Use Case

Order Splitting offers a number of services to realize the creation of consignments out of one order and enable the proper warehouse choice, depending on stock levels. Although a consignment may contain all the order entries of an order, consignments are typically created for orders that need to be broken down into two or more parts because they contain products from different warehouses, or products for different delivery addresses.

For more information about consignment, see Consignments.

Related Information

Order Splitting Implementation
The SAP Commerce processengine
Order Management Services Module

Order Splitting Implementation

OrderSplittingService

OrderSplittingService is a set of methods that split an order into sets of order entries. For each set of order entries, a consignment is created. This service is very useful when passing an order to a warehouse for realization.

`splitOrderForConsignment`: calls `splitOrderForConsignmentNotPersist` and persists data.

`splitOrderForConsignmentNotPersist`:

Spring XML declaration:

```
<bean id="orderSplittingService" class="de.hybris.platform.ordersplitting.impl.DefaultOrderSplittir
        <property name="modelService" ref="modelService"/>
        <property name="consignmentService" ref="consignmentService"/>
        <property name="strategiesList">
                <list>
                        <!-- <ref bean="splitByAvailableCount"/> -->
                        <ref bean="splitByDeliveryMode"/>
                        <ref bean="splitByNamedDeliveryDate"/>
                        <ref bean="splitByWarehouse"/>
                </list>
        </property>
</bean>
```

Splitting Strategy

The `strategiesList` stores a list of strategies for how to split orders. You can add a new strategy to split an order in a different way. To do so, add a new splitting strategy that implements the interface `SplittingStrategy` and add it to the list of strategies above.

If splitting is not necessary, you can declare this service without any strategy. In this case, only one consignment is created.

```
public interface SplittingStrategy
{

        /**
         * Perform the strategy.
         *
         * @param orderEntryGroup
         *           the order entry list
         *
         * @return the list< OrderEntryGroup>
         */
        List<OrderEntryGroup> perform(final List<OrderEntryGroup> orderEntryGroup);

        /**
         * After splitting.
         *
         * @param group
         *           the group
         * @param createdOne
         *           the created one
         */
        void afterSplitting(final OrderEntryGroup group, final ConsignmentModel createdOne);
}
```

- `perform`: This method must contain logic for implementing the splitting of an **orderEntry**. **OrderEntryGroup** is a group of order entries where you can add further information about the group that was split. This information is passed to the **afterSplitting** method and can be used there to write some additional data to a consignment. This class extends **ArrayList<AbstractOrderEntryModel>** for a map of parameters and adds the function `getEmpty`, which makes a copy of the parameters and returns an empty list of entries. This is very useful when you want to create your own grouping routine.
- `afterSplitting`: This is invoked for every group for which a consignment is created. There is the opportunity to fill or overwrite some attributes for a consignment (**createdOne**). For example, when splitting an order by warehouse, you can implement an optimization method that chooses the best warehouse to resolve an order and store this optimized result in **OrderEntryGroup** parameters. In **afterSplitting**, you can then use this parameter and simply transfer it into a createdOne warehouse parameter so that you do not have to do it twice (once for split order entries and then again to add the chosen warehouse into a consignment).

AbstractSplittingStrategy

`AbstractSplittingStrategy` is a tool class to make your own splitting strategy. It implements a grouping algorithm based on a grouping object. All you have to do is to choose an object that groups orders from `AbstractOrderEntryModel` and in `afterSplitting` you receive it in a parameter. You can add additional information for a specified consignment (for example fill an additional field added by you) or add some comments needed to resolve a consignment correctly in the warehouse.

Usage Example: SplitByDeliveryMode

This is an example for splitting an order according to the varying delivery modes contained in an order. It uses `AbstractSplittingStrategy` class as a template.

```
public class SplitByDeliveryMode extends AbstractSplittingStrategy
{

        @Override
        public Object getGroupingObject(final AbstractOrderEntryModel orderEntry)
        {
                return orderEntry.getDeliveryMode();
        }

        @Override
        public void afterSplitting(final Object groupingObject, final ConsignmentModel createdOne)
        {
                createdOne.setDeliveryMode((DeliveryModeModel) groupingObject);
```

```
        }

}
```

Usage Example: SplitByNamedDeliveryDate

This is an example for splitting an order according to the varying delivery dates contained in an order. It uses `AbstractSplittingStrategy` class as a template. It only gets `orderEntry.getNamedDeliveryDate()` and uses it for grouping.

```
public class SplitByNamedDeliveryDate extends AbstractSplittingStrategy
{

        @Override
        public Object getGroupingObject(final AbstractOrderEntryModel orderEntry)
        {

                return orderEntry.getNamedDeliveryDate();
        }

        @Override
        public void afterSplitting(final Object groupingObject, final ConsignmentModel createdOne)
        {
                createdOne.setNamedDeliveryDate((Date) groupingObject);

        }

}
```

Split by Warehouse

This strategy is rather complex since choosing warehouses is an optimization task. In the platform, it is implemented as a strategy that chooses a warehouse that can realize the order based on the stock level in the database. The algorithm matches as many order entries as possible to any one warehouse (in other words: do not split if not necessary). If more than one warehouse can fulfill the order, one from the set is randomly chosen. Subject to existing order options and the warehousing system, additional requirements besides product availability and stock levels may have to be considered, causing the strategy to be further extended.

WarehouseService

In this service, there are two methods that return a list of warehouses that could be used to resolve consignments. This service only provides utility methods for order splitting and consignment creation.

getWarehouses: Returns warehouses that can realize a complete order entry list. It is based on stock level assignment, but does not check the actual amount in stock. It should be used where the exact stock level does not matter within the scope of the order process.

getWarehousesWithProductsInStock: Returns warehouses that can realize a given, single order entry. It checks also the amount on order and the stock level. Note that due to the complexity of such a check the scope is a single order entry, not the entire list.

ConsignmentService

A Consignment provides information to warehouses which products belong to one delivery. It defines the communication between warehouse andSAP Commerce platform.

ConsignmentService offers a set of methods that help to create consignments.

createConsignment: Creates consignments. This service needs the order, a list of order entries (subset of the entries of an order, normally) and the code of the new consignment.

```
ConsignmentModel createConsignment(final OrderModel order, final String code, final List<AbstractO
                    throws ConsignmentCreationException;
```

getWarehouse: Returns the warehouse that is chosen for the realization of delivery for a set of order entries. The default realization of this service simply randomly returns any one of the warehouses that can realize delivery.

```
WarehouseModel getWarehouse(final List<AbstractOrderEntryModel> orderEntries);
```

Payments

Payments enables you to integrate external payment service providers to handle electronic payments, offering a flexible approach to multi-channel online payment methods.

Use Case

Payments allows you to integrate external Payment Service Providers (PSPs) into SAP Commerce to enable electronic payments on your storefront.

Payments supports the integration of payment gateways into SAP Commerce by grouping the adapters supporting the cooperation with external payment service providers.

Advantages

Payments can help you to connect to the payment service providers accessible through the PSP-specific payment adapter. Payments can be extended with a customized adapter to support cooperation with almost any payment service provider.

Use Payments to:

- Increase revenue by providing customers with multiple payment options
- Eliminate complexity and reduce cost by connecting to a major PSP
- Enable immediate authorization through automated payment validation
- Centralize management of payment processing for one or more sales channels
- Ensure a secure link between you, your customer, and the credit card processor

Related Information

acceleratorservices Extension
commerceservices Extension

Payment Integration

You can integrate external payment providers into the SAP Commerce payment framework.

The **payment** extension defines interfaces for passing information to an external payment provider for payment action.

Integration Model

The following sections describe the commands required to integrate an external payment provider into the SAP Commerce payment framework.

Implementing Commands

The following commands are responsible for passing data from the payment service to an external payment provider. The base class for each of these commands is the **Command** class.

```
public interface Command<Request, Result>
{
        /**
         * perform command
         *
         * @param request
         *            request to perform
         * @return result of command
         */
        Result perform(Request request);
}
```

AuthorizationCommand

```
/**
 * Command for handling card authorizations. Card authorization is the first step in the card payme
 * amount of money remains "locked" on the card account until it is captured or until authorizatior
 * has expired.
 */
public interface AuthorizationCommand extends Command<AuthorizationRequest, AuthorizationResult>
{
        //empty
}
```

CaptureCommand

```
/**
 * Command for handling card authorization captures. Capturing an authorization means the authorize
 * actually transferred from the card holder account to the merchant account. The capture operatior
 * successful authorization that has not yet expired.
 */
public interface CaptureCommand extends Command<CaptureRequest, CaptureResult>
{
        //empty
}
```

PartialCapture

```
/**
 * Command for handling partial card authorization captures.
 */
public interface PartialCaptureCommand extends Command<PartialCaptureRequest, CaptureResult>
{
        //empty
}
```

voidCommand

```
/**
 * Command for handling voiding capture or credit. Refund means to cancel a capture or credit reque
 * be voided only if the payment service provider has not already submitted the capture or credit c
 */
public interface VoidCommand extends Command<VoidRequest, VoidResult>
{
        // empty
}
```

FollowOnRefundCommand

```
/**
 * Command for handling follow-on refunds. In a follow-on refund, money is returned to the customer
 * with the order or previous transaction. This is in contrast to the stand-alone refund {@link Sta
 */
public interface FollowOnRefundCommand<T extends AbstractRequest> extends Command<T, RefundResult>
{
        RefundResult perform(T request);
}
```

StandaloneRefundCommand

```
/**
 * Command for handling stand-alone refunds. Stand-alone refund means to return back money to custo
 * associated with any order or previous transactions. Just passes money from one account to anothe
 * {@link FollowOnRefundCommand}
 *
 */
public interface StandaloneRefundCommand<T extends AbstractRequest> extends Command<T, RefundResult
{
        RefundResult perform(T request);
}
```

Providing Configuration

This step is accomplished based on the command results, which determine whether the payment provider can serve the card.

IsApplicable command

```
/**
 * Command that each payment provider must implement - configuration that checks if for specified a
 * provider is applicable
 */
public interface IsApplicableCommand extends Command<IsApplicableCommandReqest, IsApplicableCommanc
{
        //empty
}
```

The command sends a request containing two fields that provide information about the card, namely the card information and its 3D secure status:

```
public class IsApplicableCommandReqest
{
        private final BasicCardInfo card;
        private final boolean threeD;
```

```
....
}
```

The information given in these two fields (card information, 3D secure status) determines whether the payment service provider can serve the particular card. The result is a Boolean true/false response.

Note that a payment transaction that was started by one of the payment service providers must be finished by same provider. This is realized in the payment framework.

Spring Configuration

Define a command factory bean for your payment provider, as shown in the following example. This example shows the mockup implementation provided in the `commerceservices` extension in `mock-payment-spring.xml`.

```
<bean name="mockupCommandFactory" class="de.hybris.platform.payment.commands.factory.impl.DefaultCo
        <property name="paymentProvider" value="Mockup"/>
        <property name="commands">
                <map>
                        <entry>
                                <key>
                                        <value type="java.lang.Class">de.hybris.platform.payment.co
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.IsApplicableM
                        </entry>
                        <entry>
                                <key>
                                        <value type="java.lang.Class">de.hybris.platform.payment.co
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.Authorizatior
                        </entry>
                        <entry>
                                <key>
                                        <value type="java.lang.Class">de.hybris.platform.payment.co
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.SubscriptionA
                        </entry>
                        <entry>
                                <key>
                                        <value type="java.lang.Class">de.hybris.platform.payment.co
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.CaptureMockCo
                        </entry>
                        <entry>
                                <key>
                                        <value type="java.lang.Class">de.hybris.platform.payment.co
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.PartialCaptur
                        </entry>
                        <entry>
                                <key>
                                        <value type="java.lang.Class">de.hybris.platform.payment.co
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.EnrollmentChe
                        </entry>
                        <entry>
                                <key>
                                        <value type="java.lang.Class">de.hybris.platform.payment.co
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.VoidMockComma
                        </entry>
                        <entry>
                                <key>
                                        <value type="java.lang.Class">de.hybris.platform.payment.co
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.FollowOnRefur
                        </entry>
```

```
                <entry>
                                <key>
                        <value type="java.lang.Class">de.hybris.platform.payment.commands.Standalone
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.StandaloneRef
                         </entry>
                 <entry>
                                <key>
                        <value type="java.lang.Class">de.hybris.platform.payment.commands.CreateSubs
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.CreateSubscri
                         </entry>
                <entry>
                                <key>
                        <value type="java.lang.Class">de.hybris.platform.payment.commands.DeleteSubs
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.DeleteSubscri
                         </entry>
                 <entry>
                                <key>
                        <value type="java.lang.Class">de.hybris.platform.payment.commands.GetSubscri
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.GetSubscripti
                                </entry>
                  <entry>
                                <key>
                        <value type="java.lang.Class">de.hybris.platform.payment.commands.UpdateSubs
                                </key>
                                <bean class="de.hybris.platform.payment.commands.impl.UpdateSubscri
                         </entry>
                  </map>
                  </property>
        </bean>
```

Payment Business Layer Framework

This document describes the payment service functionality.

The payment functionality is provided by the **PaymentService** class, which contains methods responsible for several operations on orders.

Payment Service Composition

The business layer consists of only one service that provides all necessary methods. If an error occurs, then the **AdapterException** exception is thrown. See the code sample for the definition of the service and its methods.

PaymentService.java

```
/**
 * service can serve payments based on orders
 */
public interface PaymentService
{
        /**
         * authorize payment
         *
         * @param merchantTransactionCode
         *           the transaction code
         * @param amount
         *           the amount
         * @param currency
         *           the currency
         * @param deliveryAddress
         *           the delivery address
```

```
     * @param paymentAddress
     *           the payment address
     * @param card
     *           the card
     * @return Payment Transaction Entry
     * @throws AdapterException
     */
    PaymentTransactionEntryModel authorize(final String merchantTransactionCode,
                                    final BigDecimal amount, final Currency currency,
                                    final AddressModel deliveryAddress,
                                    final AddressModel paymentAddress, final CardInfo card)
                                    throws AdapterException;

    /**
     * authorize payment
     *
     * @param transaction
     *           the payment transaction
     * @param amount
     *           the amount
     * @param currency
     *           the currency
     * @param deliveryAddress
     *           the delivery address
     * @param paymentAddress
     *           the payment address
     * @param card
     *           the card
     * @return Payment Transaction Entry
     * @throws AdapterException
     */
    PaymentTransactionEntryModel authorize(final PaymentTransactionModel transaction, final Bi
            final Currency currency, final AddressModel deliveryAddress, final AddressModel pa
            final CardInfo card) throws AdapterException;

    /**
     * capture payment
     *
     * @param transaction
     *           payment transaction
     * @return payment transaction entry
     * @throws AdapterException
     */
    PaymentTransactionEntryModel capture(PaymentTransactionModel transaction) throws AdapterEx

    /**
     * Cancel payment
     *
     * @param transaction
     *           payment transaction
     * @return payment transaction entry
     * @throws AdapterException
     */
    PaymentTransactionEntryModel cancel(final PaymentTransactionEntryModel transaction) throws

    /**
     * Refund transaction
     *
     * @param transaction
     *           payment transaction
     * @param amount
     *           amount to refund
     * @return payment transaction entry
     * @throws AdapterException
     */
    PaymentTransactionEntryModel refundFollowOn(final PaymentTransactionModel transaction, BigD
                   throws AdapterException;

    PaymentTransactionEntryModel refundStandalone(StandaloneRefundRequest request) throws Adapt

    PaymentTransactionEntryModel partialCapture(final PaymentTransactionModel transaction, BigD
                   throws AdapterException;

    /**
```

```
	 * Get {@link PaymentTransactionModel} by code
	 *
	 * @param code
	 *           the code of the {@link PaymentTransactionModel} we were looking for
	 * @return the {@link PaymentTransactionModel}
	 */
	PaymentTransactionModel getPaymentTransaction(final String code);

	/**
	 * Get {@link PaymentTransactionEntryModel} by code
	 *
	 * @param code
	 *           the code of the {@link PaymentTransactionModel} we were looking for
	 * @return the {@link PaymentTransactionModel}
	 */
	PaymentTransactionEntryModel getPaymentTransactionEntry(final String code);
}
```

Each service method returns an entry that is related to the specific payment transaction. It stores the data after each step of the payment transaction has finished successfully. To learn more about the payment transaction entry structure, see the following code sample for **PaymentTransactionEntry**.

payment-items.xml

```
<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:noNamespaceSchemaLocation="items.xsd">

	<enumtypes>

		<enumtype code="CreditCardType" autocreate="false" generate="false">
			<value code="maestro" />
			<value code="switch" />
			<value code="mastercard_eurocard" />
		</enumtype>
		<enumtype code="PaymentTransactionType" autocreate="true"
			generate="true">
			<value code="AUTHORIZATION" />
			<value code="CAPTURE" />
			<value code="PARTIAL_CAPTURE" />
			<value code="REFUND_FOLLOW_ON" />
			<value code="REFUND_STANDALONE" />
			<value code="CANCEL" />
		</enumtype>

	</enumtypes>

	<relations>
		<relation generate="true" localized="false"
			code="PaymentTransaction2PaymentTransactionEntry" autocreate="true">
			<sourceElement type="PaymentTransaction" qualifier="paymentTransaction"
				cardinality="one"></sourceElement>
			<targetElement type="PaymentTransactionEntry"
				qualifier="entries" cardinality="many" collectiontype="list"></tarq
		</relation>
		<relation generate="true" localized="false"
			code="Order2PaymentTransaction" autocreate="true">
			<sourceElement type="Order" qualifier="order"
				cardinality="one"></sourceElement>
			<targetElement type="PaymentTransaction" qualifier="paymentTransactions"
				cardinality="many" collectiontype="list"></targetElement>
		</relation>
	</relations>

	<itemtypes>

		<itemtype code="PaymentTransaction" autocreate="true"
			generate="true" jaloclass="de.hybris.platform.payment.jalo.PaymentTransacti
			<deployment table="PaymentTransactions" typecode="2100" />
			<attributes>
				<attribute qualifier="requestId" type="java.lang.String">
					<persistence type="property"></persistence>
```

```
                                </attribute>
                                <attribute qualifier="requestToken" type="java.lang.String">
                                        <persistence type="property"></persistence>
                                </attribute>
                                <attribute qualifier="paymentProvider" type="java.lang.String">
                                        <persistence type="property"></persistence>
                                </attribute>
                        </attributes>
                </itemtype>

                <itemtype code="PaymentTransactionEntry" autocreate="true"
                        generate="true" jaloclass="de.hybris.platform.payment.jalo.PaymentTransacti
                        <deployment table="PaymntTrnsctEntries" typecode="2101" />
                        <attributes>
                                <attribute qualifier="type" type="PaymentTransactionType">
                                        <persistence type="property"></persistence>
                                </attribute>
                                <attribute qualifier="amount" type="java.math.BigDecimal">
                                        <persistence type="property"></persistence>
                                </attribute>
                                <attribute qualifier="currency" type="Currency">
                                        <persistence type="property"></persistence>
                                </attribute>
                                <attribute qualifier="time" type="java.util.Date">
                                        <persistence type="property"></persistence>
                                </attribute>
                                <attribute qualifier="transactionStatus" type="java.lang.String">
                                        <persistence type="property"></persistence>
                                </attribute>
                                <attribute qualifier="transactionStatusDetails" type="java.lang.Str
                                        <persistence type="property"></persistence>
                                </attribute>
                                <attribute qualifier="requestToken" type="java.lang.String">
                                        <persistence type="property"></persistence>
                                </attribute>
                                <attribute qualifier="requestId" type="java.lang.String">
                                        <persistence type="property"></persistence>
                                </attribute>
                        </attributes>
                </itemtype>

        </itemtypes>

</items>
```

Payment Service Methods Description

Method Name	Description
Authorization	There are two possible methods of authorization: 1. Takes **merchantTransactionCode** as a parameter and fills all needed data in request based on it. 2. Gives more flexibility and allows user to pass each data separately.
Capture	After authorization is finished, the normal way is to **Capture** payment. All you need to do is to pass the payment transaction.
Partial Capture	Perform capture on part of the previously authorized transaction.
Cancel	Before **Capture** is done, it is possible to cancel payment - this means that all transactions are canceled.
Refund Follow On	This method performs refund of the part of payment.

Method Name	Description
Refund Standalone	This method allows you to do refund without previous authorization. It takes card information as a parameter.

For more information, see Payment Integration.

Payment with Multiple Credit Cards

In some countries, such as in the U.S., it is possible to pay for products in one order using multiple credit cards. SAP Commerce supports this multiple credit card payment process by extending the implementation of existing methods.

`PaymentTransaction` provides all necessary data for external PSPs, held in the `Cart` object. The `Cart` object is based on `AbstractOrder`, and it holds order-related data until the customer performs the successful payment transaction. After a successful transaction, the payment-related data is cloned to the `Order` object, where it can be stored for further tracking. Additionally, `PaymentTransaction` holds the `PaymentInfo` reference, containing information about how the transaction was created. This can be customized to meet unique business requirements.

`PaymentTransactionEntry` can now handle a Return Merchandise Authorization (RMA). This is a unique identifier generated by the payment service itself.

With `PaymentInfo`, it is possible to store information about multiple payments per order, each with a different billing address.

`PaymentAuthorizationTransaction` provides status information in string format, so that the interpretation of the result should be delegated to an AuthorizationService. The customer can implement this service based on their business logic requirements. Generally, the service checks if the `PaymentAuthorizationTransaction` is successful or still valid, by applying business rules defined by the merchant.

Code Examples

The following code examples show methods that support payment with multiple cards:

'Return Request' create, search

```
@Resource
private ReturnService returnService;

private OrderModel order;

// create
final ReturnRequestModel request = returnService.createReturnRequest(order);
// RMA generation
request.setRMA(returnService.createRMA(request));
// list alle existing 'return request' of the specified order
final List<ReturnRequestModel> requests = returnService.getReturnRequests(order.getCode());
```

'ReplacementOrder' create

```
@Resource
private ReturnService returnService;

private OrderModel order;

final ReturnRequestModel request = returnService.createReturnRequest(order);
request.setRMA(returnService.createRMA(request));
```

```
final ReplacementOrderModel replacementOrder = returnService.createReplacementOrder(request);

final AbstractOrderEntryModel originalEntry = order.getEntries().iterator().next();
final ReplacementEntryModel replacementEntry = returnService.createReplacement(request, originalEnt
                                    Long.valueOf(3), ReturnAction.IMMEDIATE, ReplacementReason.LAT

returnService.addReplacementOrderEntries(replacementOrder, Arrays.asList(replacementEntry));
```

'Refund' create

```
@Resource
private ReturnService returnService;

private OrderModel order;

final ReturnRequestModel request = returnService.createReturnRequest(order);
request.setRMA(returnService.createRMA(request));

final AbstractOrderEntryModel originalEntry = order.getEntries().iterator().next();

final RefundEntryModel refundEntry = returnService.createRefund(request, originalEntry, "no.1",
                      Long.valueOf(3), ReturnAction.IMMEDIATE, RefundReason.LATEDELIVERY);
```

'Refund' calculation

```
@Resource
private OrderService orderService;
@Resource
private ReturnService returnService;
@Resource
private RefundService refundService;

// the original order the customer wants to have a refund for
private OrderModel order;

// lets create a RMA for it
final ReturnRequestModel request = returnService.createReturnRequest(order);
returnService.createRMA(request);

// based on the original order the call center agent creates a refund order kind of preview (**)
final OrderModel refundOrderPreview = refundService.createRefundOrderPreview(order);

// all following "refund processing", will be based on the refund order instance (copy of the origi
final AbstractOrderEntryModel productToRefund = refundOrderPreview.getEntries().iterator().next();

// create the preview "refund"
final RefundEntryModel refundEntry = returnService.createRefund(request, productToRefund, "no.1",
                                    Long.valueOf(1), ReturnAction.IMMEDIATE, RefundReason.LATEDELIV

// calculate the preview refund ...
refundService.apply(Arrays.asList(refundEntry), refundOrderPreview);

// based on presented "preview" (see **) the customer decides if he wants to accept the offered ref
// ... and in the case the customer agrees, the call center agent will now recalculate
// the "original" order (which is attached to the assigned 'request')
refundService.apply(refundOrderPreview, request);
```

Authentication with Credit Card by Saved Token

To comply with PCI regulations, we recommend that you store only a subscription ID or tokenized representation on the SAP Commerce side, and allow the Payment Service Provider to store the full credit card information. This also accelerates the payment authorization process.

→ Tip

The **CreditCardPaymentInfo** has a place to save the **subscriptionID** when sensitive credit card details, such as the full card number and CVV number, are saved in a third-party payment system. The **subscriptionID** is used in the SAP Commerce payment service during order placement.

Payment Methods for Subscriptions

The **Payment API** contains the following methods for dealing with subscriptions. These are provided by the `payment` extension.

de.hybris.platform.payment.PaymentService

```
//
                // SUBSCRIPTION BASED AUTHORIZATION
                //
                PaymentTransactionEntryModel authorize(final String merchantTransactionCode, final
                final AddressModel deliveryAddress, final String subscriptionID) throws AdapterExce

                PaymentTransactionEntryModel authorize(final PaymentTransactionModel transaction, f
                final AddressModel deliveryAddress, final String subscriptionID) throws AdapterExce

                PaymentTransactionEntryModel authorize(final PaymentTransactionModel transaction, f
                final AddressModel deliveryAddress, final AddressModel paymentAddress, final CardIn
                throws AdapterException;

                //
                // CREATE SUBSCRIPTION
                //

                /**
                * Creates a subscription on the payment provider side and stores sensitive data the
                * and refundStandalone calls will not have to provide the sensitive information, su
                * Call this method after a successful authorize txn and provide THE SAME billing ar
                * card and billing info is not saved in the returning PaymentTransactionEntryModel
                * PURPOSE.
                */
                public NewSubscription createSubscription(final PaymentTransactionModel transactior
                final CardInfo card) throws AdapterException;

                /**
                * Creates a subscription at the payment provider side and stores sensitive data the
                * and refundStandalone calls will not have to provide the sensitive information, su
                * This method does not need an authorized payment transaction, the authorization is
                * by the payment provider.
                */
                public NewSubscription createSubscription(final String merchantTransactionCode, fir
                final Currency currency, final AddressModel paymentAddress, final CardInfo card)
                throws AdapterException;

                //
                // UPDATE SUBSCRIPTION
                //

                /**
                * Updates the data of the subscription at the payment provider.
                *
                */
                public PaymentTransactionEntryModel updateSubscription(final String merchantTransac
                final String paymentProvider, final AddressModel paymentAddress,
                final CardInfo card) throws AdapterException;

                //
                // GET SUBSCRIPTION
                //

                public PaymentTransactionEntryModel getSubscriptionData(final String merchantTransa
                final String paymentProvider, final BillingInfo billingInfo,
                final CardInfo card) throws AdapterException;

                //
                // DELETE SUBSCRIPTION
```

```
                //

                public PaymentTransactionEntryModel deleteSubscription(final String merchantTransac
                final String paymentProvider) throws AdapterException;
```

ℹ Note

In the current release, there is no support for **Subscription Delete**, offered by Cybersource.

Using Payment Subscription

Sample Calls

```
// Plain authorization
                final PaymentTransactionEntryModel ptem = paymentService.authorize("1234567", AMOUN
                deliveryAddress, card);

                // Create subscription out of the authorization
                final NewSubscription ns = paymentService.createSubscription(ptem.getPaymentTransac
                final String subscriptionID = ns.getSubscriptionID();

                // Use the subscription to make an authorization
                final PaymentTransactionEntryModel txn1 = paymentService.authorize("1234568", AMOUN

                // Get subscription data
                final PaymentTransactionEntryModel txn2 = paymentService.getSubscriptionData("1234
                billInfo, cardInfo);

                // Update the subscription
                final PaymentTransactionEntryModel txn3 = paymentService.updateSubscription("12345
                null, cardInfo);
```

payment Extension

The `payment` extension constitutes the foundation for the payment framework in the SAP Commerce.

The `payment` extension, together with payment gateway adapter, works to provide the payment service provider functionality to the SAP Commerce. The `payment` extension is, by design, ready to support customer-made adapters to communicate with any external payment service provider.

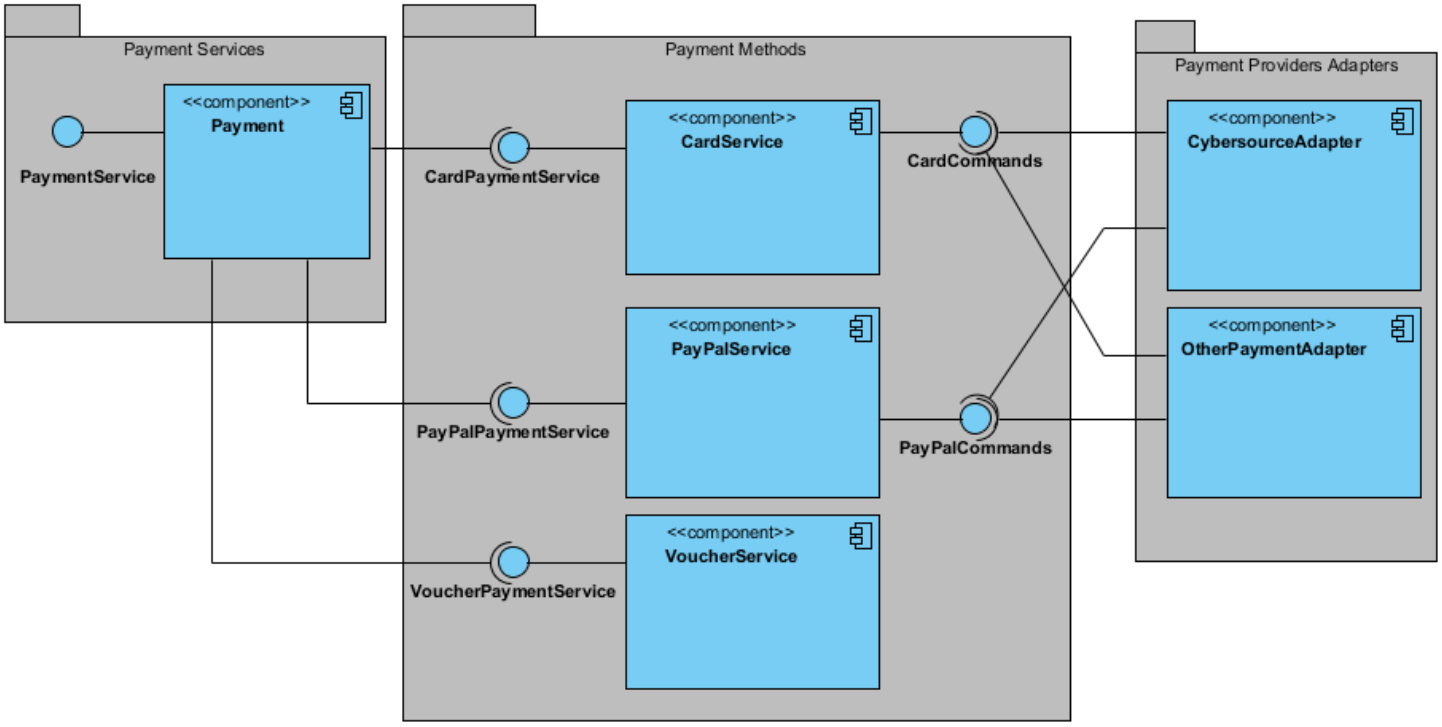
ℹ Note

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.

About the payment Extension

The `payment` extension allows you to support a SAP Commerce environment. As of the current version of this extension, it only supports card payments. Technically speaking, the main task of the payment extension is to provide interfaces that let you build an adapter to integrate external payment service providers into the extension allows you to support payment operations for the SAP Commerce system and store the data necessary to support transactions.

Architecture



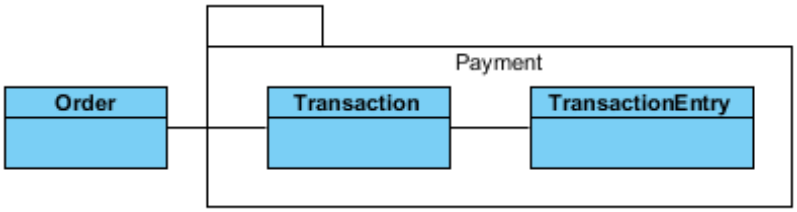
The payment provider functionality may be depicted as split into 3 layers while technically delivered by 2 extensions: the `payment` extension and the `paymentAdapter` extension. The name of the particular implementation of the `paymentAdapter` extension may differ depending on your choice. The adapter for the CyberSource payment provider is named `cyberSource`. There can be several payment adapters going along with each other, connecting payment extension to the particular payment service provider. Check the diagram above to see how they are related to the payment methods layer.

The `payment` extension contains implementation for 2 layers:

- Payment Services: It works on orders and allows you to authorize and take payment of an order.
- Payment Methods: It defines different payment methods and sets of requests and results for each of the actions.

To communicate between the payment extension and the `paymentAdapter` extension, you need to implement payment command interfaces in your extension that are compatible with the `payment` extension (for details see Payment Integration). The `payment` extension also implements the command factory, which is responsible for creating payment commands. Right now, only card payment is implemented.

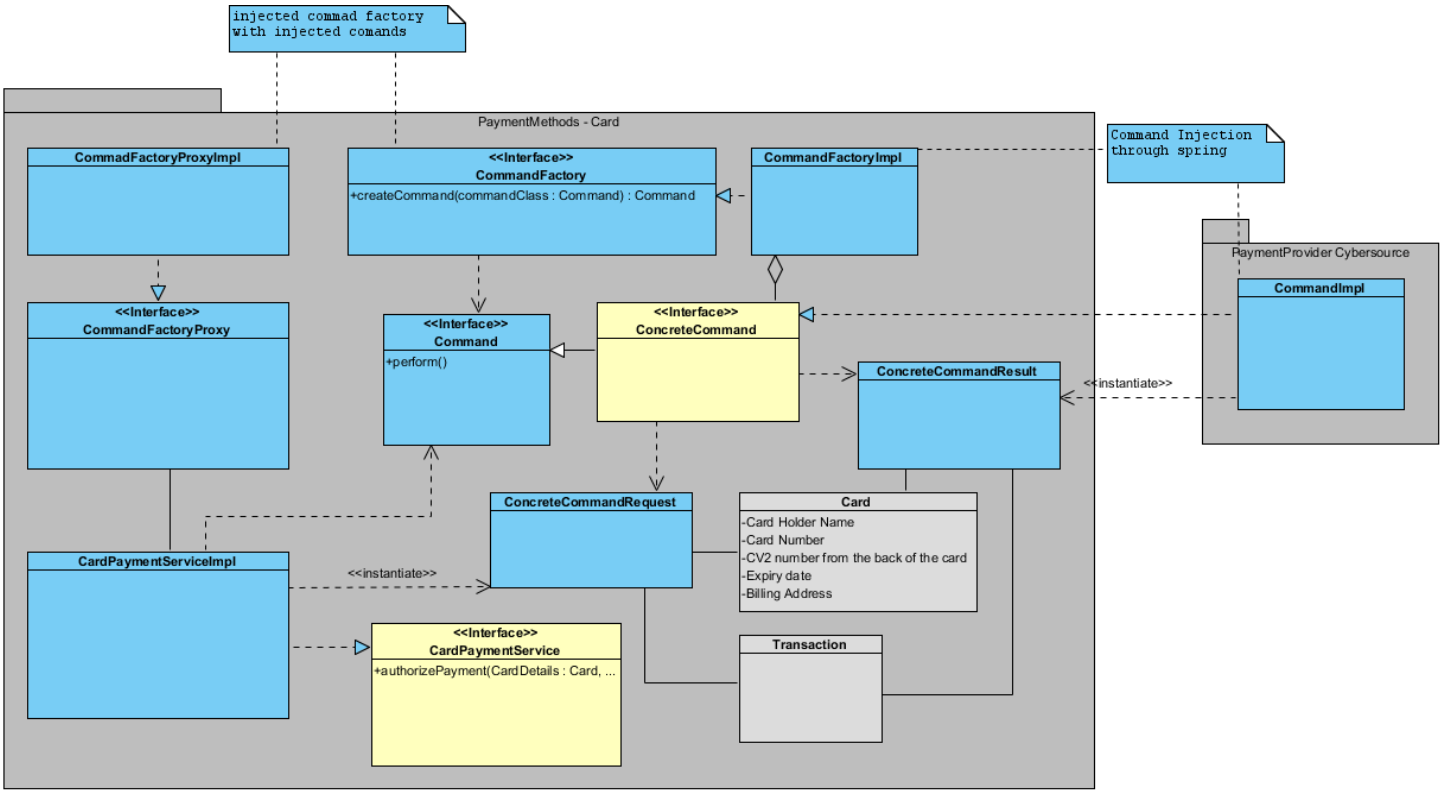
Data



The `payment` extension does not store any data for used payment cards. It only stores:

- Payment transaction data
- Payment history data

Interfaces



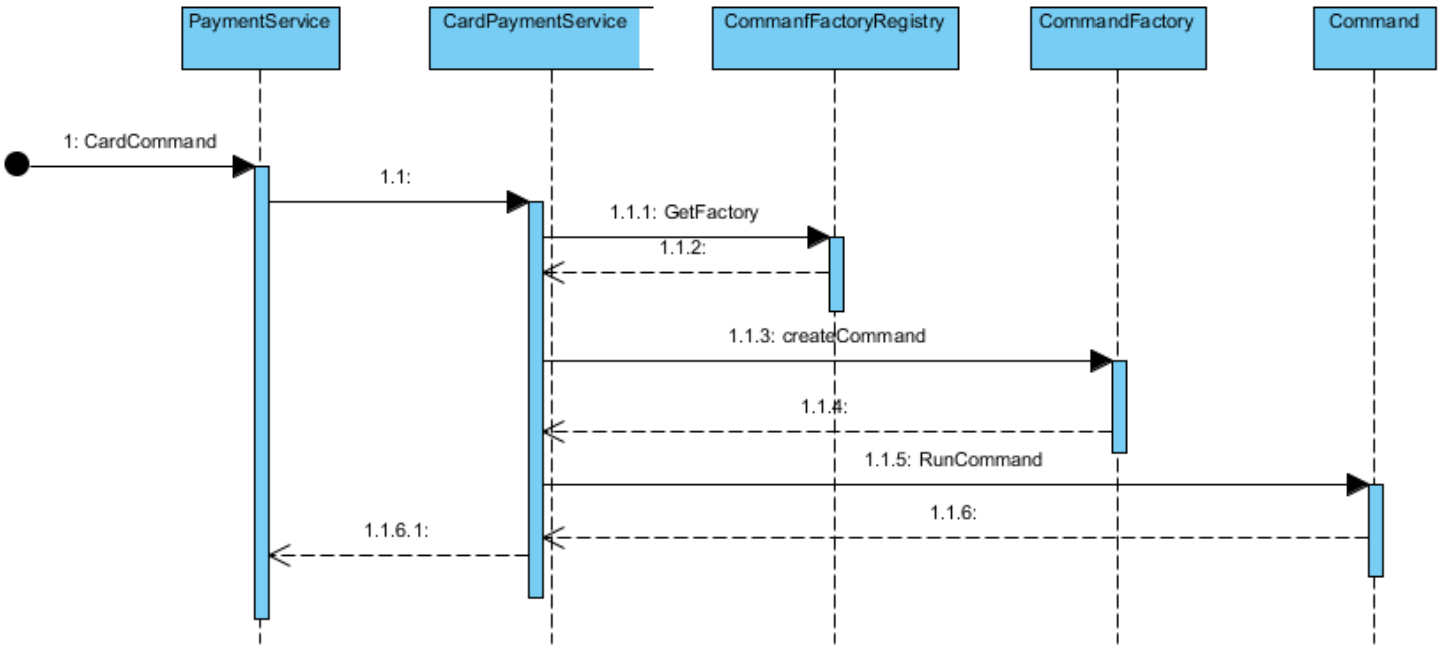
The grey color is used for transfer classes: card and transaction. They are used to pass data to the payment provider.

The yellow color is used for the representation of concrete commands.

Behavior

Normal behavior starts from payment service layer. It fires the service in the card payment service. Next, CommandFactoryRegistry returns a command that is used to finish the request. In the next step, this command is used to contact the payment provider and interpret its answer. At the end, the result is passed into the output of the service.

Typical behavior is presented in the diagram below.



Expected Responses

Each payment command returns the following information defined in **public abstract class AbstractResult**:

```
public abstract class AbstractResult
{
        private String merchantTransactionCode;
        private String requestId;
        private String requestToken;
        private String reconciliationId;
        private TransactionStatus transactionStatus;
        private TransactionStatusDetails transactionStatusDetails;
```

Additionally there is information returned specific to each command as shown in the table below.

Command	Result	Result Description
Authorization	de.hybris.platform.payment.commands.result.AuthorizationResult	Returns information about: currency, money amount, account balance, authorization time, authorization code, AVS status, CVN status, and payment provider.
Capture	de.hybris.platform.payment.commands.result.CaptureResult	Returns information about: currency, total money amount, and request time.
CardValidation	de.hybris.platform.payment.commands.result.CardValidationResult	Returns information about: validation errors, card issuer, issue country code, and card type.
IsApplicable	de.hybris.platform.payment.commands.result.IsApplicable	Returns information if payment provider is applicable.
Refund	de.hybris.platform.payment.commands.result.RefundResult	Returns information about: currency, total money amount, request time, and payment provider.
Void	de.hybris.platform.payment.commands.result.VoidResult	Void result uses the transactionStatus and transactionStatusDetails from AbstractResult class to inform if voiding transaction was a succes of failure. Possible results: • ACCEPTED: Transaction was accepted. • ERROR: An Error occured. • REJECTED: Transaction was rejected by PSP, payment processor, or bank. However, sometimes it might still be continued under some conditions. For example, the transaction might be

Command	Result	Result Description
		authorized by the issuing bank even when the AVS check fails. Settlement of such a transaction might still be possible. • REVIEW: Manual transaction review is needed. This requires calling either the PSP, processor, or bank and proceeding with the transaction manually.

AVS Status

Address Verification Service Status Values:

```
public enum AvsStatus
{
        PARTIAL_MATCH, //Partial match.
        MATCHED, //Matched.
        NOT_MATCHED, //Not matched.
        NOT_SUPPORTED, //AVS is not supported for this processor or card type.
        INVALID, //The processor returned an unrecognized value for the AVS response.
        NO_RESULT, //No AVS result
        UNRECOGNIZED_RESULT, //Unrecognized AVS result
        SYSTEM_UNAVAILABLE; //System unavailable.
```

CVN Status

Code Verification Number Status Values:

```
public enum CvnStatus
{
        NO_RESULT, //No result code was returned by the processor.
        UNRECOGNIZED_RESULT, // An unrecognized result code was returned by the processor.
        NOT_SUPPORTED, //Card verification is not supported (by issuing bank or by card associatior
        NOT_PROVIDED, //The CVN is on the card but was not included in the request.
        NOT_PROCESSED, //The CVN was not processed by the processor for an unspecified reason.
        NOT_MATCHED, //The CVN did not match.
        MATCHED, //The CVN matched.
        NOT_VALIDATED, //The CVN failed the processor's data validation check.
        REJECTED, //The transaction was determined to be suspicious by the issuing bank.
}
```

Stock Service

Stock Service is part of the `basecommerce` extension. It offers functionality to manage and query product stock level and product availability information, aggregated or for a specific warehouse.

Use Case

The Stock Service is defined by interfaces that enable you to display and manage product stock levels and product availability information for a specific warehouse, or aggregated for multiple warehouses.

Stock Service also comes with strategies for splitting orders into different consignments depending on a product availability. To achieve this, Stock Service uses other parts of the SAP Commerce, for example Warehouse Integration and Order Splitting.

Features

Product Stock Level Information

It is possible to update, access, and display product stock level information, in particular:

- The amount of a certain product that is available (actual, local, and aggregated stock levels)
- Stock status for one or more warehouses or stores
- Forcing always in stock and out of stock for a specific product
- Allow overselling
- Keep track of stock levels without requiring live information from a warehouse

As a shop owner, you can customize the rules for order cancellation - for example, to not permit customers to cancel orders at all.

Product Availability Information

Stock Service makes it possible to manage and display product availability information, for example when a certain amount of a product will be available. It also provides:

- Localized text messages intended for use in front-end applications, that may be also customized
- Strategies for calculating the best availability over multiple warehouses

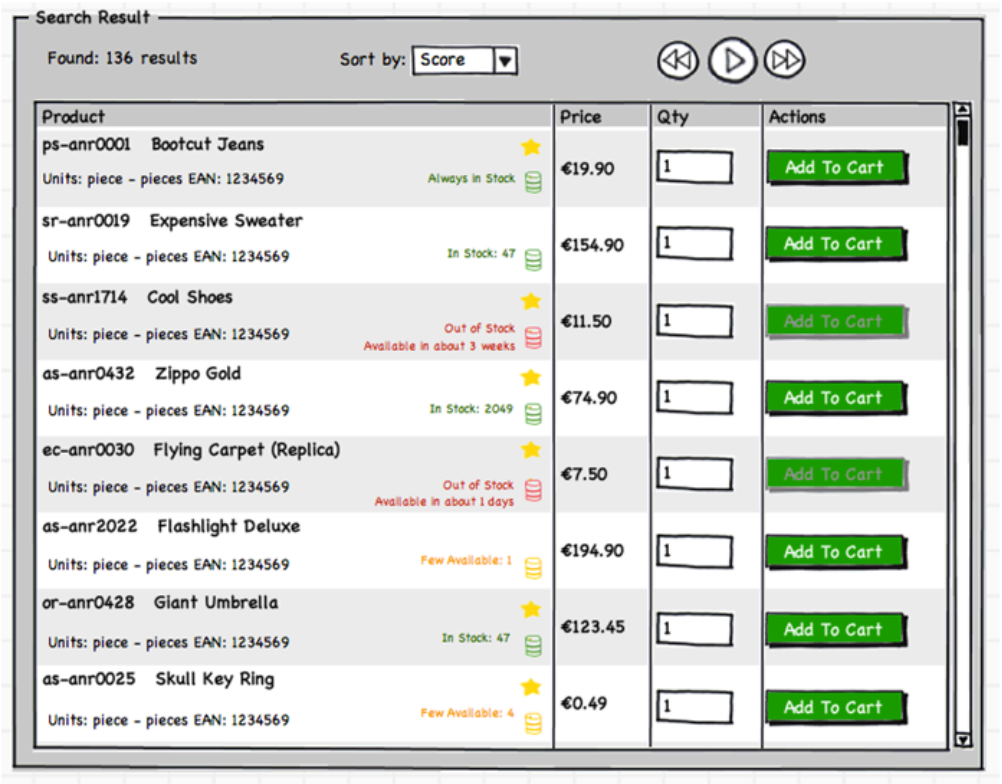


Figure: An example of displaying product stock level and product availability information in a front-end application.

Availability-Based Order Splitting

Depending on the availability of a product, an order may be split into different consignments. This is done by grouping order items into two different groups: **Currently Available** and **Currently Not Available**. The default strategy only takes aggregated stock levels into account.

Related Information

Warehouse Integration
Order Splitting
Order Management Services Module

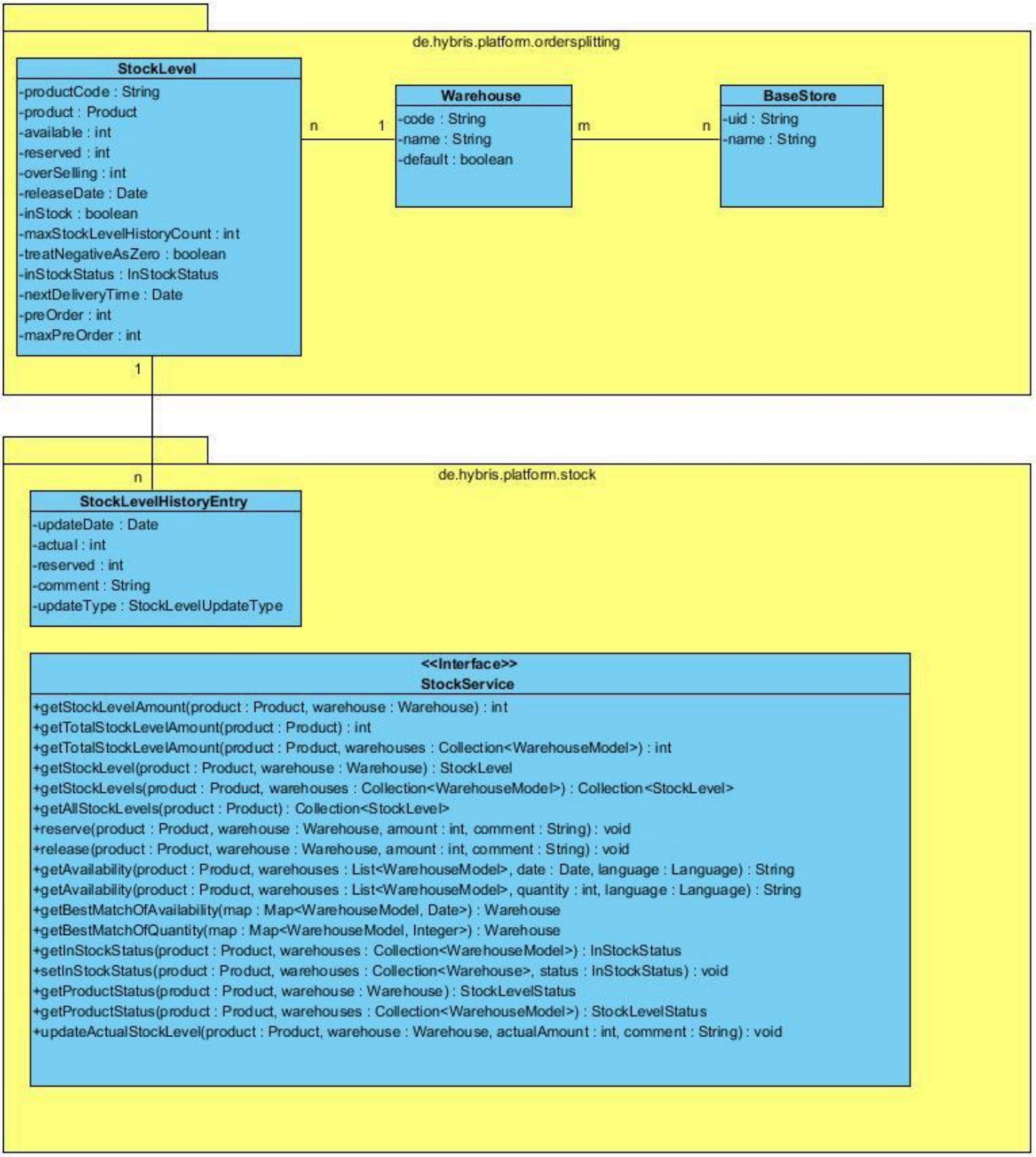
About Stock Service

The Stock Service is defined by interfaces that enable you to display and manage product stock levels and product availability information for a specific warehouse, or aggregated for multiple warehouses.

Stock Service Overview

The Stock Service extends the API with methods and services allowing front-end applications to access and display product stock and availability information. It also interacts with Warehouse Integration and Order Splitting.

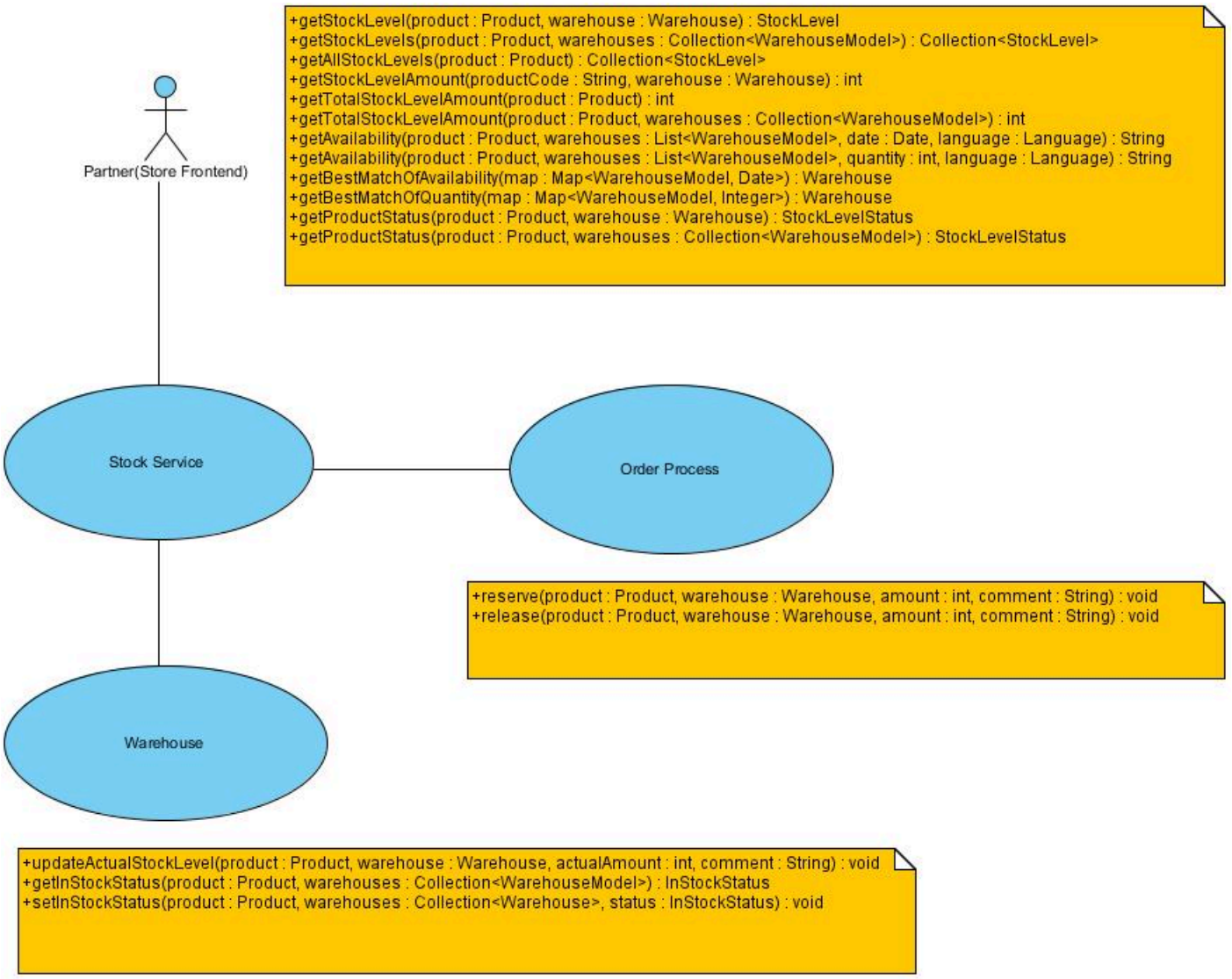
The following figure gives an overview of the Stock Service:



The most important use cases of the Stock Service are the following:

- Updating the actual and local stock levels
- Displaying the total stock level
- Displaying the stock level for a specific warehouse or many warehouses
- Displaying information about product availability for a specific warehouse or many warehouses
- Enabling and disabling **Always In Stock** status
- Enabling and disabling **Always Out of Stock** status
- Configuring overselling of products
- Displaying product stock status

The following figure provides an overview from the perspective of different roles:



Stock Level

The stock level of a product is the available quantity of the specific product at a certain time. In many cases, product stock levels are not updated in real-time, but are instead updated periodically, such as once a day, or once a week. As a result, there may be a difference between the actual stock level and the local (calculated) stock level. The actual stock levels are levels typically reported by a warehouse, such as after an inventory. The local stock level is the calculated value that can be treated as a way of keeping track of sold product quantities, even though we do not know the exact amount that is available in the warehouse.

Retrieving Stock Levels

The aggregated stock level of a certain product is the total stock level for all stores or warehouses where the product is available. The Stock Service enables you to get aggregated stock levels for all warehouses, and aggregated stock levels for specified warehouses, as follows:

Aggregated stock level for all warehouses

```
//ProductModel product = ...
          int totalStockLevelAmount= stockLevelService.getTotalStockLevelAmount(product);
```

Aggregated stock level for specified warehouses

```
//ProductModel product = ...
          //Collection<WarehouseModel> warehouses = ...
          int totalStockLevelAmount= stockLevelService.getTotalStockLevelAmount(product, warehouses
```

Maintaining Stock Levels

The Stock Service keeps track of the number of reserved products since the last update from a warehouse. Reserved products can be treated as sold but still returnable. Consider the following example: a warehouse reports that there are 250 caps available. This is the actual stock level. A customer then buys two caps in a web store. At this point, we would expect the reported stock level to be 248. However, the only real value we have is the one last reported by the warehouse, which is the actual stock level. The local stock level is 248, and to be able to say that there are 248 products available in the warehouse, the Stock Service provides methods to reserve product quantities, which in this case is 2 caps.

Warehouse Stock Levels

After taking inventory in a warehouse, it is possible to update the actual stock level for a given product, as follows:

```
//ProductModel product = ...
          //WarehouseModel warehouse = ...
          //int amount = ...
          //String comment = ...
          stockService.updateActualStockLevel(product, warehouse, amount, comment);
```

Reserve and Release

For customers, it is important to know if they can buy or reserve a product. The Stock Service takes care of these actions as well. The Stock Service supports the operation to reserve a product from a specific warehouse. If there are not enough quantities of a product to be reserved, the `InsufficientStockLevelException` is thrown, as follows:

```
//ProductModel product = ...
          //WarehouseModel warehouse = ...
          //int amount = ...
          //String comment = ...
          try
          {
          stockService.reserve(product, warehouse, amount, comment);
          //business code goes on
          }
          catch(InsufficientStockLevelException e)
          {
          LOG.error("not enough products in stock");
          //exception handling
          }
```

The customer may cancel an order or a part of an order. Then the product (or products) are returned and are available again, as follows:

```
//ProductModel product = ...
          //WarehouseModel warehouse = ...
          //int amount = ...
          //String comment = ...
          stockService.release(product, warehouse, amount, comment);
```

Product Stock Status

Basic stock statuses for a product can be set as follows:

- **`In stock`**: A specific product is indicated as available for given warehouses

- **`Out of stock`**: A specific product is indicated as not available for given warehouses
- A product can also have a custom status that is configured by a warehouse maintainer.

You can display a stock status for a given product for a specific warehouse or for multiple warehouses, as follows:

A specific warehouse

```
//ProductModel product = ...
        //WarehouseModel warehouse = ...
        StockLevelStatus stockLevelStatus = stockService.getProductStatus(product, warehouse);
```

Multiple warehouses

```
//ProductModel product = ...
        //Collection<WarehouseModel> warehouses = ...
        StockLevelStatus stockLevelStatus = stockService.getProductStatus(product, warehouses);
```

Force In Stock

Sometimes there are not enough products available, but the seller still wants to sell them. In this case, the **`FORCEINSTOCK`** can be set for a specified product, as follows:

```
//enable "ForceInStock"
        //ProductModel product = ...
        //Collection<WarehouseModel> warehouses = ...
        stockService.setInStockStatus(product, warehouses, InStockStatus.FORCEINSTOCK);
        //disable "ForceInStock"
        stockService.setInStockStatus(product, warehouses, InStockStatus.NOTSPECIFIED);
```

Force Out Of Stock

The stock status can also be set to **`FORCEOUTOFSTOCK`**, even if the product is still available, as follows:

```
//enable "ForceOutOfStock"
        //ProductModel product = ...
        //Collection<WarehouseModel> warehouses = ...
        stockService.setInStockStatus(product, warehouses, InStockStatus.FORCEOUTOFSTOCK);
        //disable "ForceOutOfStock"
        stockService.setInStockStatus(product, warehouses, InStockStatus.NOTSPECIFIED);
```

Overselling

In some cases it may be useful to allow the sale of more products than are available in a warehouse. In this case, you can allow the warehouse to configure the overselling of such products, as follows:

```
//ProductModel product = ...
        //WarehouseModel warehouse = ...
        //int overSelling = ...
        //ModelService modelService = ...
        StockLevelModel stockLevel = stockService.getStockLevel(product, warehouse);
        stockLevel.setOverSelling(overSelling);
        modelService.save(stockLevel);
```

It is also possible to get the current overselling amount for a product, as follows:

```
//ProductModel product = ...
        //WarehouseModel warehouse = ...
        StockLevelModel stockLevel = stockService.getStockLevel(product, warehouse);
        int overSelling = stockLevel.getOverSelling();
```

Product Availability

The product availability contains information about the availability of a specific product from a specific warehouse or from many warehouses.

Retrieving Product Availability

Product availability returns a configurable availability message. It may display a date of its availability, or a product quantity for a specific warehouse, or for multiple warehouses, as follows:

A single warehouse

```
//ProductModel product = ...
          //WarehouseModel warehouse = ...
          //Date date = ...
          //LanguageModel language = ...
          String dateAvailability = stockService.getAvailability(product, warehouse, date, language
          //int quantity = ...
          String quantityAvailability = stockService.getAvailability(product, warehouse, quantity,
```

Multiple warehouses

```
//ProductModel product = ...
          //List<WarehouseModel> warehouses = ...
          //Date date = ...
          //LanguageModel language = ...
          String dateAvailability = stockService.getAvailability(product, warehouses, date, languag
          //int quantity = ...
          String quantityAvailability = stockService.getAvailability(product, warehouses, quantity,
```

Configuring Availability Messages

To use the availability message for the stock level, it is necessary to configure the following details:

```
//ProductModel product = ...
          //WarehouseModel warehouse = ...
          //int amount = ...
          //Date date = ...
          StockLevelModel stockLevel = stockService.getStockLevel(product, warehouse);
          stockLevel.setMaxPreOrder(amount);
          stockLevel.setNextDeliveryTime(date);
          modelService.save(stockLevel);
```

To implement your own **`ProductAvailabilityStrategy`**, you need to override the getAvailability() and/or **`parse()`** methods. Another option is to define the messages for availability in the `BasecommerceMessages.properties` file, as follows:

BasecommerceMessages.properties

```
//warehouse: warehouse code;
          //product: product code;
          //date: date when the products will be available;
          //availability: amount of product that will be available;
          //total: total amount of product that will be available;
          de.hybris.platform.validation.services.impl.DefaultProductAvailabilityStrategy.availabil
```

```
Product: {product} Availability: {availability} Date: {date}
de.hybris.platform.validation.services.impl.DefaultProductAvailabilityStrategy.total=Tota
```

Stock Level Histories

There is a **`maxStockLevelHistoryCount`** property on each `StockLevel` that indicates how many histories should be saved related to the updates. If this property is anything other than `0` (zero), then unlimited `stockLevelHistories` will be created. This property is defined on the **`StockLevel`** in the `basecommerce-beans.xml` file, and defaults to `0`, indicating that no histories are associated with the **`StockLevel`**. If unlimited histories are desired (for example, to preserve an audit of all interactions with the `StockLevel`), the **`maxStockLevelHistoryCount`** should be set to any number other than zero (the number can be either positive or negative). Stock level histories are created in the **`DefaultStockLevelService`** whenever the `reserve`, `release`, or `updateActualStockLevel` methods are called. Also, if for some reason there is no **`StockLevel`** found, the **`StockLevelService`** will create a new `StockLevel`,and at this time, a stock level history will be created if the **`maxStockLevelHistoryCount`** is not equal to `0`.

Example Use of Stock Service

The following example presents a typical flow of calling methods when using the Stock Service:

```
final StockLevelStatus status = stockService.getProductStatus(product, warehouse);
        //StockLevelStatus status = stockService.getProductStatus(product, warehouses);
        if (StockLevelStatus.INSTOCK.equals(status))
        {
        //rendering "Always_In_Stock"
        }
        else if (StockLevelStatus.OUTOFSTOCK.equals(status))
        {
        //rendering "Out_Of_Stock"
        //disabling the "Add_To_Cart" button
        //int quantity = ...
        //LanguageModel = ...;
        final String availableMessage = stockService.getAvailability(product, warehouse, quantity,
        //String availableMessage = stockService.getAvailability(product, warehouses, quantity, lar
        //rendering availableMessage
        }
        else
        {
        final int amount = stockService.getStockLevelAmount(product, warehouse);
        //int amount = stockService.getTotalStockLevelAmount(product, warehouses);
        //rendering in stock amount
        }
```

Store Locator

The SAP Commerce Store Locator helps customers find brick-and-mortar retail locations in the proximity of a postcode, or using Global Positioning System (GPS) information. Store Locator is an addon for Cart & Checkout, and works with both desktop and mobile solutions.

Use Case

A customer is browsing the online store, but would prefer to purchase an item in a retail outlet, where they can get personal assistance and collect the item immediately, or buy online and collect in person. Using either a postcode or address that the customer enters into a standard desktop interface, or using the GPS information available to a mobile device, the system calculates the nearest stores which have the requested items available, and then directs the customer to these stores for purchase and collection based upon item availability.

Features

Geocoding

Uses third party services for translating address data to GPS coordinates, and finding routes between two points.

Flexibility

Follows current standards for geographical data presentation. Supports GoogleMaps API out of the box, but works with other similar services. If a service has limitations on the amount of traffic or request frequency, Store Locator is configurable to avoid exceeding this limit.

Maplet Display

Follows KML standards to keep map information well structured and easy to display

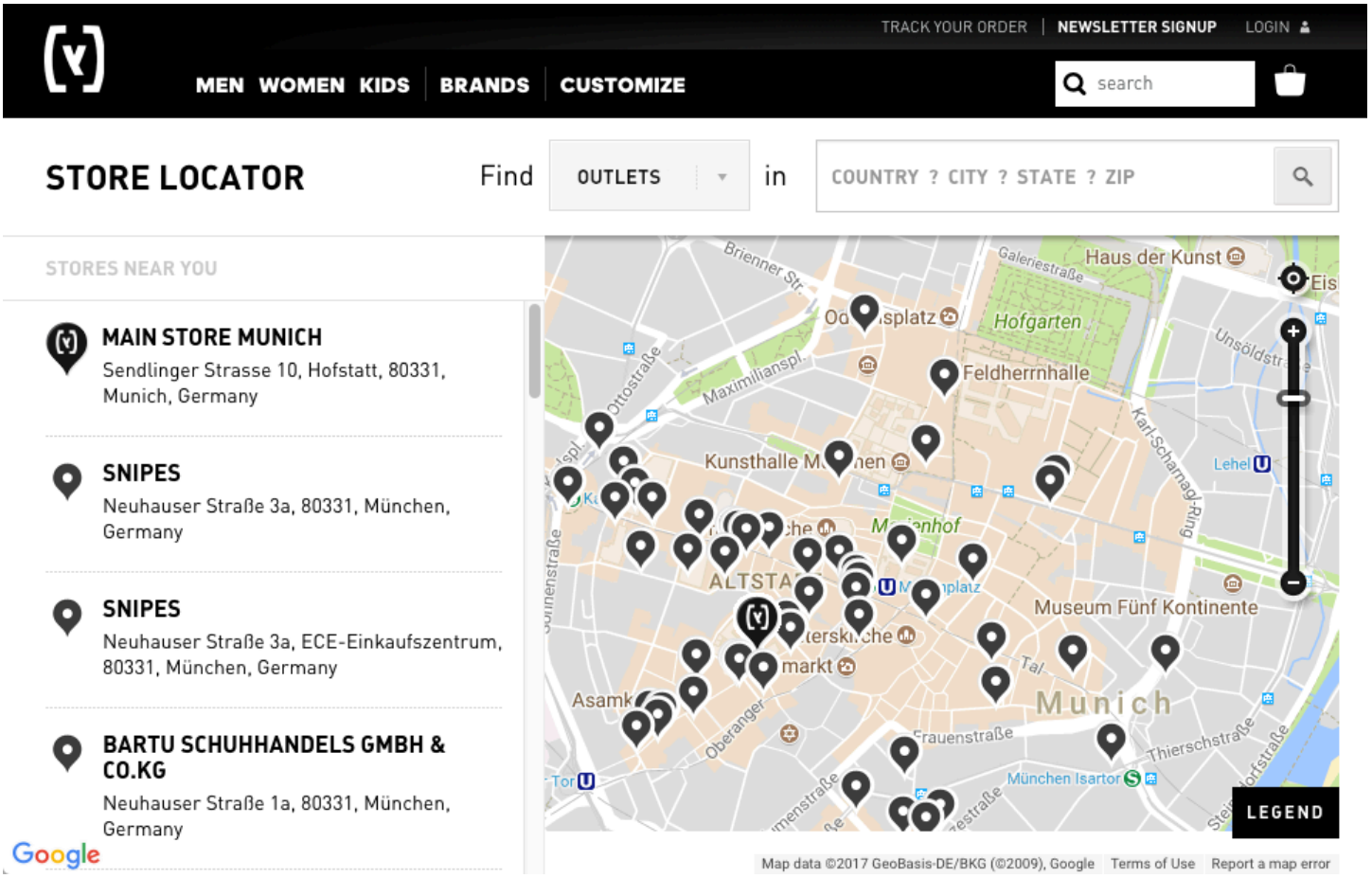
Operations Manager Input

Takes input from catalog store maintainers, who can enter and update a precise address information for a store. This input can be provided in Backoffice or using a dedicated service. A dedicated cron job on the server checks for any new or updated locations in order to update the GPS coordinates.

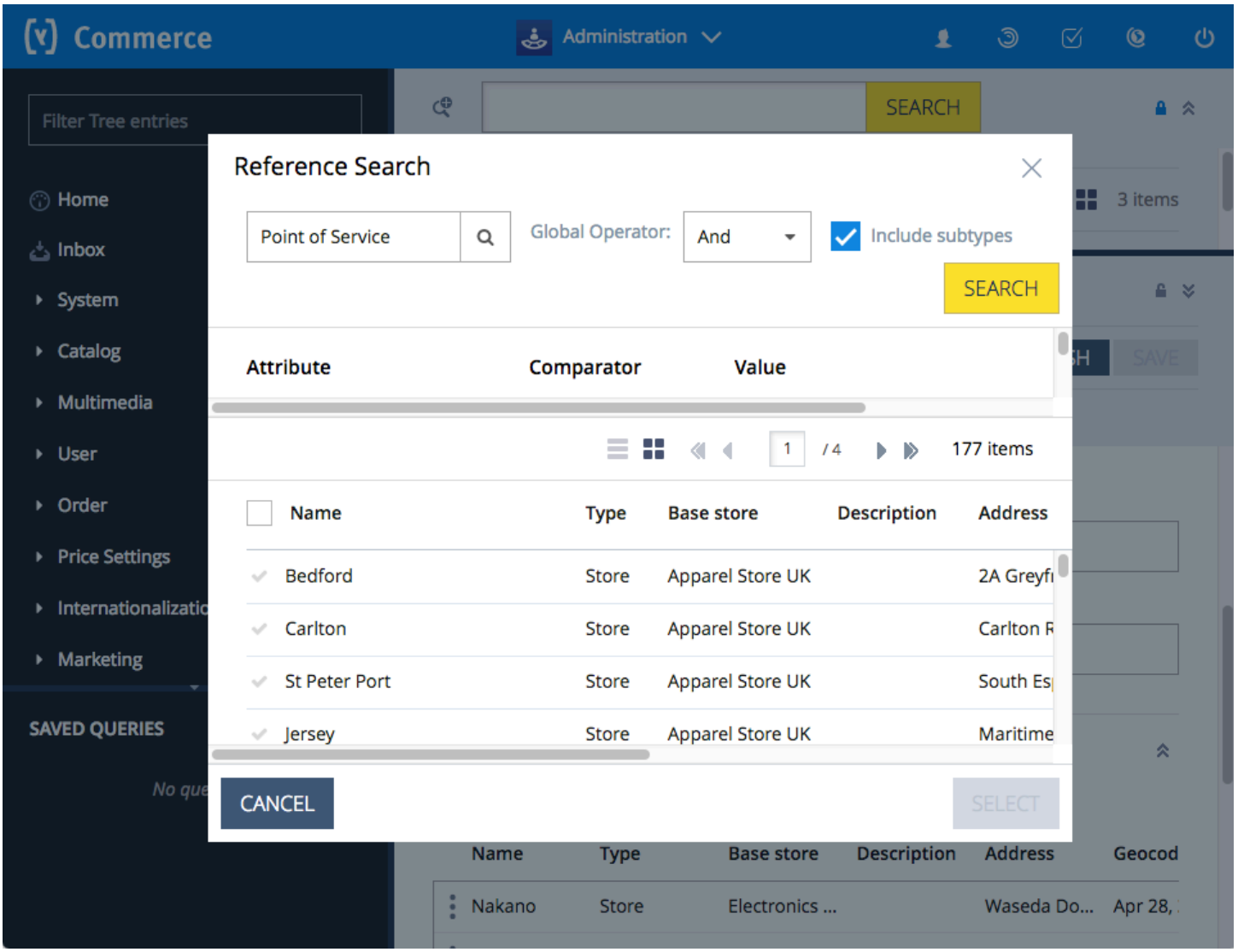
Dependencies

There are no specific dependencies for using Store Locator.

Screenshots



Store Locator End User Perspective



Store Locator Operations Manager Perspective

Related Information

[Store Locator Implementation]
[Using Store Locator Service]
[Google as Geo Service Provider]

Store Locator Implementation

The Store Locator includes data items types and dedicated services as part of the implementation of the basecommerce extension. It also has several interfaces that allow you to produce map objects, for example, placemarks, routes, styles, and so on, that you can easily transform into Keyhole Markup Language (KML) compatible format.

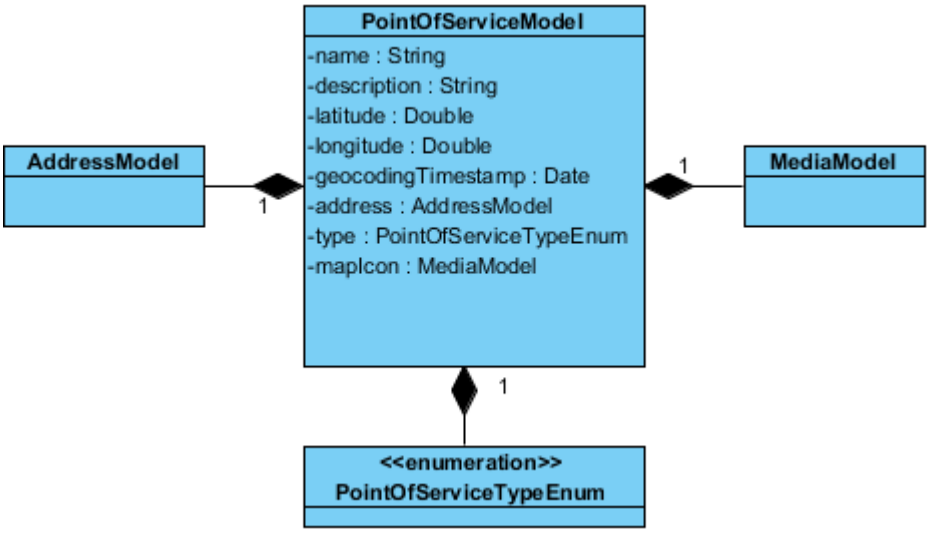
Data Structure

Data structure introduced by the Store Locator involves new item types. The internal services, however, make also use of many helpful POJO classes that all contribute to the provided functionality.

PointOfService Item

The Store Locator introduces new item type, **PointOfService**. The purpose of this item is to equip address data of **AddressModel** data type with geographic position. You can attach the **PointOfService** to some other items to make them **geographically**

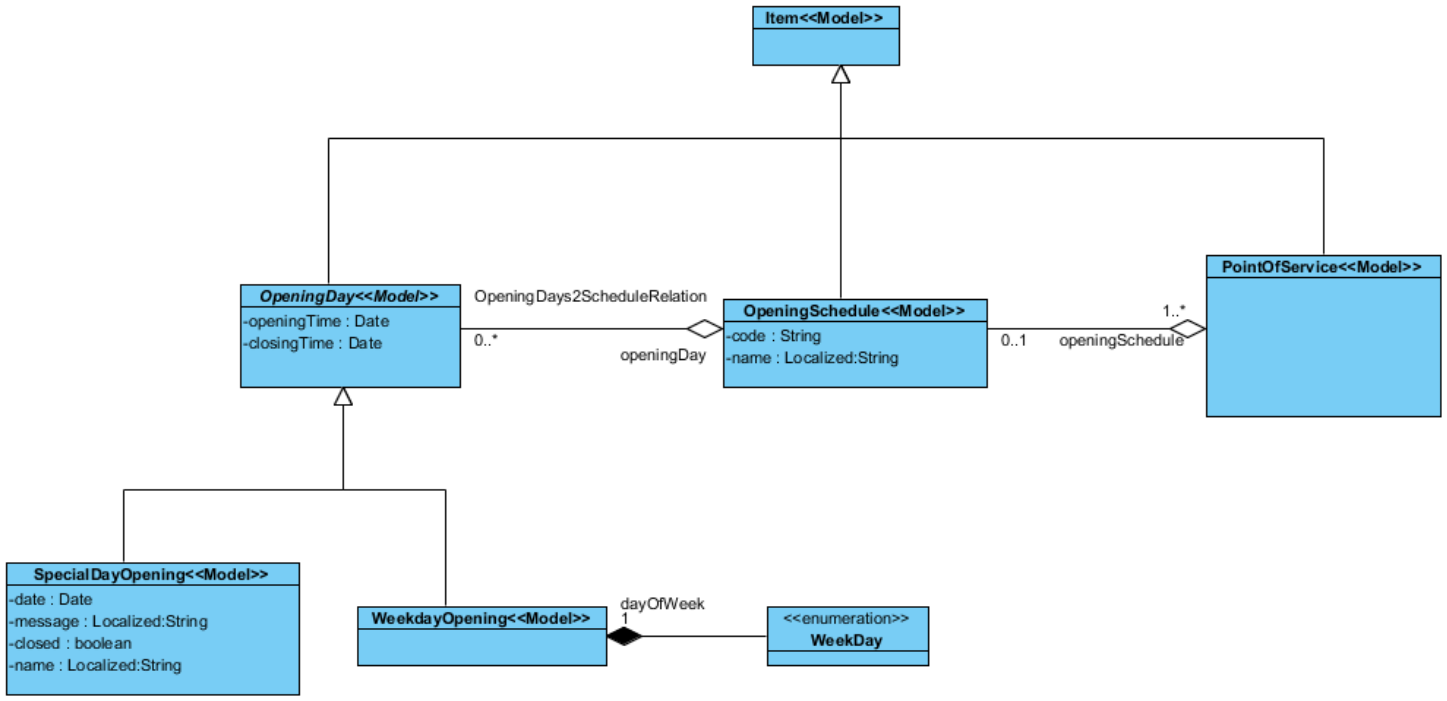
searchable. Moreover, thanks to **PointOfServiceModel**, you can represent the items on maps.



OpeningSchedule Item

Store Locator brings the **OpeningSchedule** item type. The purpose of this item is to provide a **PointOfService** with its opening and closing information.

You can define which of weekdays are opened and what are the opening and closing hours. You can also define what special date the point of service (POS) is opened or closed at, for example for national holidays.

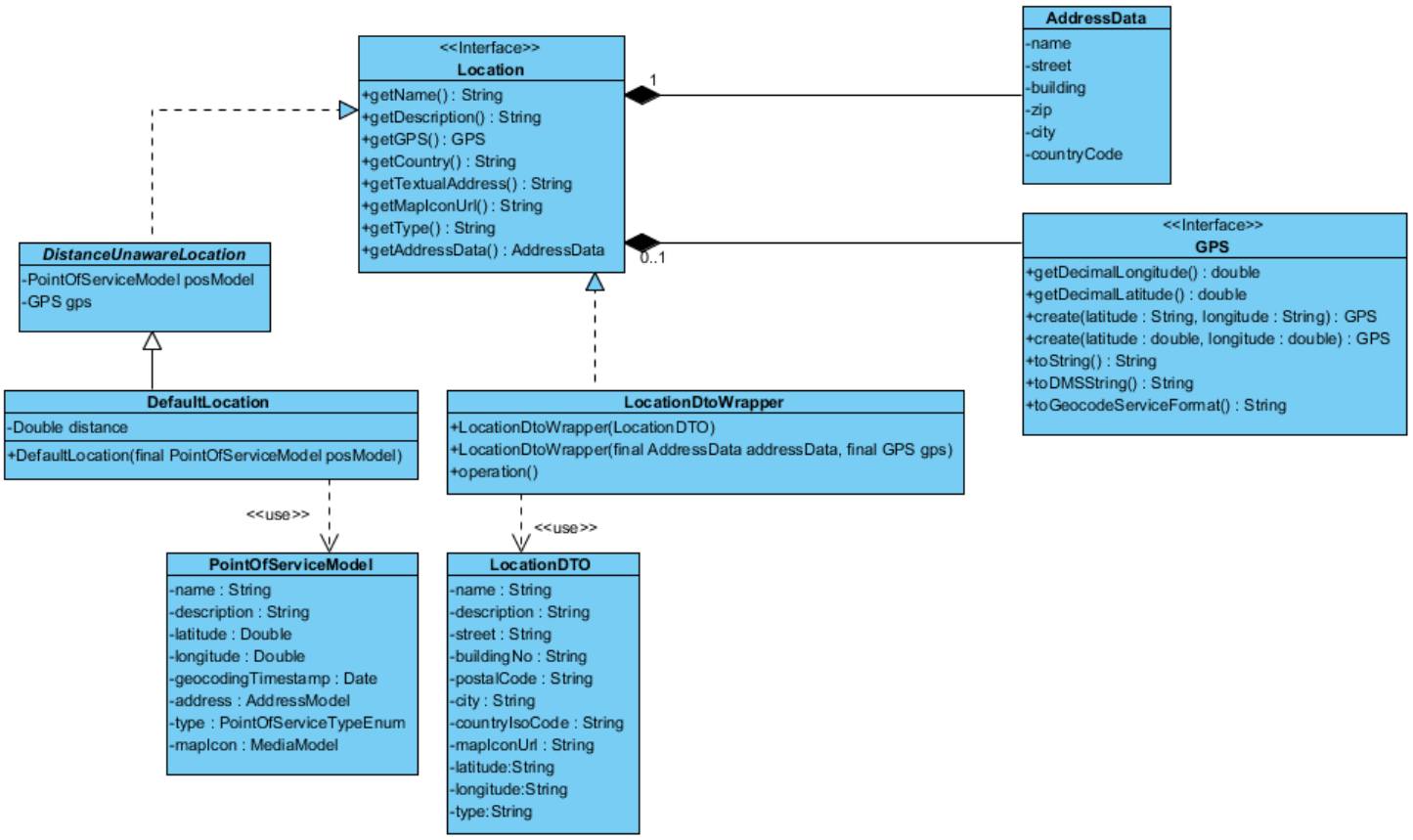


POJO Classes Dedicated for Services

Considering the Store Locator data types you can distinguish between the main data types and additional data types. Furthermore, there is also a whole structure of objects relevant to the KML Structure.

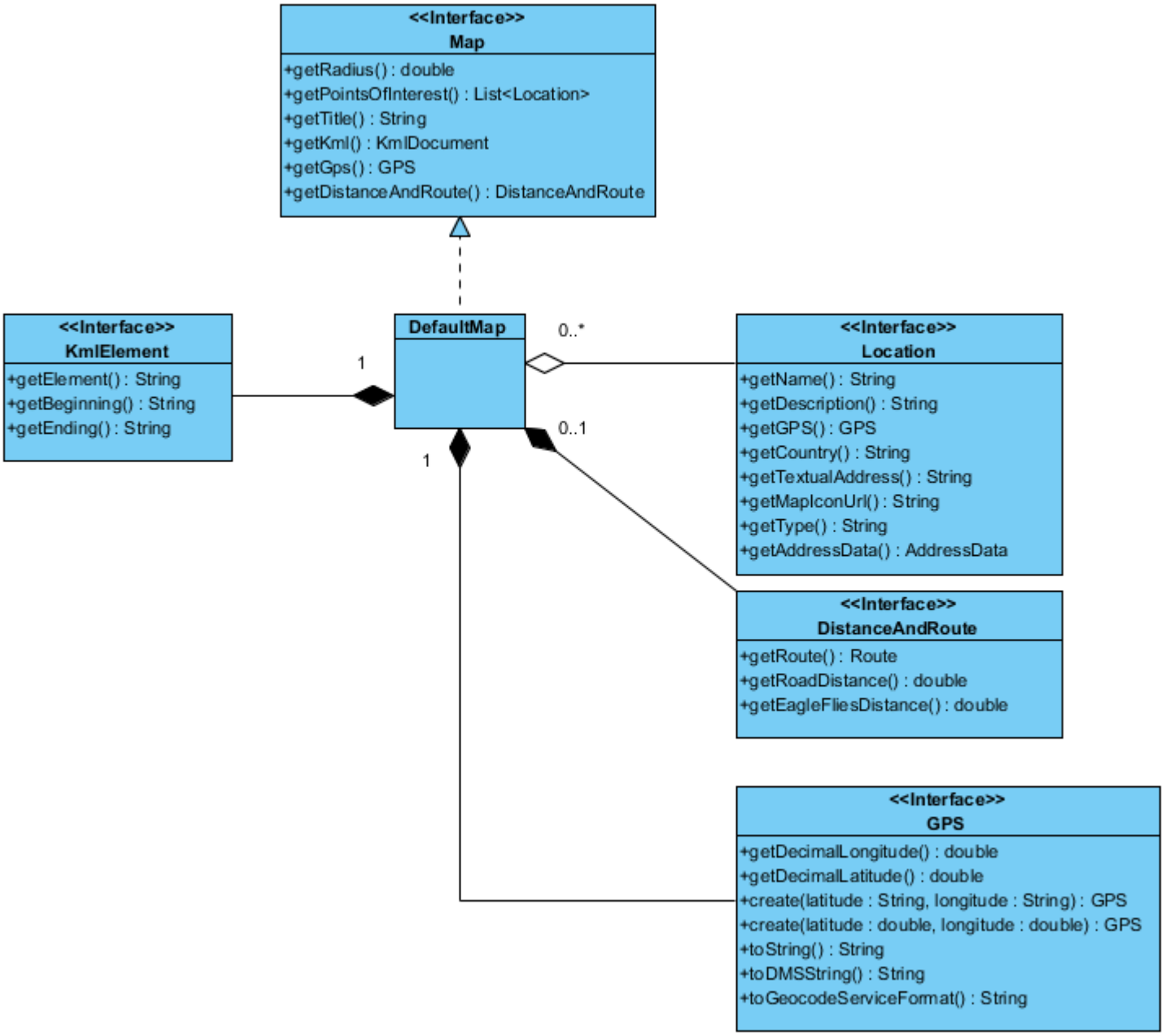
- **Location**: Interface represents a location meaningful from the business perspective. You can bind it to any item like store, warehouse, and so on. The **Location** interface has the following implementations:
    - **DefaultLocation**: Implements **Location** based on the **PointOfServiceModel** and hence allows to operate on locations defined in the SAP Commerce Platform.
    - **LocationDTOWrapper**: Provides the means to integrate with other systems using **LocationDTOs**.

- **GPS**: Represents geographic location or, in other words, a valid geographical position. The **Location** may have a GPS coordinates, in which case we say it is geocoded. The **GPS** interface has one default implementation: **DefaultGPS**.



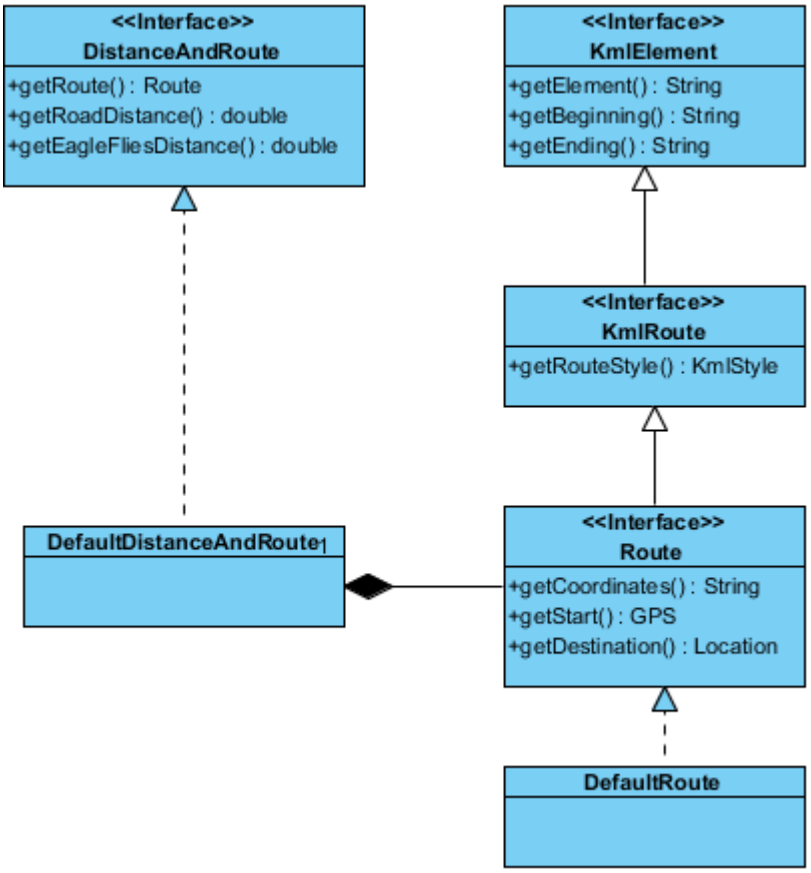
Additional types useful for the front-ends: **Map**, **DistanceAndRoute**, **Route**.

- **Map**: Represents a single map that is a product of the MapService. It may contain a collection of Point of Interest (POI), as well as the route information. You can display maps using their KML elements. It consists of:
    - Center GPS
    - Radius
    - Title



- **DistanceAndRoute**: Represents a set of information concerning a route. It is a product of the RouteService. It consists of a distance values: road distance, straight line distance.

  Moreover, it contains a set of coordinates that make all together a route on the map. Those coordinates are wrapped inside the **Route** object that implements KML element, which enables you to display it on the map.



Dedicated Services

This section contains a technical overview on internal interfaces.

Location Service

- Returns the nearest locations of POI to the given coordinates, in the number adequate to the user definition. Use when you have got defined coordinates and need only sorted locations nearby, without any map.

```
List<Location> getLocationsForPoint(GPS gps, int limitLocationsCount, BaseStoreModel baseStor
```

- Returns nearby locations of POI in the place defined by search term, which can be either a postal code or a town name and country. Basically it combines two other methods that retrieve locations for town name and postal code.

```
List<Location> getLocationsForSearch(String searchTerm, String countryCode, int limitLocation
```

- Returns the nearest locations of POI to the place defined by postal code and country. Number of returned locations is adequate to the user definition. Use when you do not need any map, only list of sorted locations.

```
List<Location> getLocationsForPostcode(String postalCode, String countryCode, int limitLocati
```

- Returns the nearest locations of POI to the place defined by a city name. Number of returned locations is adequate to the user definition. Use when you do not need any map, only list of sorted locations.

```
List<Location> getLocationsForTown(String town, int limitLocationsCount, BaseStoreModel baseS
```

- Returns the list of sorted items of **DistanceAwareLocation**. Sorting order is ascending - the closest locations come first.

```
List<Location> getSortedLocationsNearby(final GPS gps, final double distance, final BaseStore
```

Location Map Service

- Returns nearby locations for given postal code or town name and creates **Map** object with found data:

```
Map getMapOfLocations(String, String, int, BaseStoreModel) throws LocationMapServiceException
```

- Returns the nearest POIs to the place defined by postal code and country, and creates the **Map** object with given data. Number of returned locations is adequate to the user definition. Use when map representation is required:

```
Map getMapOfLocationsForPostcode(String, String, int, BaseStoreModel) throws LocationMapServi
```

- Returns the POIs nearest to the place defined by a city name, and creates **Map** object with given data. Number of returned locations is adequate to the user definition. Use when map representation is required:

```
Map getMapOfLocationsForTown(String, String, int, BaseStoreModel) throws LocationMapServiceEx
```

- Returns a **Map**, basing on number of points of interest defined by a user and the proper radius:

```
Map getMapOfLocations(GPS, int, BaseStoreModel) throws LocationMapServiceException
```

GeoWebServiceWrapper

This service hides all the communication to the third party geocoding services, for example Google Maps. It should be used for getting geographical data about addresses as well as getting information on travel directions.

These service interfaces use:

- **Location**
- **GPS**
- **AddressData**

These service interfaces return:

- **GPS**
- **DistanceAndRoute**

Available calls are as follows:

- Translates the **Location** into **GPS** using third party geocoding service:

```
GPS geocodeAddress(final Location address) throws GeoServiceWrapperException;
```

- Translates the address from **AddressData** DTO into **GPS** using third party geocoding service:

```
GPS geocodeAddress(final AddressData address) throws GeoServiceWrapperException;
```

- Gets the distance and route between two **Locations**:

```
DistanceAndRoute getDistanceAndRoute(final Location start, final Location dest) throws GeoSer
```

- Gets distance and route between start **GPS** position and destination **Location**:

```
DistanceAndRoute getDistanceAndRoute(final GPS start, final Location dest) throws GeoServiceW
```

- Provides the address in a formatted String. The format is compliant with the service of choice:

```
String formatAddress(final Location address) throws GeoServiceWrapperException;
```

Geocoding Based on GoogleMaps: GoogleMapsServiceWrapper

When a geocoding request is issued, a new HTTP **GET** request is composed. Such request is sent to the Google service. The received response is then parsed into the **GPS** or **DistanceAndRoute** data structure.

```
(g) No Use of Content without a Google Map. You must not use or display the Content without a corre
Link : https://developers.google.com/maps/terms?hl=en
```

In other words, Google requires you to show a map when you use their service to geocode a location. However, you are permitted to "cache" known locations (such as store locations), as long as you do not store them for more than 30 days. This "caching" makes the use of the ***geocodingCronJob*** permitted.

For example, in Accelerator, in the following scenarios, we geocode a location, but we do not show a map. As this goes against the legal requirement, we must instead use a mock :

- Geocoding a location as part of facetted search
- Geocoding a location in the pickup-in-store modal so as to find a store that stocks your product

For this reason, the `basecommerce` extension is delivered with a mock implementation out-of-the box. This implementation does not trigger any external webservice call; instead, it returns pre-defined geolocations. These geolocations are unique for each country. For example, a geolocation in the United States will always return `40.7127, 74.0059`, or in Germany, `52.514524, 13.35029`. If no country can be found on the submitted address, or if the geocode is not configured for the submitted country, then a default geolocation is returned.

MapService

These service interfaces use special types:

- **Location**
- **GPS**

These service interfaces return: **Map**.

Available calls are as follows:

- Returns the titled **Map** object centered on the **GPS** position. Please mind, that this method call would respond with a **Map** instance with codenullcode KML field. This is beacuse no KML representation is needed for a map for which we know only center and default radius. KML field is populated when there are some markups to be displayed on the map, such as POIs, route, and so on:

  ```
  Map getMap(GPS gps, String title) throws MapServiceException;
  ```

- Returns the titled **Map** object centered on the **GPS** position with a given radius, containing marker points for each point of interest given as input.

  ```
  Map getMap(GPS gps, double radius, List<Location> poi, String title) throws MapServiceExcepti
  ```

- Returns the titled **Map** object centered on the **GPS** position with a given radius, containing marker points for each point of interest given as input. Map also contains a road route between map center (**GPS**) and a given destination **Location** (**routeTo**):

  ```
  Map getMap(final GPS gps, final double radius, final List<Location> poi, final String title,
  ```

- Returns the titled **Map** centered around the **GPS** location with a given radius. As no additional placemarks are required, returned map object has null KML member, **Map.getKml()**:

```
Map getMap(GPS gps, String title, double radius) throws MapServiceException;
```

- Returns the map bounds of the given map. Calculations are perfomed basing on the rectangular coordinates, not just a radius.

```
MapBounds getMapBoundsForMap(final Map map) throws GeoLocatorException;
```

RouteService

This services provides method for getting route and distance information between two points on the map.

These service interfaces use types:

- **Location**
- **GPS**

These service interfaces return **DistanceAndRoute**.

Available calls are as follows:

- Returns a **DistanceAndRoute** object that connects starting and destination **Location**:

```
DistanceAndRoute getDistanceAndRoute(Location start, Location dest) throws RouteServiceExcept
```

- Returns a **DistanceAndRoute** object that connects starting **GPS** position and destination **Location**:

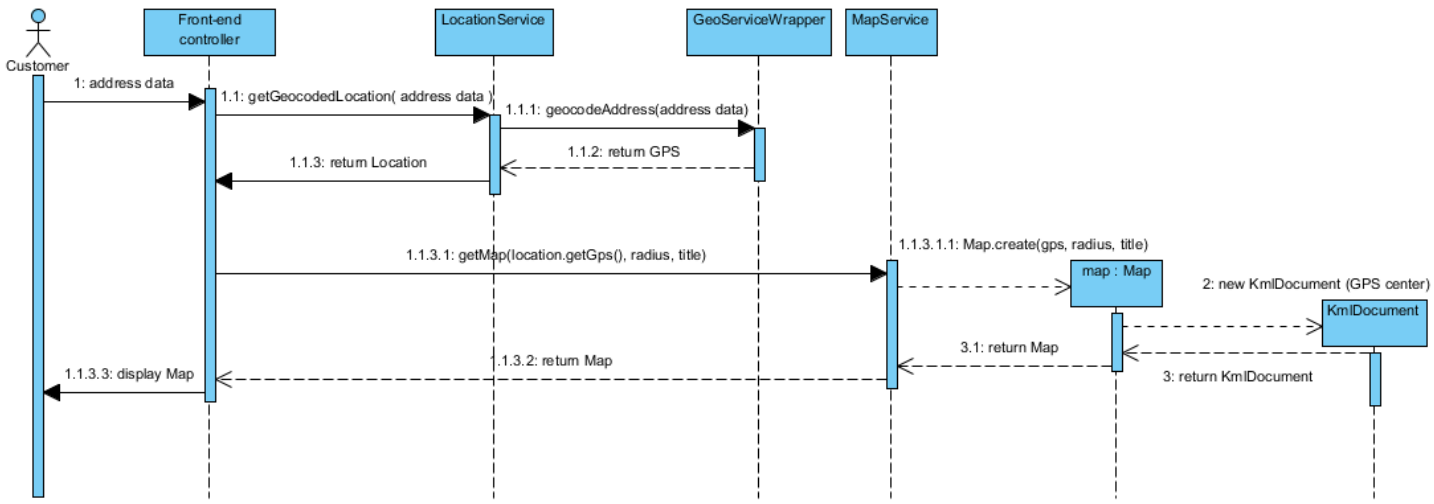
```
DistanceAndRoute getDistanceAndRoute(GPS start, Location dest) throws RouteServiceException;
```

Store Locator Use Cases

Diagrams presented in the sections below show how the Store Locator works and how its components interact with each other.

Getting a Map with Customer Location

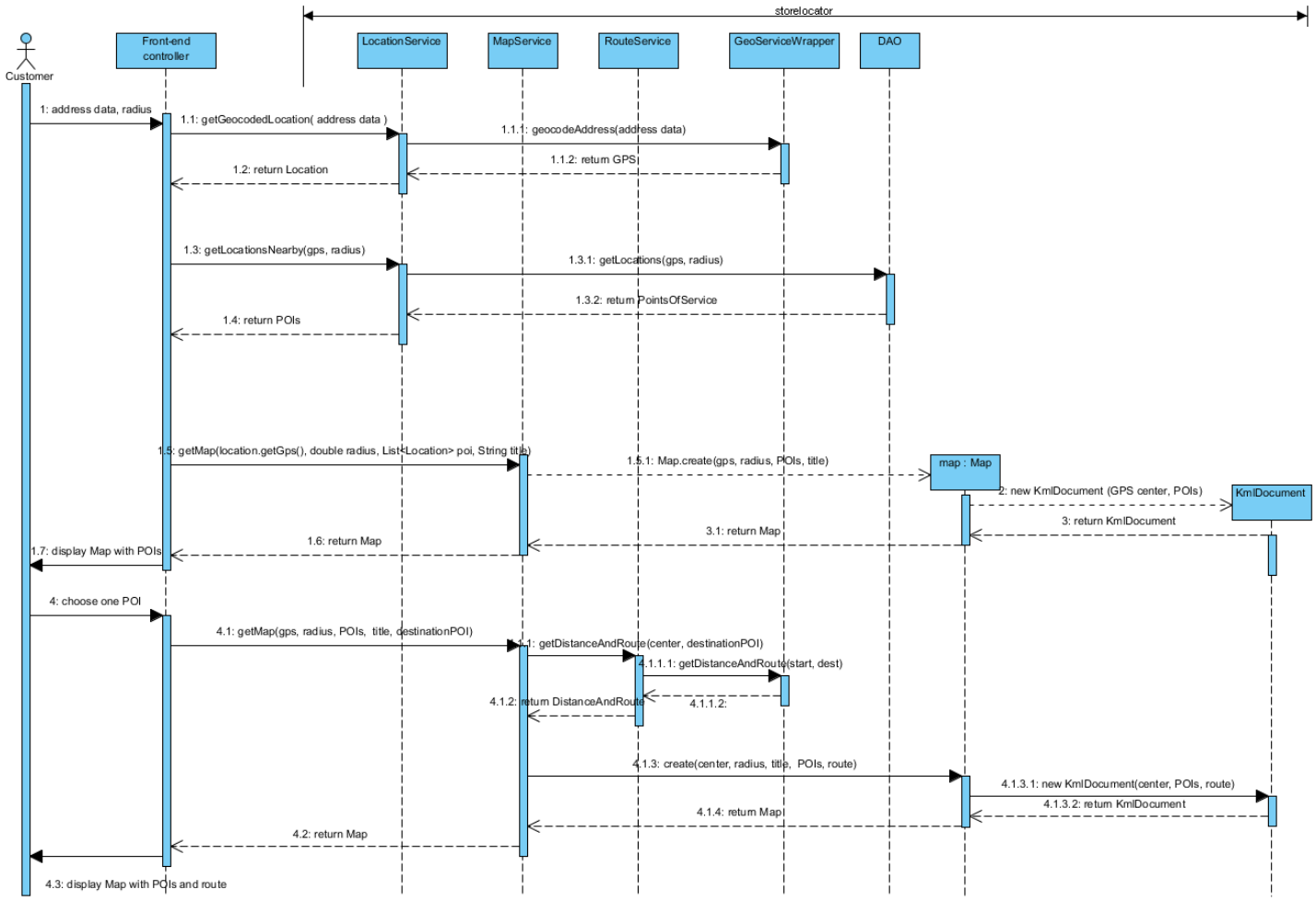
A front-end controller may request a map from **storelocator** extension based on the customer address data:



The third party geocoding service has its limitations, and geocoding addresses for each customer visiting the site may be not the best approach. If Google Maps is chosen as a service provider, then follow the Google Maps Limitations link for the ideas how to replace this operation with the client side geocoding.

Getting a Map with Stores and a Route to the POI of Choice

If the customer specifies a radius for looking up neighboring stores or POIs, the controller may request to get a map with all the objects included:

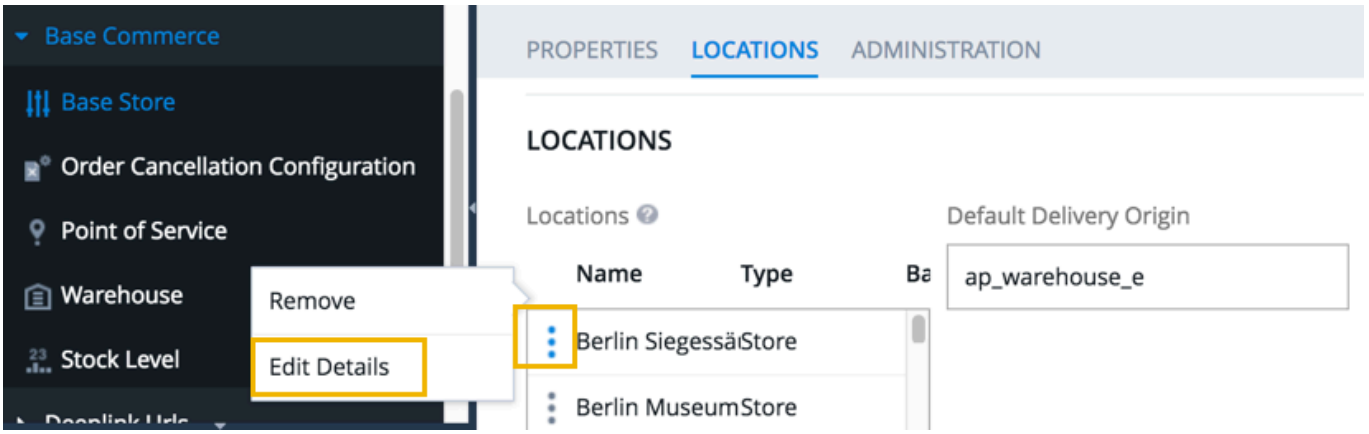


Points of Service Maintenance

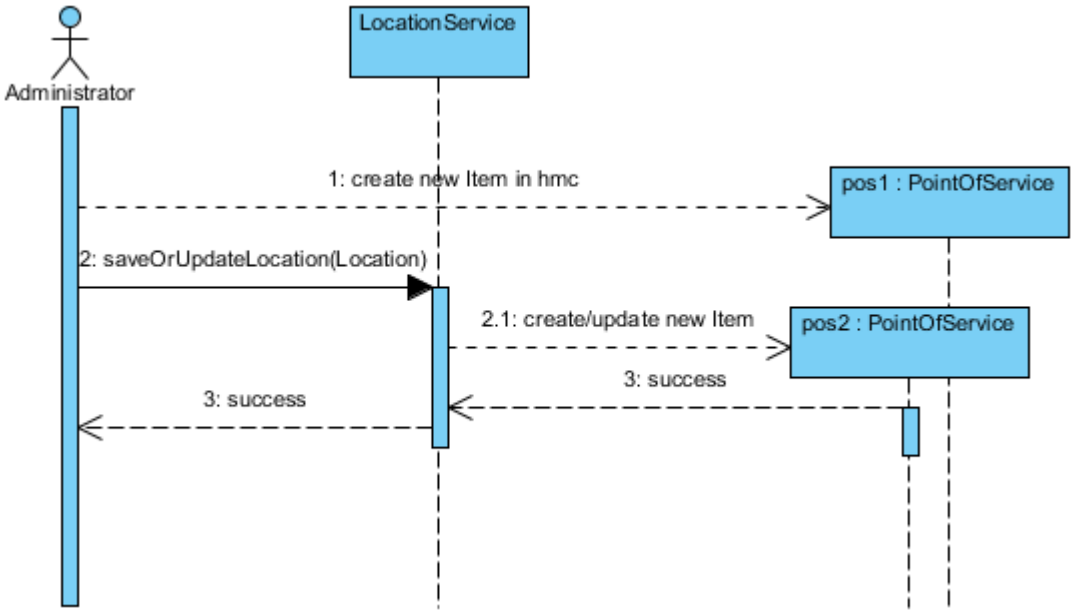
Before you can find any location on the map, the store administrators need to enter the location data into the database. The store administrators task is to maintain a valid list of POS. They can also give the POS a description and icon to determine its appearance on the map.

Administrators can add new locations using the Backoffice Administration Cockpit. To find POS available for a base store, perform following steps:

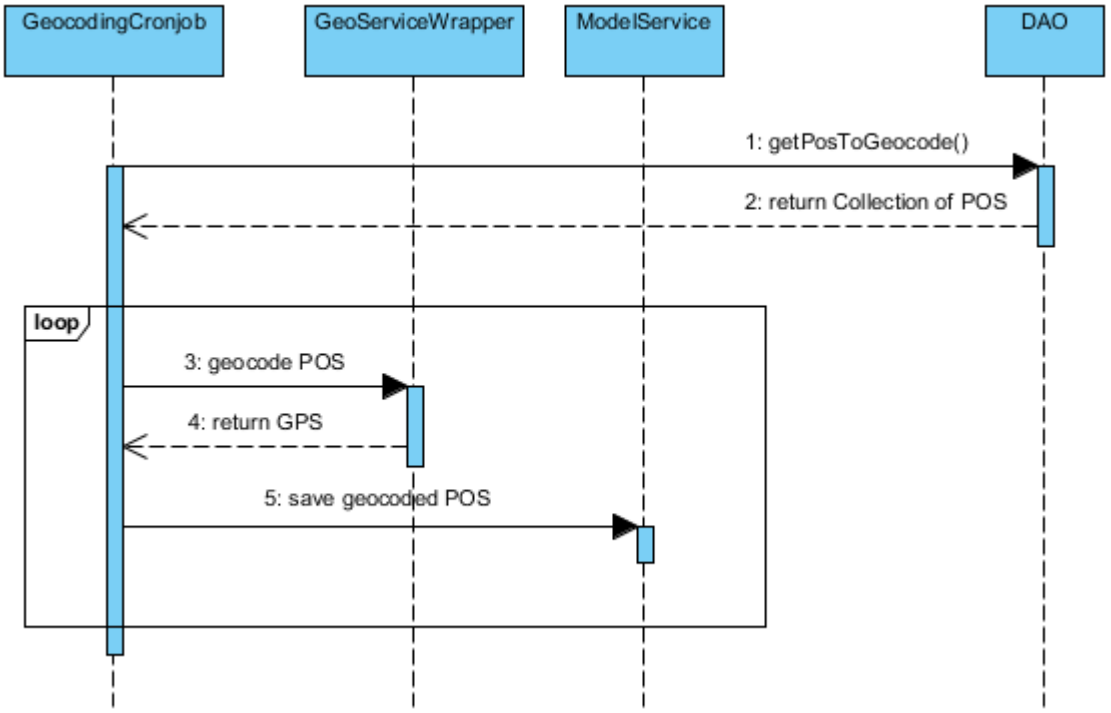
1. Log on to the Backoffice Administration Cockpit.
2. Navigate to the **Base Commerce** > **Base Store**.
3. Select the base store from the list provided.
4. Go to the **Locations** tab.
5. Select the POS that you want to edit from the list.
6. Click the corresponding action button and select **Edit Details**.



It is also possible to add new locations using the **LocationService**. In this case, maintenance of the Store Locator is possible via web services.



The **storelocator** extension is initialized together with the **GeocodingCronJob** that collects all new or modified POS entries and requests a geocoding action for each of them. This task is triggered as a cronjob and could be given the parameters to determine the triggering frequency and POS entries batch size. Adjusting these parameters should be done in accordance to potential limitations of the third party geocoding services, for example, Google Maps API limitations.

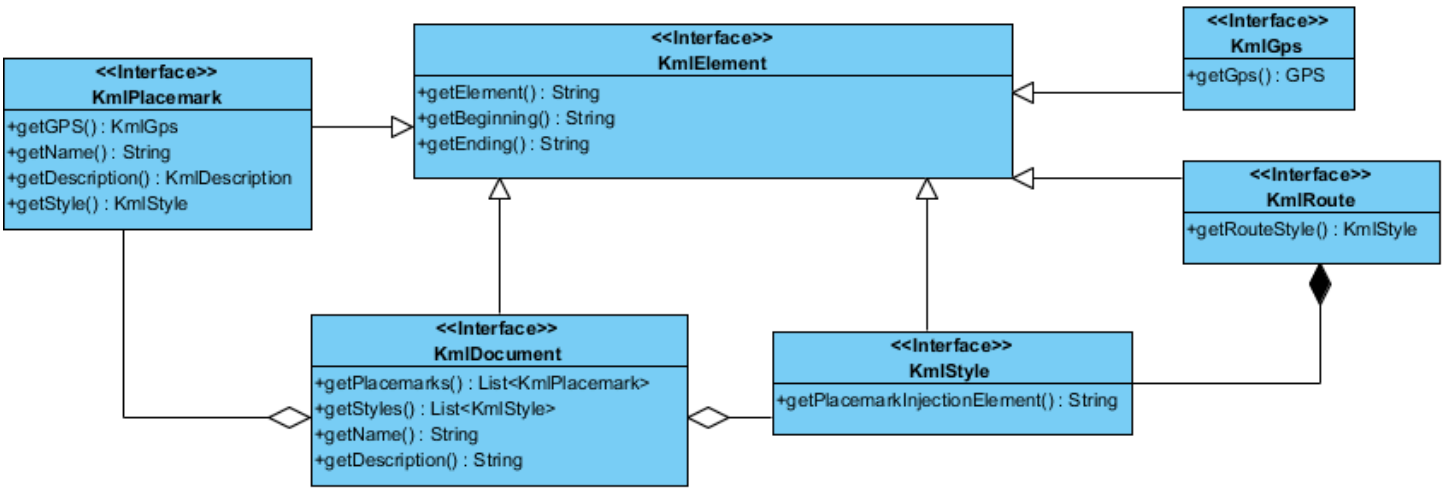


KML Structure

KML structure is XML-compliant. It is an open standard that is useful for modeling points, polygons, and so on. It is adopted in the Google Maps in order to provide structured way to represent maps and all contained objects on maps.

The Store Locator has several interfaces that allow to produce map objects, for example, placemarks, routes, styles, and so on, that you can easily transform into KML compatible format. They all realize **KmlElement** interface and therefore can be given to front-end as displayable map elements.

The main object is **KmlDocument**. It contains the child elements: **content** and **styles** of the map. The **KmlDocument.getElement()** method produces the whole map content that could be parsed into a Google mapplet on the front-end.



Related Information

acceleratorservices Extension
http://code.google.com/apis/maps/index.html

Using Store Locator Service

Follow the instructions in using the Store Locator functionality. Understand the most common use cases of the internal services and how their arguments are described. Learn how to add new store locations, how the storelocator finds the added locations, and how it provides the maps with the routes.

Overview

The storelocator provides three main services. The table below shows details about each of these services.

Service Name	Purpose	Input	Output
LocationService	CRUD operation on locations	***Location***	boolean
	Getting locations from neighborhood	***GPS***, radius	Collection ***Location***
	Getting temporary location	address data (***String***)	***Location***
MapService	Getting a structured map instance that could be displayed on the fron-end	***GPS***, ***Location***	***Map***
RouteService	Getting information on route and distance from staring point to destination	***Location***, ***GPS***	***DistanceAndRoute***

Adding New Store Locations

Adding Points Of Service in the Backoffice Administration Cockpit

Use Backoffice Administration Cockpit to add new location entries to Base Stores.

1. In the Backoffice Administration Cockpit, select Base Commerce > Point of Service.
2. Click the Add button to open the Create New Point of Service window.

See Also

Store Locator Implementation

When a new item with address data, name, description and even a map icon is saved, user may manually trigger geocoding of this new location. This task can also be performed by a dedicated cron job that comes together with the storelocator.

Adding Locations Using locationService

The following code sample shows you how the locations are added by means of ***locationService***.

```
LocationDTO locationDto = new LocationDTO("Nymphenburger strasse", "86", "80636", "Muenchen","DE");
locationDto.setName("hybris");
locationDto.setDescription("hybris main location");
//if gps position is known, latitude and longitude can be set.
//Location entry will be considered as geocoded
locationDto.setLatitude("48.149772");
locationDto.setLongitude("11.54667");
Location location = new LocationDtoWrapper(locationDto);
try
{
```

```
        locationService.saveOrUpdateLocation(location);
    }
    catch (LocationServiceException e)
    {
        // handler.
    }
```

Adjusting Geocoding Cron Job

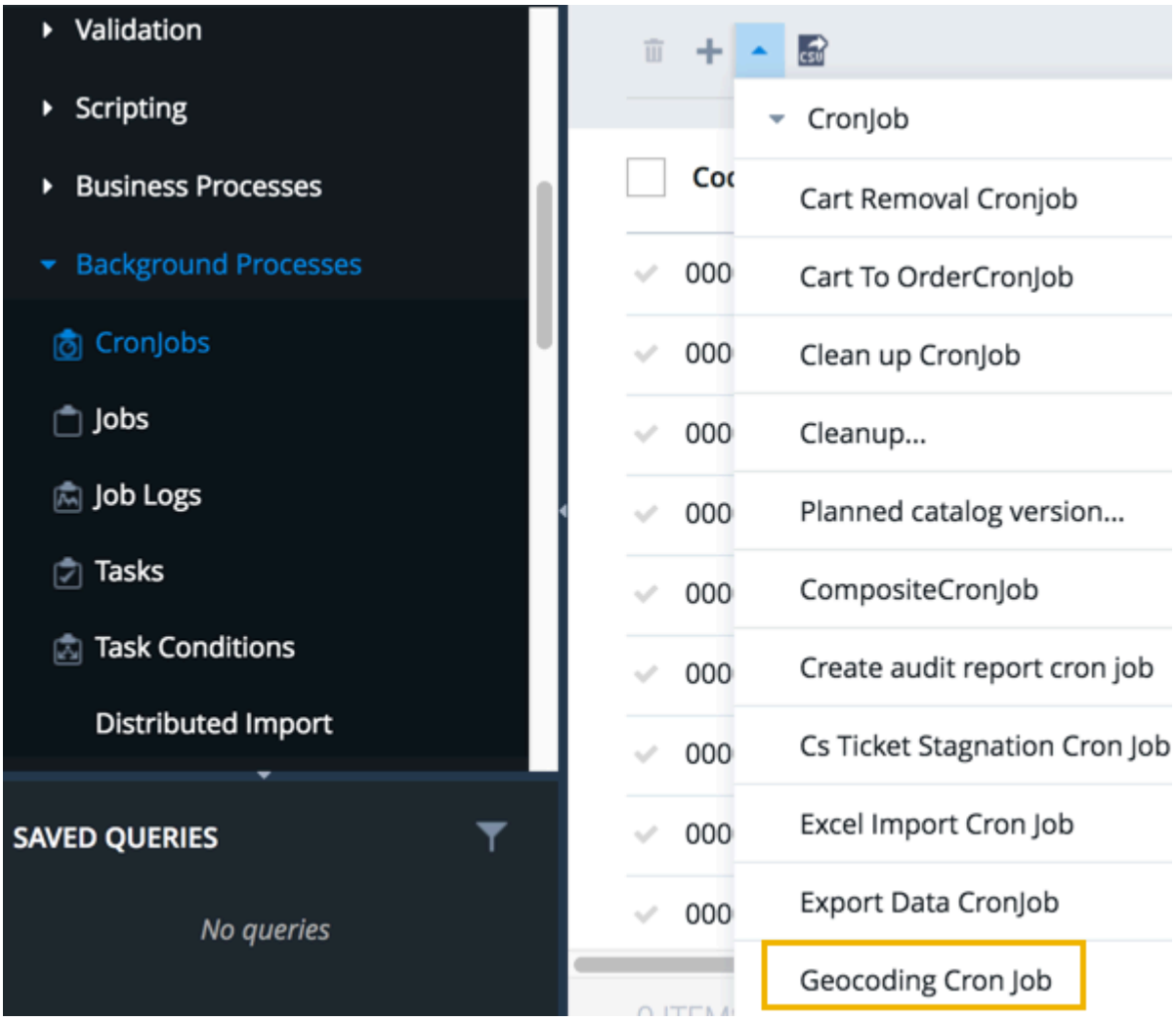
The extension is initialized with a cronjob task responsible for geocoding. It collects new entries or modified ones and issues a geocoding request for each of them.To change the initial parameters of the cronjob, you can modify the corresponding parameters in the `project.properties` file.

Parameter Name	Meaning	project.properties Key	Default Value	Unit
trigger interval	Time interval between two consecutive cron job triggers	geocode.cronjob.triggerinterval	15	minutes
internal delay	Time interval between two consecutive geocoding requests (to avoid querying 3rd part service to frequently)	geocode.cronjob.internaldelay	3	seconds
batchsize	How many entries are taken for one cron job execution	geocode.cronjob.batchsize	100	

See Also

The Cronjob Service

Users can come up with their own cron job for geocoding purposes, however, after system reinitialization. the default cron job is restored. To start a custom cron job, select **Background Processes** > **CronJobs** in Backoffice Administration Cockpit. Select **Create CronJob** > **Geocoding Cron Job** to display the Create New Geocoding Cron Job window.



Set the cron job scheduling to get the **trigger interval** of your choice. Additionally, set the **batch size** and the **internal delay** on the editor. Select ***geocodeAddresJob*** for the option **Job**.

Finding Stores

After you have created POS entries and geocoded them, either manually or via geocoding cron job, it is possible to locate them.

The main task of the storelocator is to find stores in the vicinity of some location and to display them on the map. To have a full knowledge on how this is done, read Store Locator.

Getting the locations nearby is performed by ***LocationService***, while getting a corresponding **Map** instance is done by ***MapService***.

```
...
private LocationService locationService;
private MapService mapService;
locationService
...

//InputForm
AddressForm addressForm = ...
//create Location from the input data
Location myLocation = locationService.getLocation(addressForm.getStreet(), addressForm.getBuildingN
//2 kilometers radius
double radius = 2
List<Location> pois = locationService.getLocationsNearby(myLocation.getGPS(), radius);
...
...
//get Map
Map map = mapService.getMap(myLocation.getGps(), radius, pois, "My map");
```

Finding Routes

MapService already covers getting a map instance with a route marked on it. It can be done by calling an overloaded **getMap()** method which has a ***routeTo*** argument. So in context of the previous example, the following one will provide the map with the route.

```
...
//Find route to the first point of interest on the map.
Location routeTo = poi.get(0);
Map mapWithRoute = mapService.getMap(myLocation.getGPS(), radius, poi, title, routeTo);
...
```

In order to find some custom routes or travel directions, you can use the ***RouteService*** to obtain the route coordinates, road and straight line distances.

```
...
private RouteService routeService
...

Location start= ...
Location destination = ...

DistanceAndRoute result = routeService.getDistanceAndRoute(start, destination);
//getDistanceAndRoute is overloaded, so the following produces the same.
DistanceAndRoute result1 = routeService.getDistanceAndRoute(start.getGps(), destination);

KmlRoute coordinatePath = result.getRoute();
double roadDistance = result.getRoadDistance();
double straightLine = result.getEagleFliesDistance();
//
```

Google as Geo Service Provider

The Store Locator uses the third party geocoding services to provide its full functionality. The main task for the Store locator is to translate the given address to the GPS coordinates. Although the module can be integrated with any of the third party geocoding services, the current document puts focus on the Google Maps, which is integrated with Store locator by default.

Overview

To equip the Store locator with geocoding service, the developer needs to implement an adapter customized for the chosen service. In case of the Google Maps service, the store locator already contains a built-in adapter:

```
de.hybris.platform.storelocator.impl.GoogleMapsServiceWrapper
```

The Google Maps is an on-line service that allows to geolocate addresses, commonly known places, find traveling directions, etc. The functionality of the service is rapidly growing, showing some potential in community services as well. Using the service in commercial domain is possible only after signing up for the Google Maps API key.

Client Side API

Google Maps service provides a javascript API that may be used to construct embedded Mapplets that utilize all features from both Google Maps and Google Earth. In order to use the API a significant part of the business logic must be delegated to the client side which makes it hard to keep the service-oriented architecture.

Client Side Geocoding

Before customers can locate any stores, they must be self-located on the Mapplet. Here, the javascript API comes in handy with the **GClientGeocoder** object, because it allows to send geocoding request from the client side.

HTTP Requests

On the other hand, it is possible to construct an HTTP request and GET the map object in return. The format of the map depends on the request parameter. HTTP requests are used in the GoogleMaps geocoding adapter in the storelocator.

Web Service Geocoding

To make the HTTP request, the following format applies:

```
http://maps.googleapis.com/maps/api/geocode/OUTPUT_FORMAT?address=ADDRESS_DATA&sensor=true
|false
```

URL encoded parameters used:

- **ADDRESS DATA**: This should be a human readable data encoded using UTF-8 standard.
- **OUTPUT FORMAT**: One of the following: json, xml.

For example:

```
http://maps.googleapis.com/maps/api/geocode/xml?
address=1600+Amphitheatre+Parkway,+Mountain+View,+CA&sensor=true
```

ℹ Note

To make the example request directly from the hybris dev-net page is impossible without a valid API key. If you do not have the key, copy the link above and use it in a separate browser window instead.

The example request should produce the following response:

```
<GeocodeResponse>
                                        <status>OK</status>
                                        <result>
                                        <type>street_address</type>
                                        <formatted_address>
                                        1600 Amphitheatre Parkway, Mountain View, CA 94043, USA
                                        </formatted_address>
                                        <address_component>
                                        <long_name>1600</long_name>
                                        <short_name>1600</short_name>
                                        <type>street_number</type>
                                        </address_component>
                                        <address_component>
                                        <long_name>Amphitheatre Parkway</long_name>
                                        <short_name>Amphitheatre Pkwy</short_name>
                                        <type>route</type>
                                        </address_component>
                                        <address_component>
                                        <long_name>Mountain View</long_name>
                                        <short_name>Mountain View</short_name>
                                        <type>locality</type>
                                        <type>political</type>
                                        </address_component>
                                        <address_component>
                                        <long_name>Santa Clara</long_name>
                                        <short_name>Santa Clara</short_name>
                                        <type>administrative_area_level_2</type>
                                        <type>political</type>
                                        </address_component>
                                        <address_component>
```

```
<long_name>California</long_name>
<short_name>CA</short_name>
<type>administrative_area_level_1</type>
<type>political</type>
</address_component>
<address_component>
<long_name>United States</long_name>
<short_name>US</short_name>
<type>country</type>
<type>political</type>
</address_component>
<address_component>
<long_name>94043</long_name>
<short_name>94043</short_name>
<type>postal_code</type>
</address_component>
<geometry>
<location>
<lat>37.4221913</lat>
<lng>-122.0845853</lng>
</location>
<location_type>ROOFTOP</location_type>
<viewport>
<southwest>
<lat>37.4208423</lat>
<lng>-122.0859343</lng>
</southwest>
<northeast>
<lat>37.4235403</lat>
<lng>-122.0832363</lng>
</northeast>
</viewport>
</geometry>
</result>
</GeocodeResponse>
```

Getting Directions

Store locator can geocode addresses and find routes between them. An HTTP request is constructed in the adapter, and follows the given pattern: `http://maps.googleapis.com/maps/api/directions/OUTPUT_FORMAT?origin=START_GPS&destination=DESTINATION_GPS&units=DISTANCEUNIT&sensor=true` |false

URL encoded parameters used:

- **START GPS**: Starting location.
- **DESTINATION GPS**: Destination location.
- **OUTPUT_FORMAT**: Only **kml** produces an output from which the route's coordinates can be extracted.
- **DISTANCEUNIT**: Distance unit.

For example:

```
http://maps.googleapis.com/maps/api/directions/xml?origin=50.249313,18.674927&destination=48.170482,11.5979&sensor=false
```

The example link should respond with XML document with all the data about traveling direction. Follow the link to investigate the service response.

Google Maps Limitations

Using ***GClientGeocoder*** in javascript (see: Client Side Geocoding) to do the geocoding means that the query comes from website visitor's IP address. As a result it remains irrespective of what site they are using. The requests hit the user's IP quota. As any single user is unlikely to make many queries we really do not need to worry about customers quota.

If you are using the HTTP interface (see: Web Service Geocoding) then the request comes from our server IP, in which case all queries come from the same IP and hence the system may hit the quota. To avoid it, you may adjust ***geocogingCronJob*** that equips locations from data base with GPS coordinates.

A good idea behind geocoding is to combine both approaches. For example: when a customer fills in his location in a form, the address data should be submitted for geolocation by javascript. This way the limitation problem is isolated from the number of site's visitors. The addresses of stores, warehouses, etc. should be handled via storelocator that uses HTTP web service.

How to Write Your Own Adapter

It is possible to use other adapters for Google Maps service or other services. Such adapter must implement the **de.hybris.platform.storelocator.GeoWebServiceWrapper**. For more details, see Store Locator Implementation.

```
public interface GeoWebServiceWrapper
                                {

                                /**
                                * Translate IAddress to GPS using 3rd party service
                                *
                                * @param address
                                * @return GPS
                                * @throws GeoServiceWrapperException
                                */
                                GPS geocodeAddress(final Location address) throws GeoServiceWrapper

                                /**
                                * Translate AddressData to GPS using 3rd party service
                                *
                                * @param address
                                * @return GPS
                                * @throws GeoServiceWrapperException
                                */
                                GPS geocodeAddress(final AddressData address) throws GeoServiceWrap

                                /**
                                * Get distance and route that is between two addresses
                                *
                                * @param start
                                *           {@link Location}
                                * @param dest
                                *           {@link Location}
                                * @return {@link DistanceAndRoute}
                                * @throws GeoServiceWrapperException
                                */
                                DistanceAndRoute getDistanceAndRoute(final Location start, final Lo

                                /**
                                * Get distance and route that is between start GPS location and des
                                *
                                * @param start
                                *           {@link GPS}
                                * @param dest
                                *           {@link Location}
                                * @return {@link DistanceAndRoute}
                                * @throws GeoServiceWrapperException
                                */
                                DistanceAndRoute getDistanceAndRoute(final GPS start, final Locatio

                                /**
                                * Formats the address textual information depending on the implemer
                                *
                                * @param address
                                *           {@link Location}
                                * @return {@link String}
                                * @throws GeoServiceWrapperException
                                */
                                String formatAddress(final Location address) throws GeoServiceWrapp
                                }
```

Warehouse Integration

The Warehouse Integration interface supports communication between a warehouse and SAP Commerce during the Order Management process.

Use Case

Warehouse Integration consists of two interfaces through which:

- Other systems, such as warehouses, can query the SAP Commerce platform for orders.
- SAP Commerce can send information to the warehouse.

Related Information

The SAP Commerce processengine

Warehouse Integration Implementation

Warehouse Integration allows consignments to be sent from the SAP Commerce platform to the warehouse in the course of an order process so that the consignments can be prepared and shipped. It also enables the warehouse to report updates on the consignment status back to the SAP Commerce platform.

The Warehouse Integration provides three methods, as shown in the basic use case diagram and described in detail.

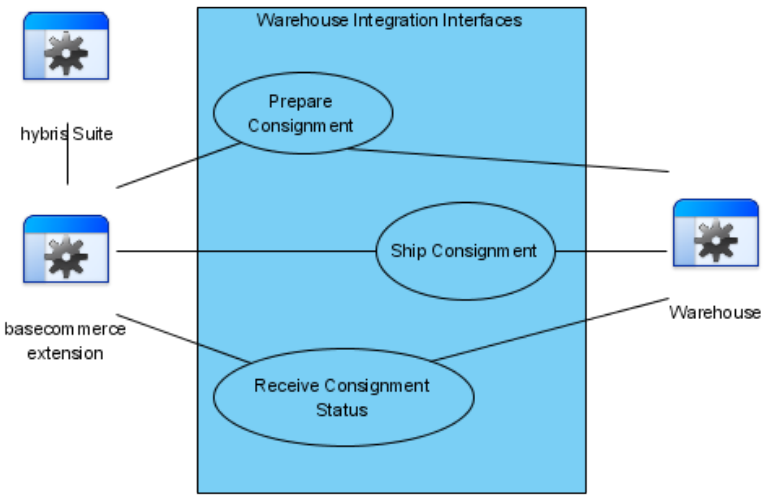


Figure: Warehouse Integration Use Cases

Use Case	Description
Prepare Consignment	Preparing shipment of ordered products
Ship Consignment	Shipping the prepared consignment
Receive Consignment Status	Receiving consignment status from the warehouse

The Warehouse Integration interfaces are typically used when - in the order process - a warehouse needs to get information about a consignment shaped out of an order. The order process then waits for the warehouse to send updates on the consignment's status. After the warehouse has sent the status, the processing of the order is resumed.

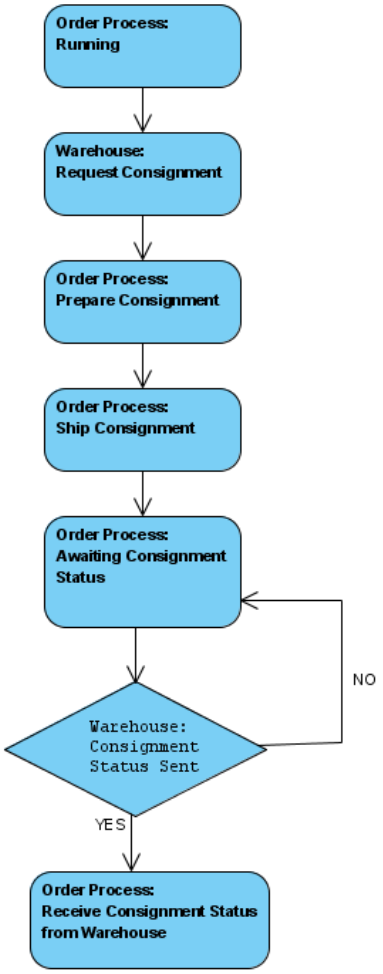


Figure: Basic activities in the process of using Warehouse Integration interfaces.

The Warehouse Integration is a simple reference interface. It is meant to provide an example or foundation on which you can build your own interface according to the specifics of a given order process and the warehouse to be connected.

Reference Interface Example Implementations

Process2WarehouseAdapter

This interface realizes communication from the SAP Commerce platform to the warehouse. It defines two methods:

- **prepareConsignment**: Method tells the warehouse to prepare the shipment of ordered products grouped in the consignment.
- **shipConsignment**: Method allows shipment of the prepared consignment.

```
public interface Process2WarehouseAdapter
{

        /**
         * Send consignment.
         *
         * @param consignment
         *           the consignment
         */
        void prepareConsignment(ConsignmentModel consignment);

        /**
         * Ship consignment.
         *
         * @param consignment
         *           the consignment
         */
```

```
        void shipConsignment(ConsignmentModel consignment);
}
```

Mock Implementation

Warehouse Integration also offers a mock implementation that is realizing the Process2WarehouseAdapter in the most simple way. When **prepareConsignment** is invoked, the mock immediately calls **receiveConsignmentStatus** with status **COMPLETE**. **shipConsignment** implementation is empty.

Warehouse2ProcessAdapter

This interface is used to send information about a consignment's status from the warehouse to the SAP Commerce platform. This method should be invoked when the consignment status is changed on the warehouse side.

```
public interface Warehouse2ProcessAdapter
{
        /**
         * Receive warehouse consignment status.
         *
         * @param consignment
         *           the consignment
         * @param status
         *           the status
         */
        void receiveConsignmentStatus(ConsignmentModel consignment, WarehouseConsignmentStatus stat
}
```

The warehouse provides information about consignment status back to the platform. The table below contains possible status values.

State	Description
CANCEL	A consignment has been canceled by the warehouse.
PARTIAL	Not all parts of the consignment could be realized, some are on back order.
COMPLETE	A consignment is ready for shipment.

Promotions (Legacy)

The promotions module has been replaced by Promotion Engine.

ℹ Note

The legacy promotions module is now disabled out of the box. Customers who want to continue to use the legacy promotions have the option to enable this module manually, but must be aware that legacy promotions are no longer supported.

Customers are strongly encouraged to migrate to Promotion Engine. For instructions, see Migrating to the Promotion Engine.

Overview

The **Promotion Module** consists of a code component that may be used to implement customer sales promotion functionality within web sites and a Backoffice component for creating and managing the promotion data.

The web site code component exposes information about available promotions for use in web site front end code and facilitates the calculation of the effect of promotions on shopping cart and order state. This functionality supports the following tasks:

- Presentation of available promotions to customers
- Application of activated promotions to shopping carts and orders
- Evaluation of partially activated promotions in shopping carts

Promotions (Legacy) vs. Promotion Engine

Promotion Engine offers a number of improvements over the Promotions (Legacy).

The following table describes the improvements available in the Promotion Engine .

	Promotions (Legacy)	Promotion Engine
Architecture	Implemented within the Jalo layer by the `PromotionManager.updatePromotions()` method	Runs on top of the Rule Engine that provides a mechanism for defining business rules with conditions and actions
	Offers promotions at Product-level and Order-level types	Introduces templates and Rule-Aware Object (RAO) Actions
Features	Includes two messages for all Promotion types - **Fired Message** and **Could Have Fired Message**. Fired message is displayed when the promotion was successfully applied, while Could Have Fired message indicates that the criteria of the promotion is partially fulfilled	Offers promotion messages and potential promotions. • Applied Promotion Messages • Potential Promotion Messages
	Available in the Backoffice	Represented in a user interface where you can create promotion templates using predefined actions and conditions
	Includes the order entry consumption feature. As promotions use order entry items, the same promotions consume these items.	Supports the order entry consumption concept of the legacy Promotion module. When a promotion already used an item, the Promotion Engine prohibits other promotions from using the same item.

Out-of-the-box Templates

ℹ Note

+ : refers to an improved version and supports new features.

Promotion Module (Legacy)	Promotion Engine
Order threshold fixed percentage discount	Order threshold percentage discount on cart
Order threshold fixed discount	Order threshold fixed discount on cart
Percentage discount	Product percentage discount
Not supported	Product fixed discount
Not supported	Order threshold percentage discount on product(s)

Promotion Module (Legacy)	Promotion Engine
Fixed Price	Product fixed price
Buy X get Y free	Buy X get Y Free +
Order threshold perfect partner	Order threshold fixed price on product(s)
Perfect Partner	Product perfect partner fixed price +
Not supported	Product perfect partner percentage discount
Perfect partner bundle	Product bundle fixed price +
Multi-buy	Product multi-buy fixed price +
Order threshold free gift	Order threshold free gift
Not supported	Customer specific percentage discount
Order threshold change delivery method	Order threshold change delivery method

How the Promotion Engine links to the Promotion Module (Legacy)

To maximize backwards compatibility, the new Promotion Engine provides an implementation of the Promotion service. This allows promotion results to be written by the new Promotion Engine and read by the existing application that calls the Promotion service such as the Accelerator.

This implementation retains the Promotions (Legacy) in your project. Promotion Engine overrides certain Spring beans. Promotions (Legacy) items continue to exist, but should be unused.

Migrating to the Promotion Engine

For details on migration, see Migrating to the Promotion Engine.

Related Information

Promotions (Legacy)
Promotion Engine

Migrating to the Promotion Engine

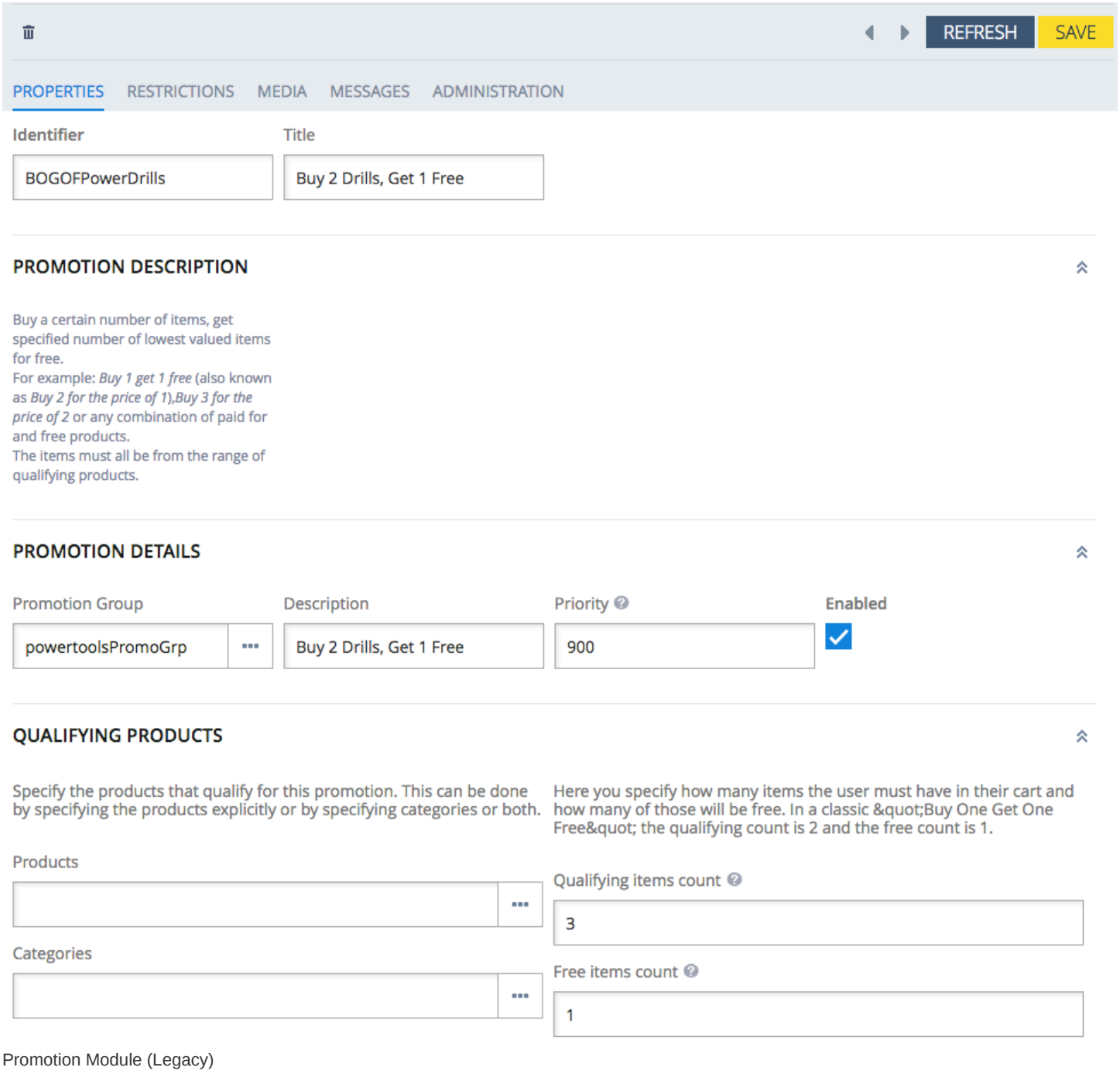
You can manually recreate legacy promotion rules in Promotion Engine by using the Backoffice Rule Builder.

Manual Migration

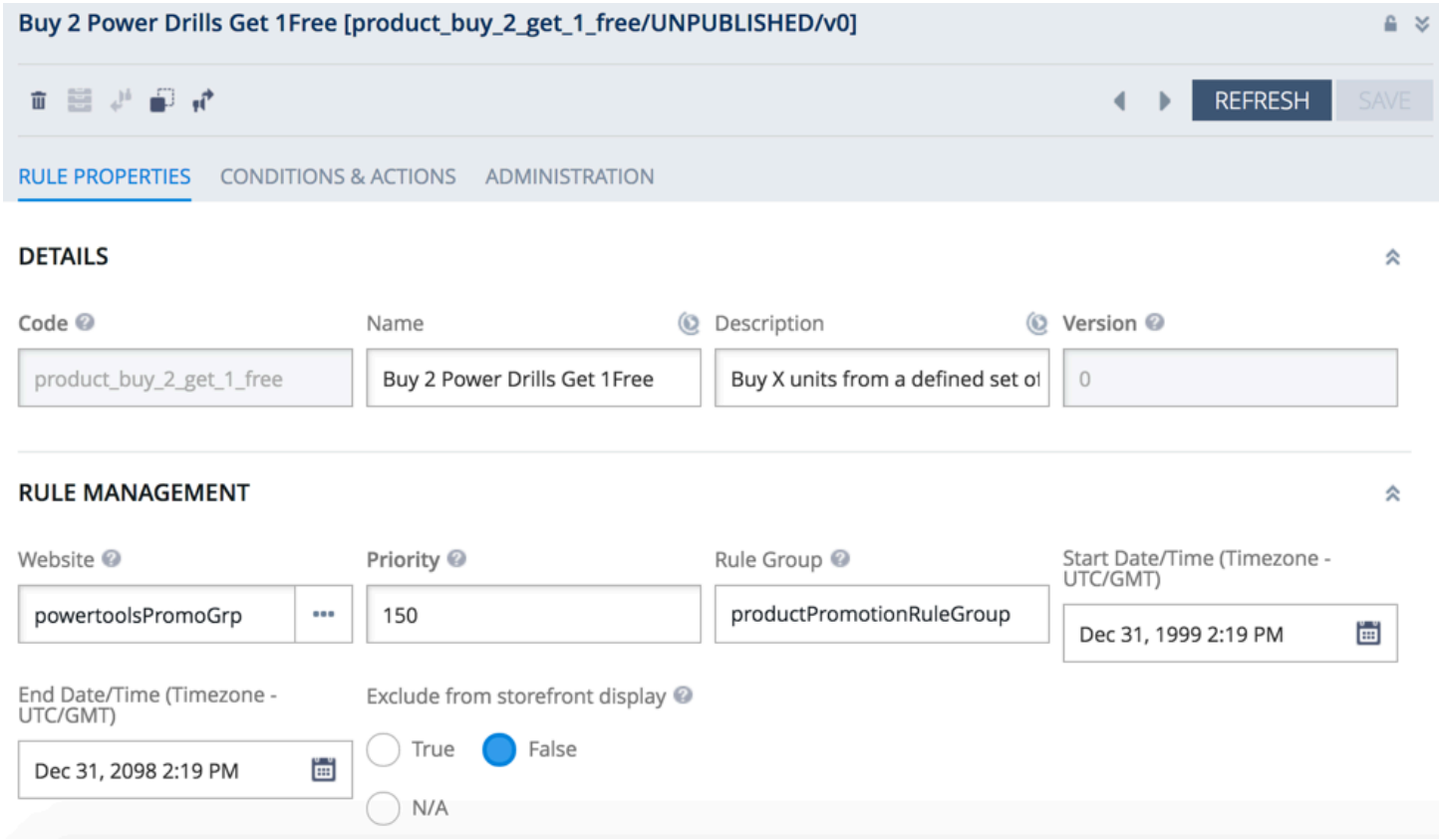
There is no automatic migration from the legacy promotions module to Promotion Engine. Rules must be recreated manually in the Backoffice.

Example 1: Buy X Get Y Free

Let us compare two out-of-the-box Promotion Rules that are both available in Promotion Module (Legacy) and Promotion Engine. The following sections demonstrate how to manially migrate your data to Promotion Engine.



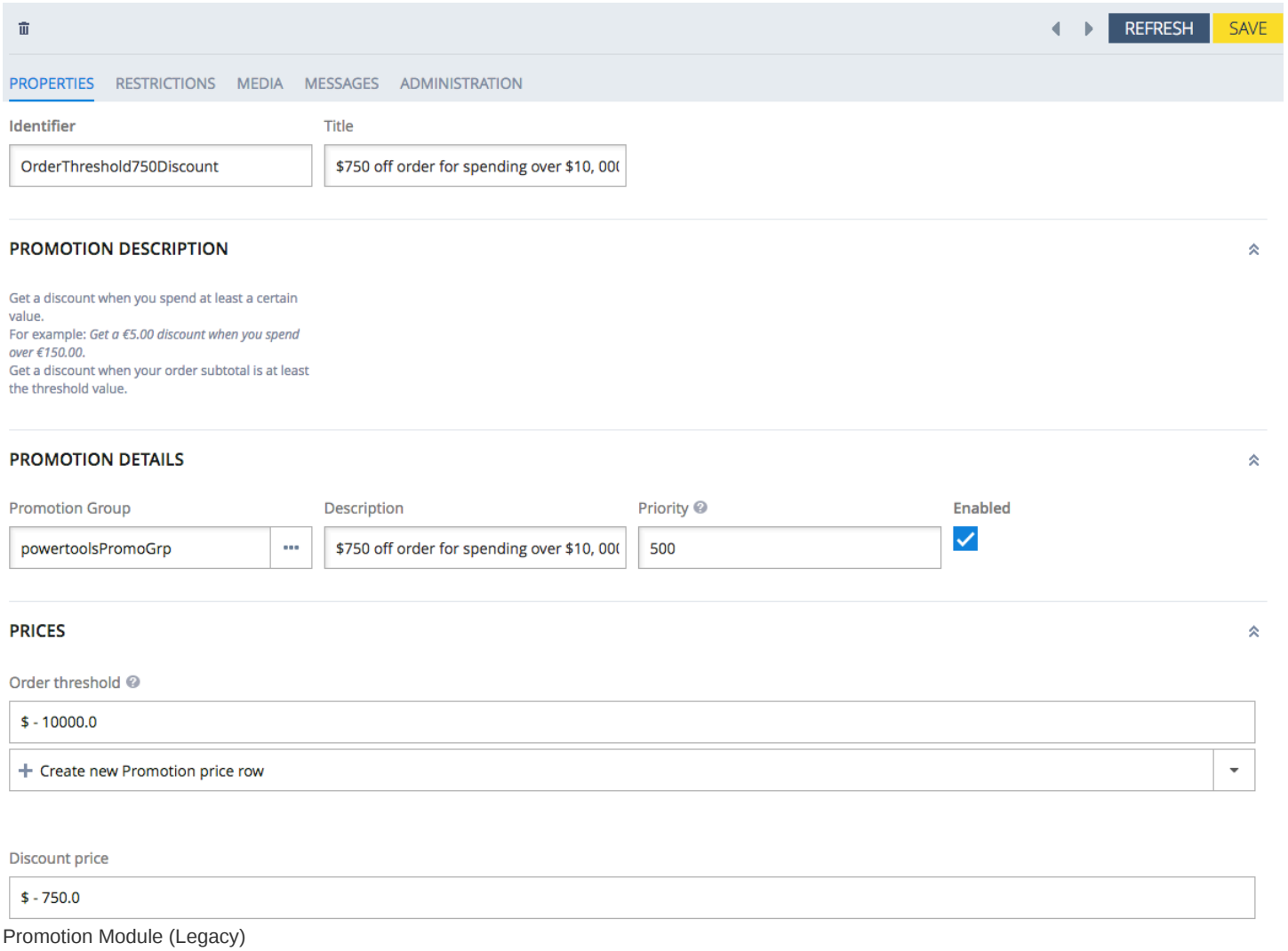
Promotion Module (Legacy)



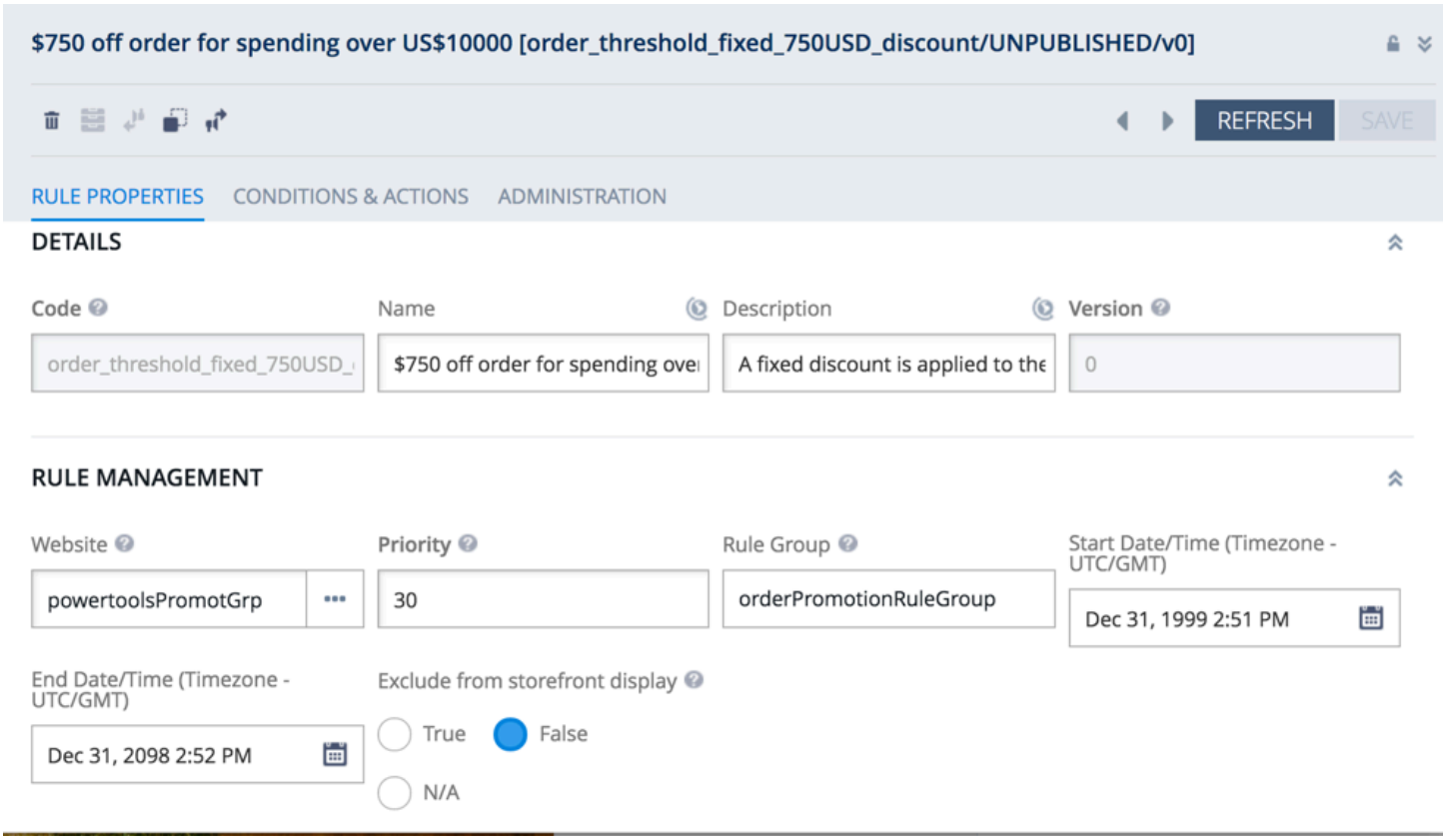
Promotion Engine

Promotion Module (Legacy)	Promotion Engine
Promotions > B2GOFPowerDrills	Promotion Rules > product_buy_2_power_drills_get_1_free
Properties > Identifier	Rule Properties > Code
Properties > Title	Rule Properties > Name
Properties > Description	Rule Properties > Description
Properties > Enabled	Rule Properties > Status
Properties > Priority	Rule Properties > Priority
Does not exist	Conditions & Actions > Conditions > Container > Container ID
Properties > Qualifying items count	Conditions & Actions > Actions > Percentage discount on partner products > Qualifying product containers
Properties > Free items count	Conditions & Actions > Actions > Percentage discount on partner products > Target product containers
Restrictions > Start date	Rule Properties > Start Date/Time (Timezone - UTC/GMT)
Restrictions > End date	Rule Properties > End Date/Time (Timezone - UTC/GMT)
Messages > Fired message Messages > Could have fired message	Rule Properties > Message

Example 2: Order Threshold Fixed Discount



Promotion Module (Legacy)



Promotion Engine

Promotion Module (Legacy)	Promotion Engine
Promotions > OrderThreshold750Discount	Promotion Rules > 750_USD_off_orders_over_10000_USD
Properties > Identifier	Rule Properties > Code
Properties > Title	Rule Properties > Name
Properties > Promotion Group	Rule Properties > Rule Group
Properties > Description	Rule Properties > Description
Properties > Enabled	Rule Properties > Status
Properties > Priority	Rule Properties > Priority
Properties > Order Threshold	Conditions & Actions > Condition > Cart total
Properties > Discount price	Conditions & Actions > Action > Fixed discount on cart
Messages > Fired message Messages > Potential to fire message	Rule Properties > Message
Restriction > Start date	Rule Properties > Start Date/Time (Timezone - UTC/GMT)
Restriction > End date	Rule Properties > End Date/Time (Timezone - UTC/GMT)

Promotion Engine ImpEx File

```
INSERT_UPDATE PromotionSourceRule;code[unique=true];priority;maxAllowedRuns;ruleGroup(code);conditi
;750_USD_off_orders_over_10000_USD;30;1;orderPromotionRuleGroup;"[{""children"":[],""parameters"":
;product_buy_2_power_drills_get_1_free;150;1;productPromotionRuleGroup;"[{""definitionId"":""y_cont
```

Promotion Module (Legacy) ImpEx File

Buy 2 Power Drills, Get 1 Free

```
INSERT_UPDATE AcceleratorProductBOGOFPromotion;PromotionGroup(Identifier[default=$defaultPromoGrp])
;;;B2GOFPowerDrills;Buy 2 Power Drills, Get 1 Free;Buy 2 Power Drills, Get 1 Free;Buy 2 Power Drill
```

$750 off order for spending over $10,000

```
INSERT_UPDATE OrderThresholdDiscountPromotion;PromotionGroup(Identifier[default=$defaultPromoGrp]);
;;;OrderThreshold750Discount;$750 off order for spending over $10,000;Place an order for over $10,0
```

`promotions` extension (Legacy)

The `promotions` extension consists of a code component that may be used to implement customer sales promotion functionality within web sites and UI components for creating and managing promotion data. The `promotions` uses the BaseStore type, which is a part of the basecommerce extension.

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.

ℹ Note

The `promotionsbackoffice` adds additional configuration to the `backoffice` extension. It depends on the `promotions`. Users have also the possibility to manage promotions using the Backoffice Administration Cockpit. For instructions, see Creating Promotions (Legacy).

ℹ Note

Promotions are not updated automatically when a cart or an order is modified. Neither a cart nor an order is aware of **promotions**. This is because Cart and Order are SAP Commerce types defined in the `core` extension and **promotions** are defined in the `promotions` extension. If the cart and order were designed to be aware of promotions, the entire definition of cart and order would have to be changed and the `promotions` extension would be required for all projects within the SAP Commerce. The consequence of the cart and order being unaware of promotions is that you must explicitly update the promotions whenever the cart or order is updated (products added, removed, number of products changed, etc). To update promotions, call the PromotionsManager's `updatePromotions(...)` method. If you call `cart.calculate()`, you must also update the promotions.

For more information, see the following:

- basecommerce Extension
- Base Store

Caching Promotion Results for Performance

(Legacy promotions module) Learn how to enable caching of promotion results to improve promotions performance.

Enabling Promotion Results Caching

ℹ Note

When you enable caching of promotions results, you should be aware of the following:

- If promotion results caching is disabled: Warnings are displayed during the server start-up for new types that were added to the `promotions-item.xml` file but were not initialized.
- If promotion results caching is enabled: A system update is required to add the meta-information to the database for any new types that were added to the `promotions-items.xml` file.

If a cart contains many items and promotions, performance may degrade when adding new items. For example, a grocery cart with more than 100 items and more than 25 promotions may exhibit slow performance when adding a new item. To improve performance in these situations, you can enable caching of promotion results

To enable caching of promotions results, uncomment three beans in the `promotions-spring.xml` file, as shown in the following example.

```
<!-- CACHING START -->

<!-- Replace the Promotions Manager with a version that understands caching -->
<bean id="promotions.manager" class="de.hybris.platform.promotions.jalo.CachingPromotionsManager" i
```

```
        <property name="cache" ref="promotionsCachingStrategy"/>
</bean>

<alias alias="promotionsCachingStrategy" name="defaultPromotionsCachingStrategy"/>
<bean id="defaultPromotionsCachingStrategy" class="de.hybris.platform.promotions.jalo.DefaultCachir

<!-- Override the Cart so that calls to get promotion results can be intercepted -->
<bean id="de.hybris.platform.jalo.order.Cart" class="de.hybris.platform.promotions.jalo.order.Promc

<!-- CACHING FINISH -->
```

ℹ Note

After enabling caching of promotion results, you must update the system and restart the server.

Known Issues

The following issues exist with promotion results caching:

- Direct access to promotion results through FlexibleSearch is not supported. There are no restrictions on orders as all promotion results for orders are persisted.
- Implementations that support the saving of carts between sessions do not recover the promotion results when a customer returns to the store at a later time. This leads to inconsistent cart information unless the cart is recalculated when it is restored in the new session.
- It is assumed that sticky sessions are used so that a customer's request is always accessing the same node. This ensures that promotion results are not lost. If the customer is not accessing the same node, then the cart must be recalculated before the promotion results can be viewed. Alternatively, a distributed cache could be implemented to allow the cached promotion results to be visible on all nodes in the cluster.
- Customer Service Cockpit users may not see the promotion results associated with a cart for the same reason as the previous point.

You can override the caching strategy in order to provide an implementation that uses a cluster-wide cache instead of the default Java HashMap. This should allow cluster-wide visibility of promotion results. However, promotion results get deleted and recreated whenever the cart is recalculated. Any design needs to accommodate the performance and latency implications of synchronizing changes to other nodes within the cluster.

Cart Restoration with Promotion Cache

When you are using the cart restoration feature, carts can be loaded from the database on any node. Since the default implementation of the promotion cache is not cluster-aware, the promotion result of the cart that you loaded might be out of sync with the promotion cache. That's why you need recalculate the cart each time you load it. By default, the cart restoration feature doesn't recalculate the cart each time it loads it from database, for performance reasons. It will only recalculate it once the "cart validity period" has passed. You can configure the cart validity period (in seconds) in the `local.properties` file, as the example below shows:

```
commerceservices.cartValidityPeriod=12960000
```

Promotion Types (Legacy)

Find out more about the types of promotions available in the legacy Promotions module.

Promotion Classes and Types

Promotions contain a rule set that defines the conditions for activation and a set of operations which determine the effect of the promotion once activated.

The following classes of promotion are included with the `promotions` extension:

- **Product level** promotions are activated based upon the products within the line items of a shopping cart or order.
- **Order level** promotions are activated based upon shopping cart or order attributes.

ℹ Note

Discount Application Logic

For details on how specific promotion types work, refer to: Promotion Discount Application Logic.

Product Level Promotions

The complete list of out-of-the-box promotion types is provided below. It is possible to identify what products qualify for a promotion by the direct assignment of products and/or categories. Any product that belongs to a qualifying category is deemed qualified for the promotion. Promotion restrictions are required to further filter the qualifying product list.

Promotion	Description	Example
Bundle	Purchase one of each product from a defined set for a combined total price.	Buy A, B, and C for €50.
Buy X get Y free	Purchase a certain number of products from within a defined set and add further products from the same set at no additional cost.	Buy one get one free.
Fixed price	Purchase from within a defined set at a fixed unit price.	All shirts €5 each.
Multi-buy	Purchase a certain number of products from within a defined set for a fixed price.	Buy any 3 shirts for €50.
One to one perfect partner bundle	Purchase a certain product and another defined partner product for a fixed total price. The cart must contain the base product and the partner product to qualify.	Buy this game and the selected partner accessory together for €25.00.
Percentage discount	Receive a percentage discount on all products within a defined set.	20% off all cameras.
Perfect partner	Purchase a certain product from within a defined set and another partner product from a different defined set and pay a fixed price for the partner product.	Buy a game for €10 with each games console.
Perfect partner bundle	Purchase a certain product along with a specified number of products from within a defined set for a combined total price.	Buy a games console and 3 accessories for €200.
Stepped multi-buy	Purchase a number of products from within a defined set, there are multiple tiers of product quantities and fixed prices.	Buy any 3 shirts for €50, 5 for €65, and 7 for €75.

Order Level Promotions

Promotion	Description	Example
Order threshold free gift	A free gift is added to the order when the threshold order value is exceeded.	Spend over €50 to receive a free t-shirt.
Order threshold free voucher	A free voucher is given out when the order reaches a certain value.	Get a free €5 voucher when you spend over €150.00. Get a free voucher when your order subtotal is at least the threshold value
Order threshold fixed discount	A fixed value discount is applied to the order when the threshold order value is exceeded.	Spend over €50 to receive a €3 discount.
Order threshold change delivery mode	A different delivery mode is applied to the order when the threshold order value is exceeded.	Spend over €10 to get free shipping
Order threshold perfect partner	Purchase a certain product from within a defined set for a fixed price when the threshold order value is exceeded.	Spend over €50 to get any shirt for €5.

Promotion Evaluation

Promotions are evaluated against a shopping cart or order by the promotions extension. There are two modes of evaluation:

- Evaluate which of the available promotions have been activated or "fired"
- Evaluate which of the available promotions have been partially activated or "could fire".

Partially Activated Promotions

Promotions are partially activated when some of the activation criteria have been met; e.g. when two qualifying products from a three product multi-buy have been added to the shopping cart. Partially activated promotions do not "consume" line items from the shopping cart (a consumed line item is one that has been used in a fired promotion and cannot be used by another promotion), and these are still available for use in evaluating additional partially activated promotions. Indeed, a single product in a line item may be associated with several partially activated promotions, for example a perfect partner and a multi-buy promotion. It is useful to know that a promotion has been partially activated so that this may be communicated to customers on a website. For instance, a message such as "Add another snowboard to your basket and receive 50% off your order" could be displayed.

Promotion Priority

Each promotion has a priority and promotions are evaluated in priority order, highest priority first. Typically product promotions are evaluated before order promotions; however this is entirely controlled by the priority.

Line items in the cart or order are consumed by fired promotions and become ineligible for inclusion in subsequent promotions during the evaluation.

Every promotion result has a certainty that is a fraction number between 0 and 1. The closer the promotion actually is to firing, the closer it is to 1. A promotion result also has a consumed count that indicates how many items in the cart can currently be consumed by the promotion. These two attributes can be used to order or filter promotions that are not close to firing. For example, you can configure it so that a "buy 4 get 1" promotion is only displayed to the customer when the customer already has 2 items in the cart.

Promotion Discount Application Logic

For each promotion, a discount is added either to the order or to the item. This makes it possible to adjust the order or item total to apply the savings following the application of the promotion.

- For all promotions that discount a group of items (Bundles), a separate prorated price is also calculated and stored for each consumed order item, that is, the discount is applied in proportion to each item price.
- For all promotions in which an item is given away for free (Buy X Get Y Free), a price of zero is provided for the correct number of units of each free item. For example, if the promotion is buy 4 get 1 free, four items will have a regular price and the price for one item will be set to zero.
- For all promotions in which discounts are applied to items individually (Fixed Price or Percentage Discount) the discount is also applied to each individual item. This calculation data is externalized by the `promotion` module and can be used in project-specific developments for purposes such as reporting or for sending sales data to ERP systems.

The following table provides examples of orders using the three kinds of promotions and the discount logic that applies to each of them.

Promotion Name	Description
Product Level Promotions	
Bundle	To benefit from the BundlePromotion, the user must add 3 items (A, B, and C) which, when bought together, cost 750 EUR and result in a saving. All the other multi-item purchase promotions are variants of a bundle where the rules regarding how an item can qualify differ, but the application of the discount remains the same. That is, a user saves by buying multiple items. If the usual Price for Item A is EUR 86.80, B is EUR 523.99 and C is EUR 381.64. SAP adds an order level discount of EUR 242.43 to ensure the customer is only charged EUR 750 for the 3 items. SAP stores additional pro-rated totals for each 'consumed' item. A is EUR 65.60, B is EUR 395.99 and C is EUR 288.41. For these types of promotions, SAP adds an order level discount and pro-rates the totals for each consumed item.
Multi Buy	Operates like Bundles, except the user is given the flexibility to choose a subset of items from a larger list.
Stepped Multi Buy	Extends the Multi Buy concept by supporting bundle price banding based on the quantity of candidate items that are added to cart. For example you may sell 4 tires at a bundle price of 400 EUR but define 5 as being priced at 480 EUR.
One-to-One Perfect Partner	Operates like bundles however the user can only be offered one partnering product when selecting a certain base product. The general principal of this promotion is to cross-sell a suitable partnering product when a user has chosen a hero product (a hero product is the one that triggers the one-to-one perfect partner promotion). For instance, sell an HDMI cable when a user adds a Blu-ray player to the cart.
Perfect Partner Bundle	Extends the one-to-one perfect partner promotion by allowing the customer to pick 1 or more (configured) of a set of partnering products when the customer has committed to buying the hero product.
Buy X get Y Free	Implements the classic promotion where a user can pick a certain number of items from a set and get the cheapest item free.

Promotion Name	Description
	Using the example promotion Buy 2 of item A (normally priced at EUR 523.99) and you get 1 free, you have an order level discount added of EUR 523.99. The system also records the fact that 1 unit of item A is full price and the second unit is free.
Fixed Price	Used to group a number of items that are all to be sold at a fixed unit price (i.e. anything for EUR 5). Percentage discount promotions are the same except a fixed discount is applied to each item's unit (or base) price (i.e. 20% off all Run Flat tires). Both promotions result in a discount being added at the item level. This results in the same behavior as applying a product level discount to a product using the default pricing functionality. Consider a Fixed Price promotion where a group of items have been defined as being sold at EUR 300 each. If Item A was priced at EUR 364 normally, it receives an item level discount of 64.
Percentage Discount	Percentage discount promotions also result in an absolute discount value being applied directly to the item rather than at order level. This absolute value is calculated by applying the percentage discount on the order entry base price (a base price is a product price before any discounts). For instance, if you have a product priced at 269 EUR that has a EUR 10 standard SAP discount (that is, not one that is based on a promotion) in addition to being applicable for a 19% percentage discount promotion, the product is discounted EUR 61.11 (EUR 51.11 which is 19% of EUR 269 and the additional EUR 10 discount). It is important to understand that the percentage discount promotion is applied before the fixed discount, as this changes the final price (please see screen shot 6). Consider the case of a product that costs EUR 100 , has a standard SAP discount of EUR 10 and is also used as part of a 10% percentage discount promotion. If the promotion is applied on the base price (EUR 100), which is how it's normally done, the final price is (100 EUR * EUR .9 - EUR 10), which is EUR 80. In the other case, where the fixed discount is applied first, the final value is (EUR 100 - EUR 10) * .9, which is EUR 81.
Perfect Partner	Results in the same item level discount rules. After selecting one of a set of hero products, the customer can choose one from a set of perfect partner products and get that partner product at a special fixed price.
Order Level Promotions	
Order Threshold Fixed Discount	Allows the customer to get a single product at a fixed price when the discounted order subtotal (i.e. not including delivery costs etc.) reaches a threshold value. The discount is stored at item level.
Order Threshold Perfect Partner	Gives the customer a fixed order level discount when the discounted order subtotal reaches a threshold value.
Order Threshold Free Gift	Adds the specified free gift product to the cart flagging it as a give-away. As with the Buy X Get Y Free, the total price of the free items is adjusted to 0.

Related Information

Promotion Types and Management
promotions Extension

Creating Promotions (Legacy)

Promotions can be created and edited through the Backoffice Administration Cockpit. By default, the Promotion type appears in the Marketing group.

Most promotion types require some initial configuration to define their behavior. Newly created promotions are initially in a disabled state, to prevent unconfigured, or partially configured, promotions from becoming available to customers. New promotions must be enabled before they become available on a website.

Promotions Tutorial

A tutorial on creating promotions is available. To learn more, see Creating Promotions Using the Backoffice Administration Cockpit.

Related Information

Promotion Types and Management

Creating Promotions (Legacy)

In the legacy Promotions module, create and manage promotions in a user interface.

Context

Most promotion types require some initial configuration to define their behavior. Newly created promotions are initially in a disabled state, to prevent unconfigured or partially configured promotions from becoming available to customers, and must be enabled before they become available on a website. Below you will find a short tutorial on how to create a promotion.

Procedure

1. 1. Log in to the **Backoffice Administration Cockpit** using an account with sufficient rights to create promotions.
2. Expand the **Marketing** entry in the Explorer Tree and click on the **Promotions** entry.
3. Select the **Promotion Type** from the list. For details on Promotion Types see: Promotion Types and Management.
4. A new window is opened. In the General section, provide the following attribute values:
    - **Identifier**
    - **Title** - a title describing your **Promotion**.
    - **Promotion Group** - select a **Promotion Group** from the list.

      ### ℹ Note

      Promotion Groups

      A promotion group is a set of promotions that may be selected for use on a website. The purpose of promotion groups is to enable website developers to alter the set of promotions available based upon some context. A promotion group may contain zero or more promotions. A promotion must always be associated with a promotion groups.

- **Description** - a description of your **Promotion**.
- **Priority** - the priority of you **Promotion.**
- **Enabled** - tick the checkbox for your **Promotion** to become active.

5. In the **Qualifying products** section specify the products and categories that qualify for your **Promotion**.
6. In the **Bundle price** section specify the price for the entire bundle.

> **Note**
>
> **Currencies**
>
> Each currency has a different value, and all currencies which qualify for this order must have a specified value.

7. In the **Restrictions** tab, you can set restrictions for your **Promotion.**

   There are four restriction types included with the **promotion** extension and additional types may be developed as required:

   - **Date Restrictions**: Promotions have a start date and end date. A promotion may only be activated between the start and end dates.
   - **Product Restrictions**: Product restrictions prevent promotions from being activated if the shopping cart or order contains any of the products specified.
   - **User and Usergroup Restrictions**: Both user and usergroup are principals. These restrictions can be used to restrict availability of a promotion to any principal or conversely to prevent principals from using a promotion.
   - **Order Restrictions**: These restrictions can be used to restrict the availability of a promotion to selected orders. Order restriction requires the **ycommercewebservices** extension to be present and enabled.

   For example if you want to create a **Product Restriction,** select this option from the list. A new window opens in which you can select the **Product** your restriction will apply to.

8. In the **Messages** tab, you can insert values from the promotion into your messages.
9. In the **Media** tab, you can specify the URL to page with details of your promotion and attach a product banner image.
10. Click **Done** to complete the process. You can then double-click the already created promotion to open it for further adjustment.

Related Information

Promotion Types and Management

Enabling Legacy Promotions

The legacy promotions module is now disabled out of the box. Customers who want to continue to use the legacy promotions have the option to enable this module manually, but must be aware that legacy promotions are no longer supported.

Procedure

1. Copy and paste the following property into the `local.properties` file, which is located in the *<HYBRIS_CONFIG_DIR>* directory. Set the value to true:

   ```
   promotions.legacy.mode=true
   ```

2. For the changes to take effect, restart the SAP Commerce Server.

Vouchers (Legacy)

Vouchers enable you to create and manage vouchers redeemable by your customers.

Overview

Vouchers are employed as a promotional medium tailored to specific marketing strategies. You can create a simple set of reductions to the order as a whole, or create special offers available only for certain products and customers.

ℹ Note

From 6.3, the **Voucher Module** will be deprecated into maintenance mode for the next 18-months, and thereafter dereleased into legacy support.

We launched the new Coupons feature of the Promotion Engine module with release 6.1 that has achieved feature parity with the legacy Voucher Module and offers many other improvements.

Customers using version 6.x of SAP Commerce will have the choice of switching to the new Coupons feature of the Promotion Engine. Because of the big architectural differences, the new Coupons feature of the Promotion Engine is not backwards compatible with the legacy Voucher Module. But it is possible to create all vouchers from the legacy module in Coupons as well. Guides for migration are available as part of product documentation: Migrating Serial Vouchers to Multi-Code Coupons and Migrating Promotional Voucher to Single-Code Coupon.

If you require further support, please contact your Customer Manager.

For more information on the planned deprecation of modules, see Deprecation Status.

ℹ Note

Compatibility with Other Accelerators

Note that you cannot have more than one flavor of the Accelerator on your machine.

ℹ Note

- The `voucherbackoffice` extension depends on the **voucher** extension and adds more configuration to the `backoffice` extension. For details, see voucher Extension and Voucher Types.
- You can also manage vouchers using the Backoffice Administration Cockpit. For details, see Voucher Management.

Key Features

- Online & Offline Vouchers
- Unique and secure voucher identification numbers
- Absolute and relative discounts as well as S&H discounts
- Employee discounts
- Ability to limit voucher validity to products, quantities, minimal order value, time-spans, and customer groups (new customers, recurring customers etc.
- Creating vouchers for cross-channel use

Through a combination of Voucher Restrictions, the creation of complex voucher eligibility models is possible as well. The Voucher Module comes with a number of predefined restrictions which can be enlarged or adapted through customization.

> **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.

Related Information

Vouchers (Legacy) Features
Voucher Types
Voucher Restrictions
Voucher Management
Voucher Technical Details

Vouchers (Legacy) Features

The `voucher` extension enables you to create and manage vouchers redeemable by your customers.

> **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.

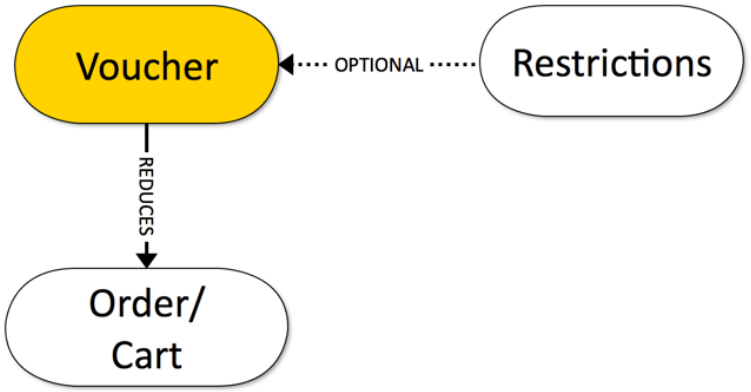
Overview

In SAP Commerce, vouchers are a special form of discounts that can be applied to an order. Ordinary discounts are automatically calculated for an order when their defined prerequisites are met by the content, time, and/or customer of that order. Customers actively redeem vouchers, typically as part of the order process of the shop frontend.

Like normal discounts within the SAP Commerce, vouchers can be set to discount prices in two different ways:

- fixed discount value (e.g. 10 off)
- percentage discount (e.g. 20% off)

A voucher with a generated list of voucher codes. The calculation of the discount called for by a given voucher is done on the total value of the **applicable** product prices, VAT inclusive. Since vouchers can be restricted to apply to certain products only, there may be entries in an order that qualify as **non-applicable** products. Such products are then not subject to the voucher discount rules.

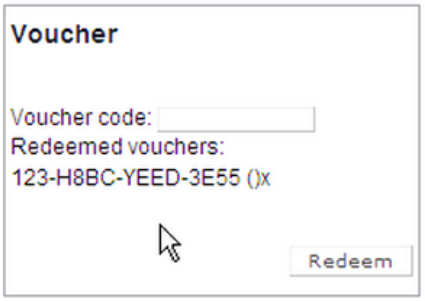


ℹ Note

A new **voucherbackoffice** extension is available. This extension depends on the **voucher** extension and adds extra configuration to the **backoffice** extension.

Voucher Features

The following table presents an overview of the most important voucher features along with links to more information.

Voucher Functionality Area	Description	More Information
Voucher Types	The **SAP Commerce** comes with the following two kinds of vouchers: **Serial Vouchers** and **Promotional Vouchers**. Serial Vouchers have a generated list of voucher codes - each of the codes is marked invalid upon redemption and cannot be used more than once. Promotional Vouchers have a single, invariable voucher code which can be redeemed for any number of times.	Voucher Types
Voucher Management	Vouchers may be created and managed using Backoffice Administration Cockpit (since Release 5.6). Additionally, developers may use the SAP Commerce's API to manage vouchers. 	Voucher Management
Voucher Restrictions	Restrictions optionally allow constraining vouchers to users, products, dates, orders, etc. A voucher can only be redeemed for an order if it conforms to all set Restrictions. A	Voucher Restriction Types

Voucher Functionality Area	Description	More Information
	Valid attribute allows to invert the constraints of a Restriction. **Note** **Voucher Restrictions** Voucher-related Restrictions do not correspond to FlexibleSearch-related restrictions. For FlexibleSearch Restrictions, see Restrictions.	

Voucher Types

The voucher module includes two voucher types: serial vouchers and promotional vouchers. A serial voucher has a generated list of voucher codes. A promotional voucher has a single, invariable voucher code.

Vouchers Types

Note

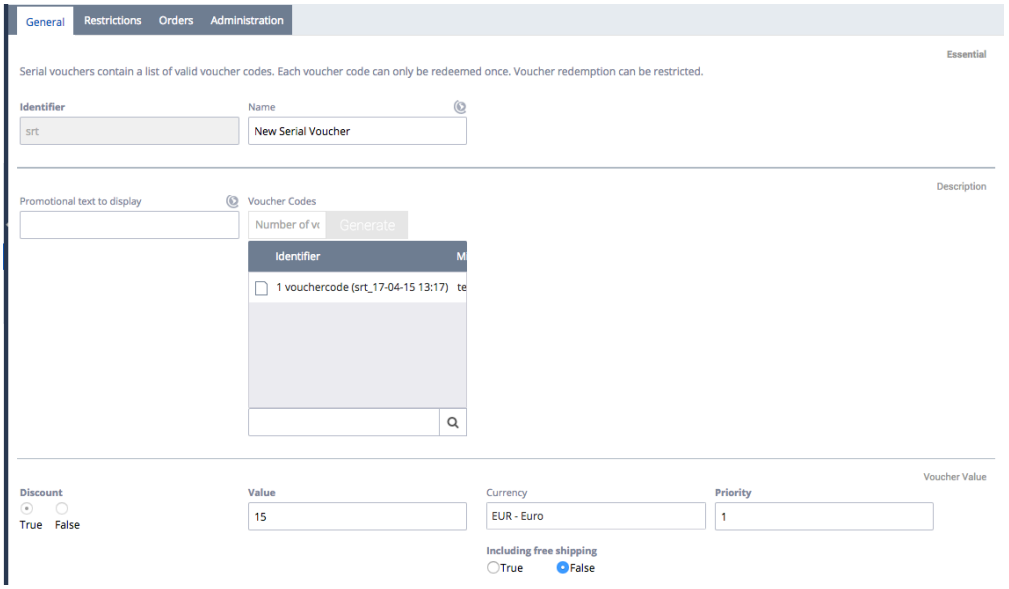
Creating Vouchers

For details about how to create a voucher, see Creating Vouchers Using Backoffice.

Voucher Type	Description
Promotional	• It has a single, invariable voucher code. • It can be set to be redeemed for any number of times, as well as one time or more by the same customer, • During voucher creation, the number of possible redemptions must be set for the Voucher as well as per user.
Serial	• It has a generated list of voucher codes. • A voucher code for a Serial Voucher is marked invalid on redemption and cannot be used more than once. • The Voucher Module keeps track of redeemed voucher codes and does not allow redeeming a single Serial Voucher code more than once.

Serial Voucher

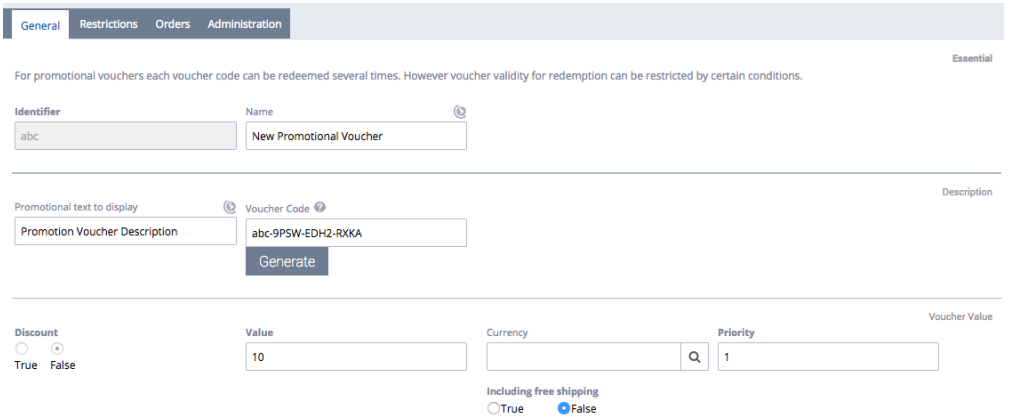
This section provides a list of attributes available for serial vouchers. The attribute values should be provided in the corresponding fields in the Backoffice Administration Cockpit.



Attribute Name	Attribute Type	Mandatory	Description
Promotion Code	String	Yes	The identifier of the voucher.
Description	localized String	No	The description of the voucher.
Promotional text to display	localized String	No	The description of the voucher displayed to the customer.
Voucher Codes	String	No	The voucher code. Can be generated by clicking on the button.
Number of voucher codes to generate	String representation of a number	No	Specifies the number of voucher codes to be generated on a click on the **Generate** button.
Value	String representation of a number	Yes	The numeric value of the voucher.
Priority	String representation of a number	Yes	The priority of the voucher
Currency	String representation of a number	No	The currency in which the `value` attribute is specified. If no currency is specified, SAP Commerce sets the voucher to be relative.
Including free shipping	Radio button	Yes	Specifies if the order this voucher is applied to is shipped for free (Yes) or not (No). The default value is No.

Promotional Voucher

This section provides a list attributes available for promotional vouchers. The attribute values should be provided in the corresponding fields in the Backoffice Administration Cockpit.



Attribute Name	Attribute Type	Mandatory	Description
Promotion Code	String	Yes	The identifier of the voucher.
Description	localized String	No	The (internal) description of the voucher.
Promotional text to display	localized String	No	The description of the voucher for the customer to see.
Voucher Codes	String	No	The voucher code. Can be entered manually or generated by clicking the button.
Value	String representation of a number	Yes	The numeric value of the voucher.
Priority	String representation of a number	Yes	The priority of the voucher.
Currency	String representation of a number	No	The currency in which the `value` attribute is specified. If no currency is specified, the SAP Commerce sets the voucher to be relative.
Including free shipping	Radion button	Yes	Specifies if the order this voucher is applied to is shipped for free (Yes) or not (No). The default value is No.

Related Information

Next Generation Cockpit Framework Admin Area

Vouchers

Voucher Management

You can create and manage vouchers in the Backoffice Administration Cockpit.

Available Documentation

The following documents describes how to create and manage vouchers using the Backoffice Administration Cockpit:

- Creating Vouchers Using the Backoffice Administration Cockpit
- Redeeming and Releasing Vouchers Using the Backoffice Administration Cockpit

Related Information

Voucher Types
Voucher Restrictions
Vouchers

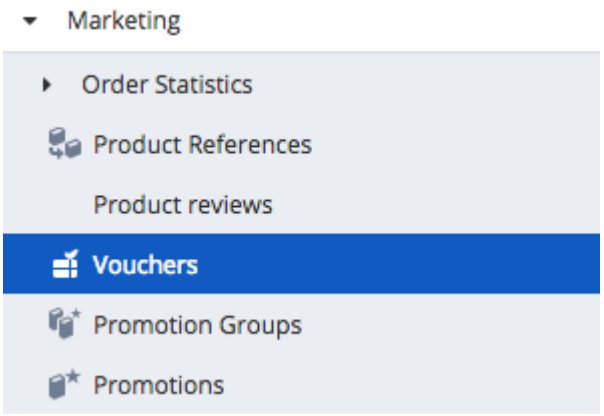
Creating Vouchers Using the Backoffice Administration Cockpit

The Backoffice Administration Cockpit allows you to easily create two predefined types of vouchers: Promotional Vouchers and Serial Vouchers.

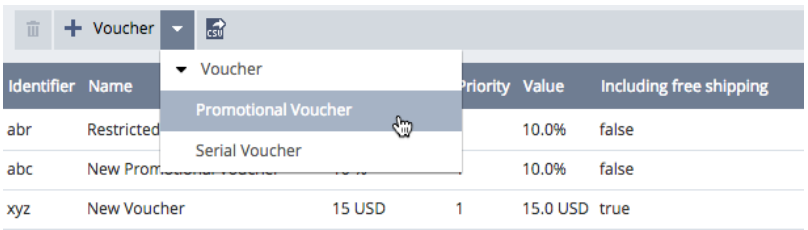
Creating Promotional Vouchers

To create and configure a promotional voucher, do the following:

1. In the Backoffice Administration Cockpit, expand the **Marketing** node in the Explorer Tree and click on the **Vouchers** entry.

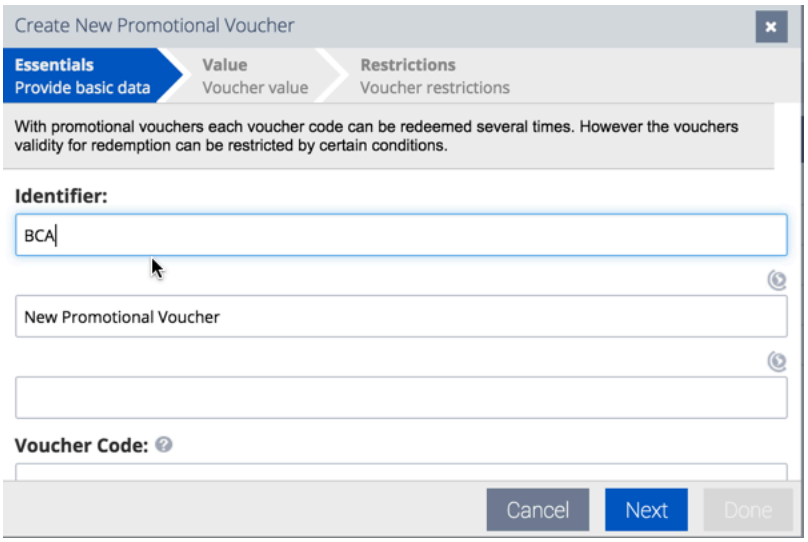


2. Select **Promotional Voucher** from the drop-down menu.



3. In the **Essentials** section, enter the **Identifier** for your voucher.

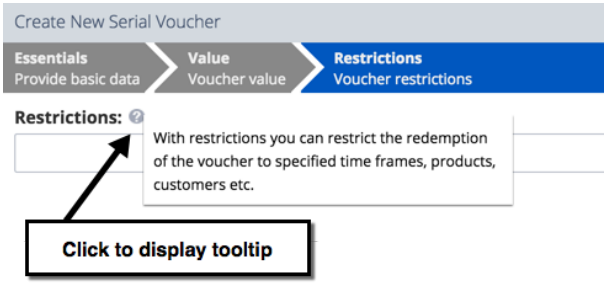
> **Note**
> The Identifier of your choice should consist of up to three characters.



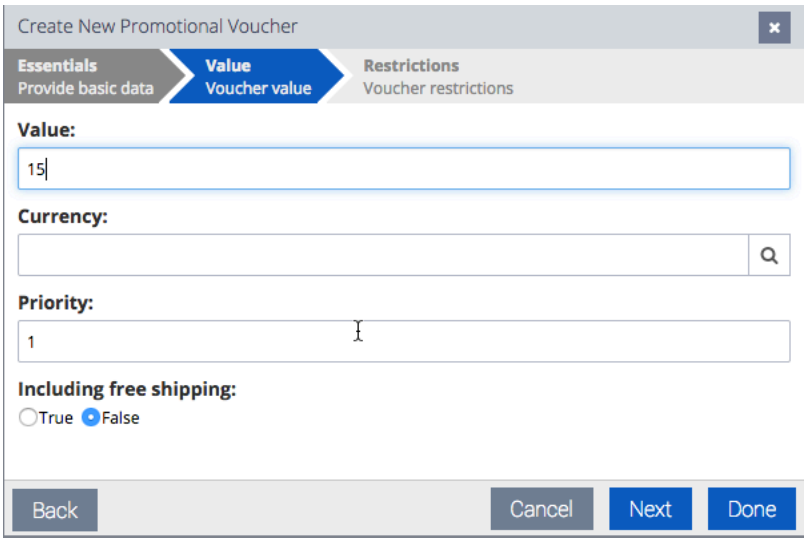
You can also manually add a voucher code or generate it later, once your new voucher is saved.

→ Tip

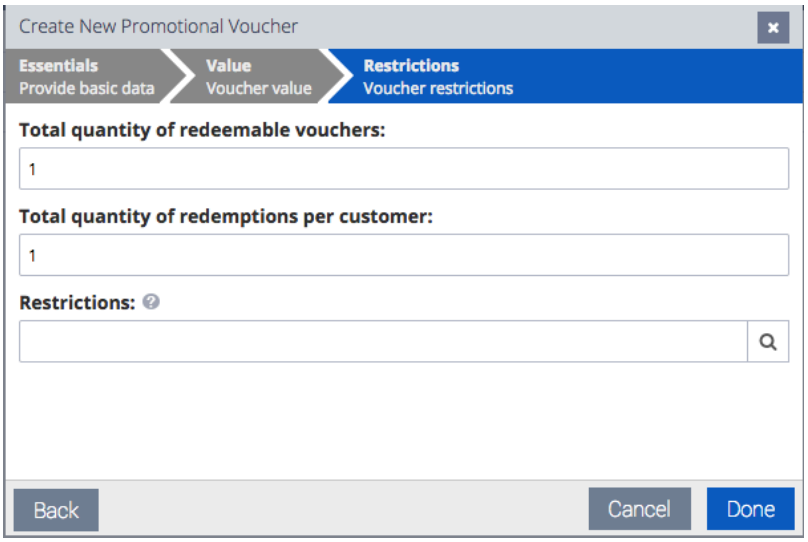
The Backoffice Administration Cockpit provides tooltips so you can quickly access information.



4. Press Next to go to the Value Section. Enter values for the following fields:
    - Value: Enter a numeric value
    - Currency: Use 🔍 to view available options.
    - Priority
5. Press Next to move to Restrictions.



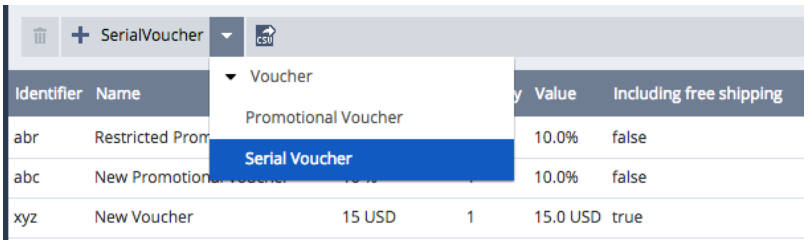
6. If required, add more restrictions for your voucher.
7. Click Done to create the voucher.



Creating Serial Vouchers

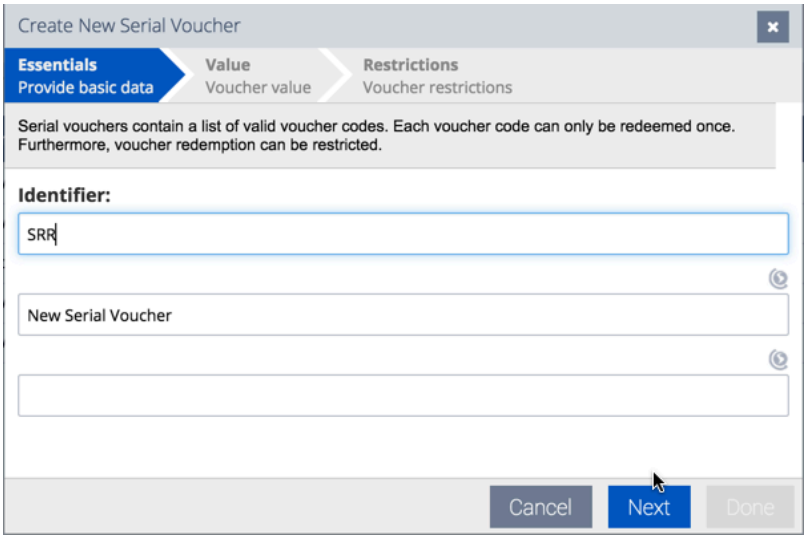
Follow the steps listed below to create and configure a Serial Voucher using a dedicated wizard.

1. In the Backoffice Administration Cockpit, expand the Marketing node in the Explorer Tree and click on the Vouchers entry.
2. Select Serial Voucher from the drop-down menu.



3. In the Essentials section, enter the Identifier for your voucher.

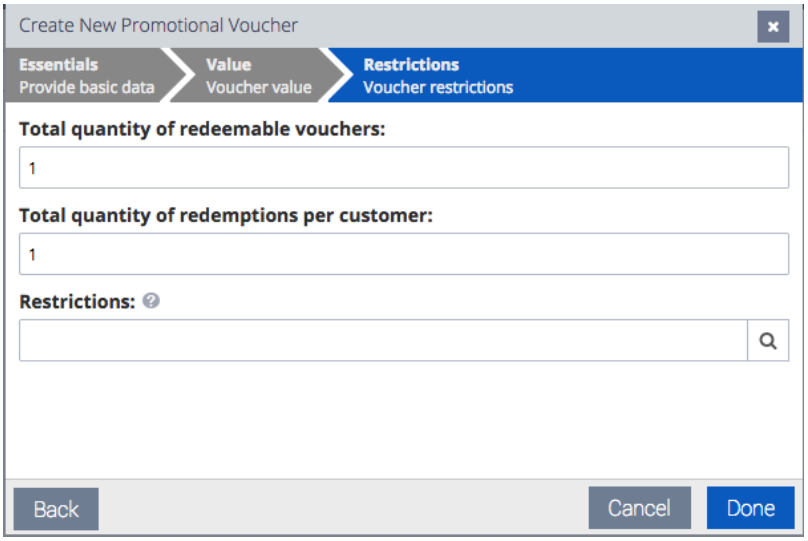
> **Note**
>
> The Identifier of your choice should consist of up to three characters.



4. Press Next to go to the Value section.
5. Enter values for the following fields:
   - Value: Enter a numeric value

- Currency: Use to view available options.
- Priority

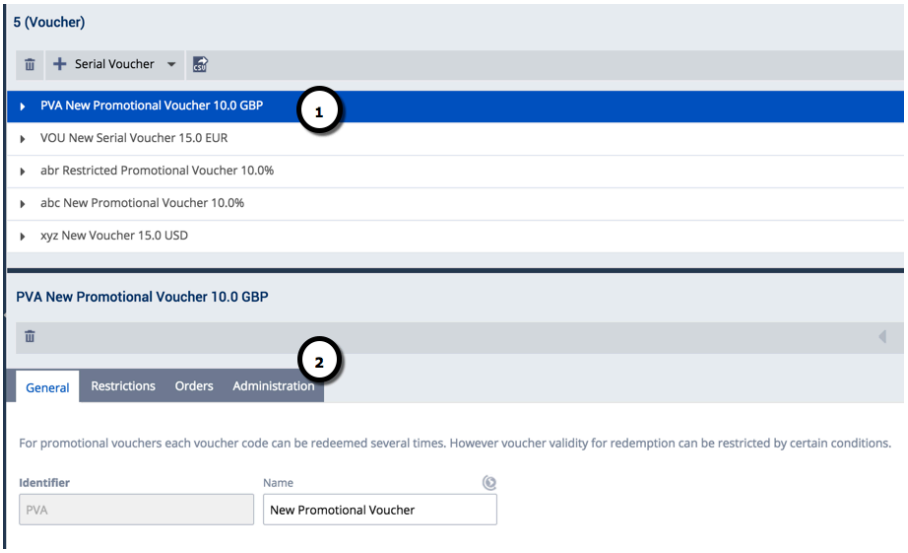
6. Press Next to move to Restrictions.
7. If required, add more restrictions for your voucher.



8. Click Done to create the voucher.

Additional Configuration

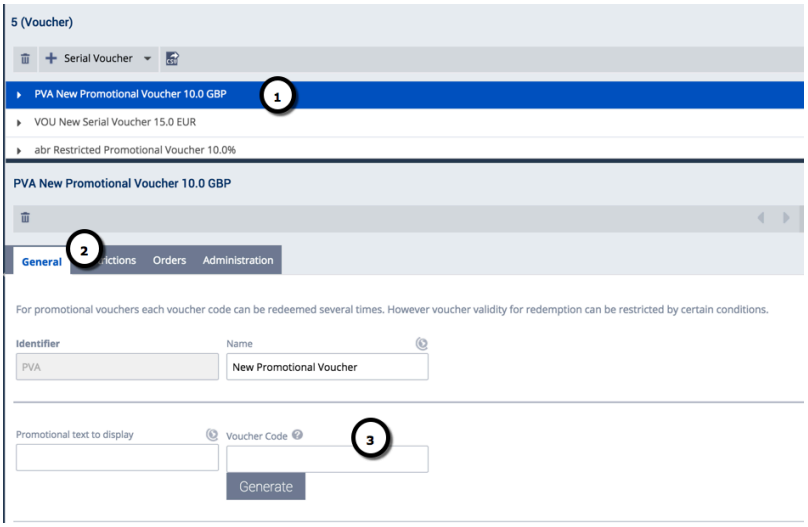
The newly created Vouchers are displayed in the main window. Click on a desired voucher to view more configuration options. Use the tabs to view/modify Voucher attributes. Grayed out fields allow only the read-only mode.



Generating Promotional Vouchers

To generate a promotional voucher, do the following:

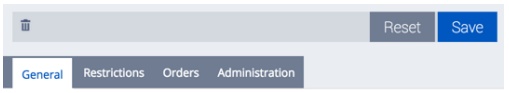
1. Select the desired Promotional Voucher and go to General tab
2. Scroll down to Voucher Code field.



3. Click **Generate**.

   A new voucher dode is automatically displayed in the field.

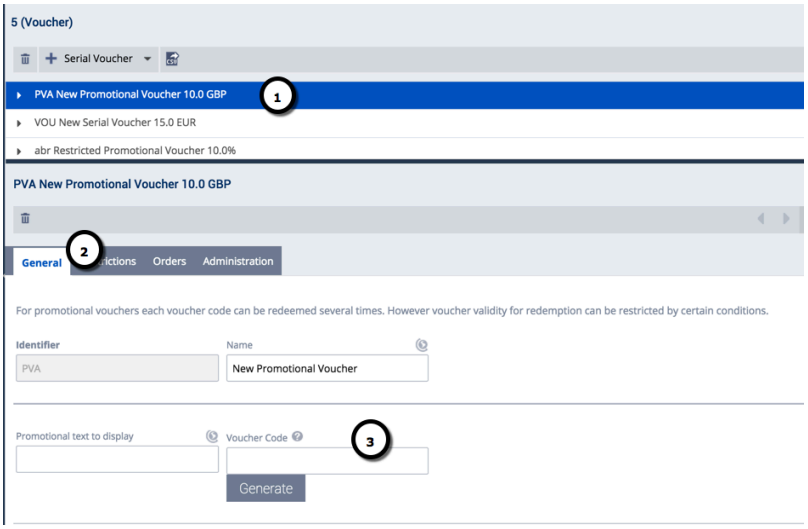
4. Click **Save**.



Generating Serial Vouchers

To generate a serial voucher, do the following:

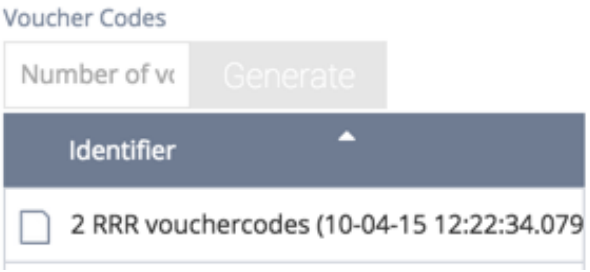
1. Select the desired serial voucher and go to **General** tab
2. Scroll down to **Voucher Code** field.



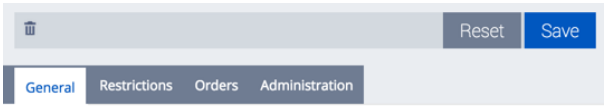
3. Enter the number of voucher codes to be generated.

4. Click **Generate**.

   The voucher codes are generated.

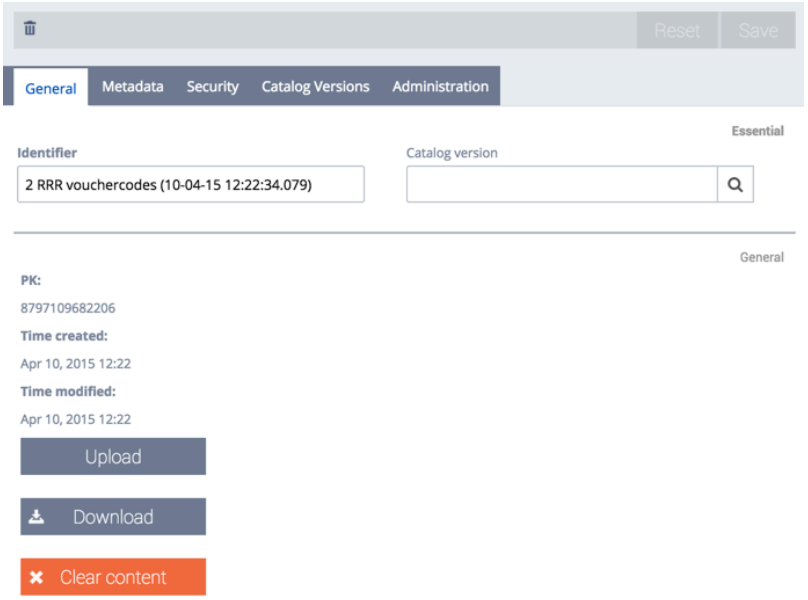


5. Click **Save**.



6. Double-click the saved voucher codes.

   A new window will appear i which you can see the details of the given code.



Related Information

Next Generation Cockpit Framework Admin Area
voucher Extension

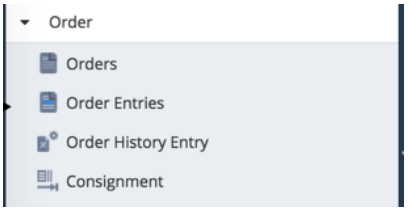
Redeeming and Releasing Vouchers Using the Backoffice Administration Cockpit

The process of assigning a voucher to an order (or a cart) is referred to as redeeming a voucher. The process of removing a redeemed voucher from an order (or a cart) is referred to as releasing the voucher.The normal way for vouchers to be redeemed is by a customer as part of the order process in the store frontend. There may be circumstances, however, when voucher redemption or release has to be handled in the Backoffice Administration Cockpit.

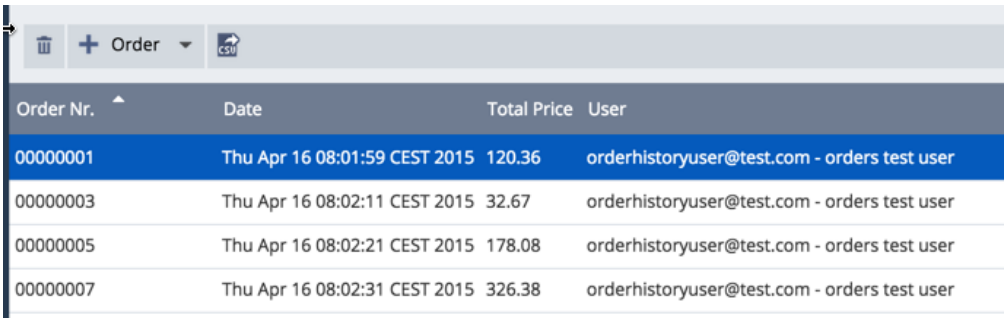
Redeeming Vouchers

To redeem a voucher, do the following:

1. Go to the Order node and select Orders.

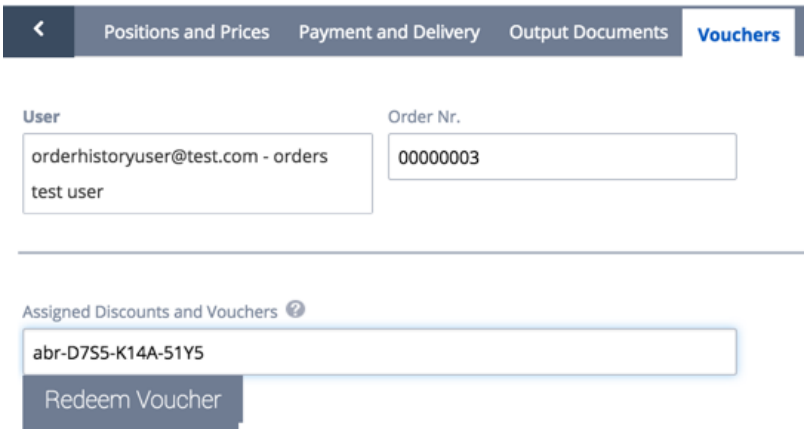


2. Double-click the desired Order in the Results section.

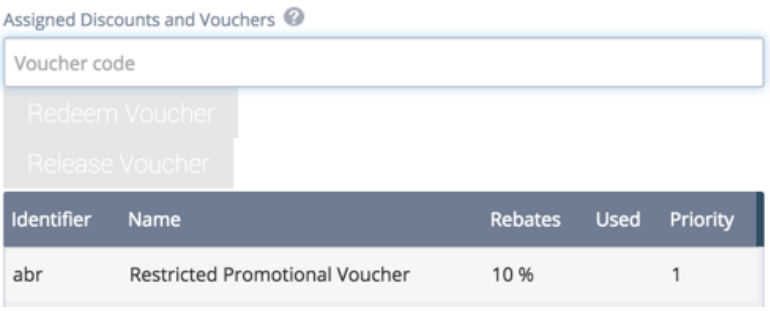


3. Go to the Vouchers tab.

4. Enter the voucher code into the text field in the Assigned vouchers section and click on the Redeem Voucher button.



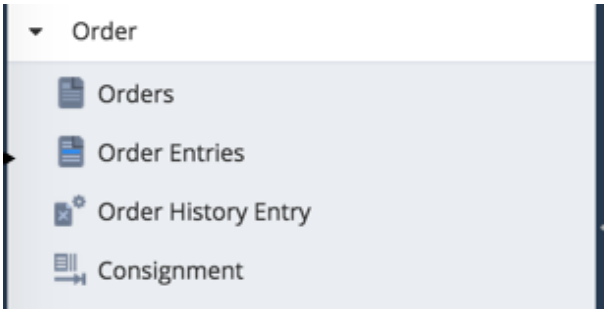
The voucher is checked whether it fulfills all Restrictions and is redeemed. Redeemed vouchers are presented.



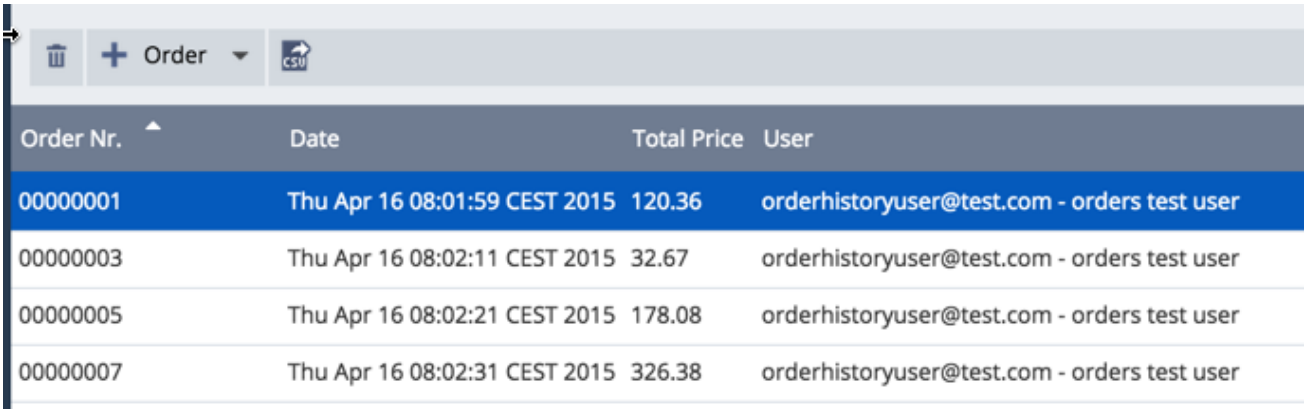
Releasing Vouchers

To release a voucher, do the following:

1. Go to the Order node and select Orders.

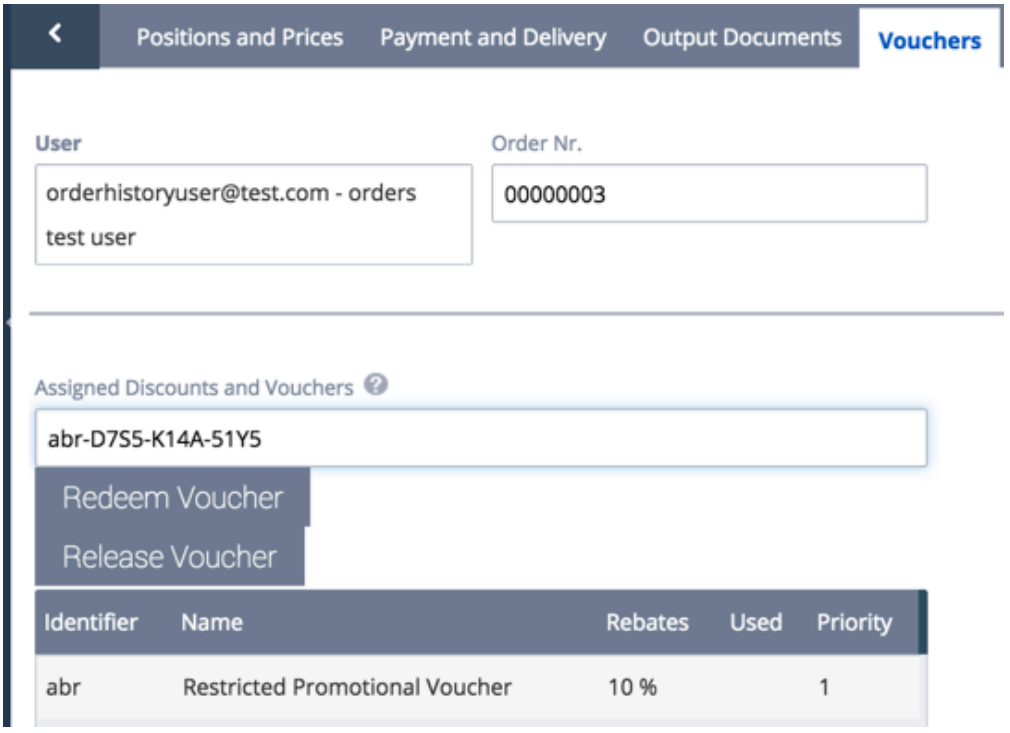


2. In the Results section, double-click the desired Order.

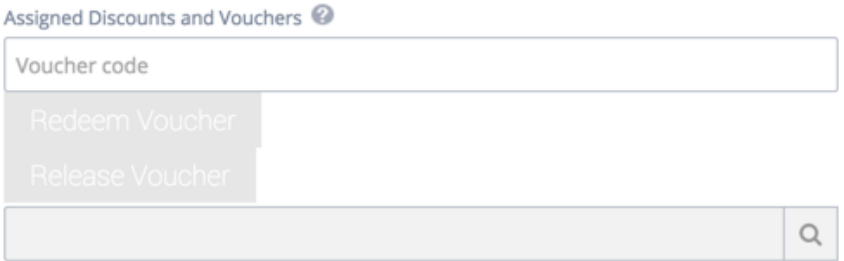


3. Go to the Vouchers tab.

4. In the Assigned vouchers section, enter the voucher code in the text field and click on the Release Voucher button.



5. The voucher is checked to verify that it fulfills all restrictions and is thenreleased.



Related Information

Creating Vouchers Using the Backoffice Administration Cockpit

Voucher Restrictions

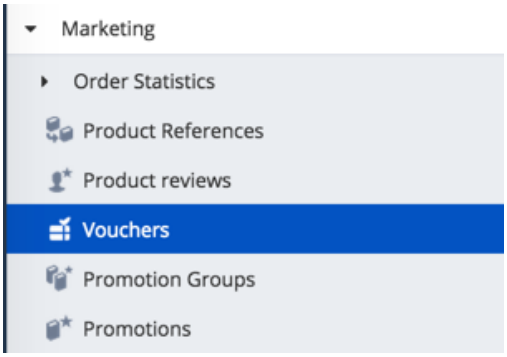
The Voucher module provides predefined restriction types. You can create restrictions for serial and promotional vouchers inBackoffice Administration Cockpit.

The Voucher module includes a list of predefined restriction types. All restrictions of a voucher must be fulfilled. Restrictions can be inverted through the `Valid` attribute where sensible. This document describes how to create voucher restrictions in Backoffice Administration Cockpit. For a list of available restrictions, see Voucher Restriction Types.

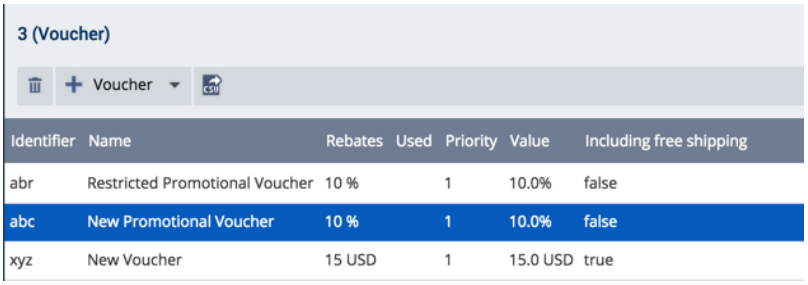
Creating Voucher Restrictions in Backoffice Administration Cockpit

To create a voucher restriction in Backoffice Administration Cockpit, do the following:

1. Go to **Marketing** and select **Vouchers**.



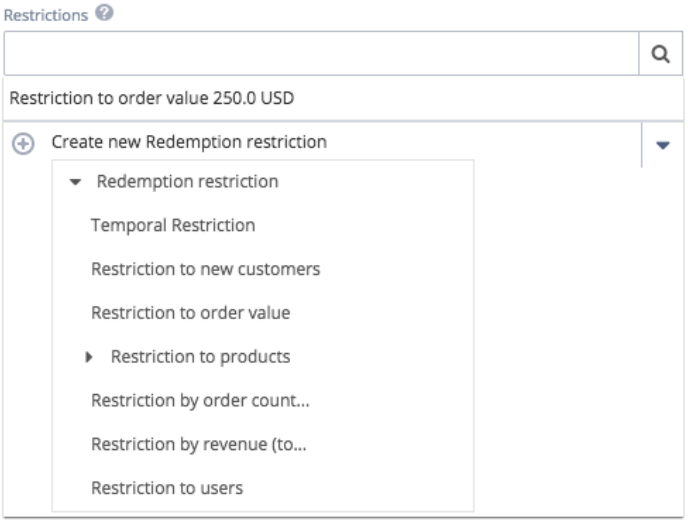
2. In the **Results** section, select a **voucher** and double-click it to view its details.



Identifier	Name	Rebates	Used	Priority	Value	Including free shipping
abr	Restricted Promotional Voucher	10 %		1	10.0%	false
abc	New Promotional Voucher	10 %		1	10.0%	false
xyz	New Voucher	15 USD		1	15.0 USD	true

3. Go to the **Restrictions** section.

4. Use the drop-down menu to select a **Restriction Type**.



Note

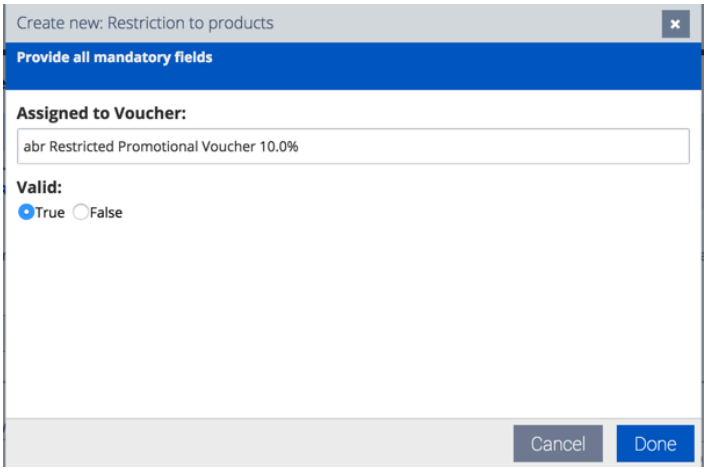
You can also add restrictions while creating vouchers. For details see: Creating Vouchers Using the Backoffice Administration Cockpit.

A new wizard window opens for each selected restriction.

5. Enter the attribute values to create a restriction (for example, **Restriction to Products**).
6. Click **Save** to create a restriction.

Note

The wizard for creating **restrictions** includes only mandatory attributes. Double-click the already created restriction to enter **Edit** mode and add optional attributes.

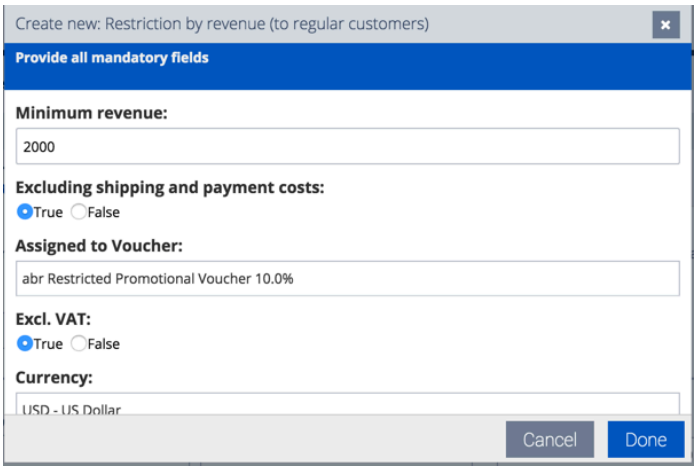


Voucher Restriction Types

The Voucher module comes with a number of pre-implemented restriction types. In order for a voucher to be redeemed, all restrictions of that voucher must be fulfilled. You can add restrictions for selected vouchers using Backoffice Administration Cockpit. If you only want mandatory attributes, then you have to use a wizard to create the restriction. To add optional attributes, double-click the already created restriction.

Restriction by Order Count (Regular Customers)

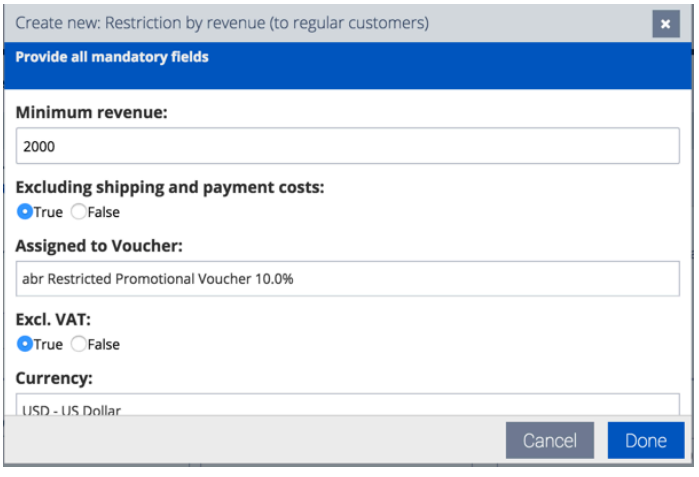
When you redeem the voucher, you must place the specified number of orders, or more, if needed.

Backoffice	

Attribute Name	Attribute Type	Mandatory	Description
Assigned to Voucher	Voucher	yes	The Voucher the Restriction is assigned to.
Description	localized String	no	Description of the voucher.
Minimum number of orders	number value as String	yes	How many orders (whatever the total) the current customer must have placed for the Restriction to be fulfilled.
Violation Message	localized String	no	A String to be displayed when the Restriction is violated (not fulfilled).

Restriction by Revenue (Regular Customers)

When you redeem the voucher, you must place orders for a total of the specified value, or more, if needed.

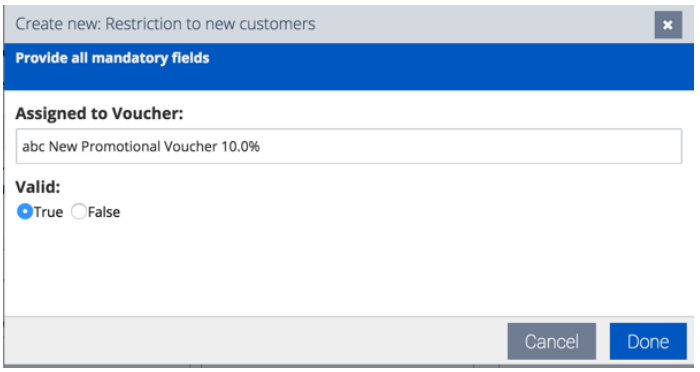
Backoffice	

Attribute Name	Attribute Type	Mandatory	Description
Assigned to Voucher	Voucher	yes	The Voucher the Restriction is assigned to.
Description	localized String	no	Description of the voucher.

Attribute Name	Attribute Type	Mandatory	Description
Minimum revenue	number value as String	yes	The total sum of all orders the current customer must have placed for the Restriction to be fulfilled.
Currency	Currency	yes	The currency in which **Minimum revenue** is measured.
Excl. VAT	Checkbox	no	Whether **Minimum revenue** is specified as net (checked) or as a gross value (unchecked).
Excl. shipping and payment costs	Checkbox	no	Whether **Minimum revenue** is specified including (unchecked) or excluding (checked) shipping costs.
Violation Message	localized String	no	A String to be displayed when the Restriction is violated (not fulfilled).

Restriction to New Customers

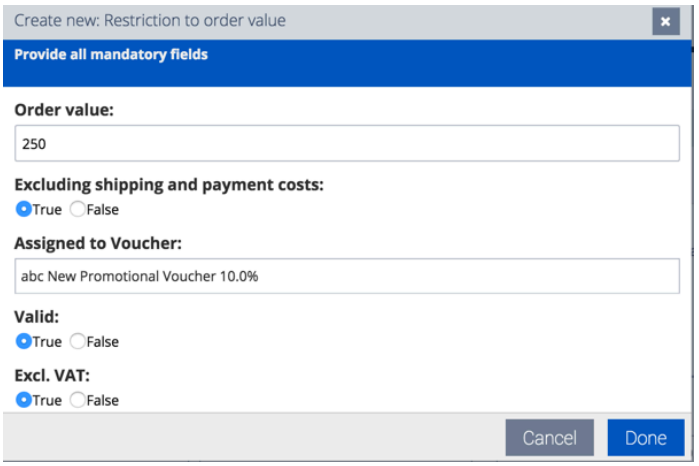
When you try to redeem the voucher, you must not have placed any orders yet.

Backoffice	

Attribute Name	Attribute Type	Mandatory	Description
Assigned to Voucher	Voucher	yes	The Voucher the Restriction is assigned to.
Description	localized String	no	Description of the voucher.
Violation Message	localized String	no	A String to be displayed when the Restriction is violated (not fulfilled).

Restriction to Order Value

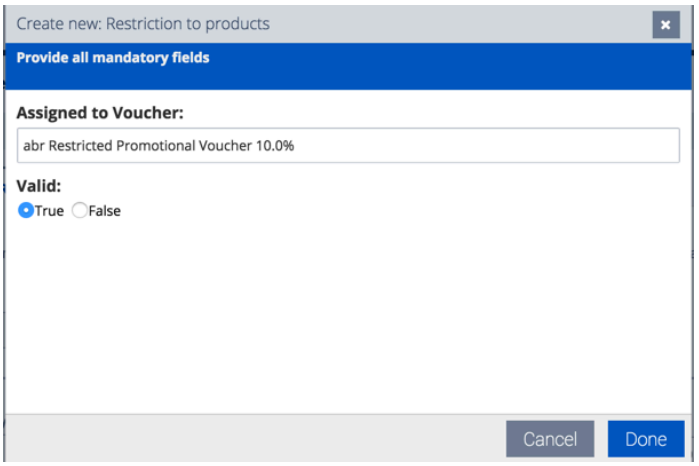
The current order has to exceed the value specified by **Order Value**.

Backoffice	

Attribute Name	Attribute Type	Mandatory	Description
Assigned to Voucher	Voucher	yes	The Voucher the Restriction is assigned to.
Description	localized String	no	Description of the voucher.
Order Value	String	yes	The value of the order that must be exceeded for the Restriction to be fulfilled.
Currency	Currency	yes	The currency in which **Order Value** is measured.
Excl. VAT	Checkbox	no	Whether **Order Value** is specified as net (checked) or as a gross value (unchecked).
Excl. shipping and payment costs	Checkbox	no	Whether **Order Value** is specified including (unchecked) or excluding (checked) shipping costs.
Valid	Radio button	no	Whether **Order Value** must or must not be exceeded for the Restriction to be fulfilled.
Violation Message	localized String	no	A String to be displayed when the Restriction is violated (not fulfilled).

Restriction to Products

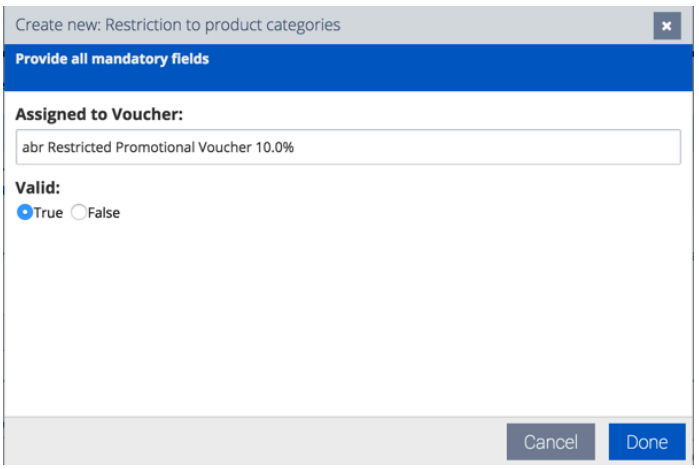
The product has to be or must not be on a specific list, depending on the value of the **Valid** attribute.

Backoffice	

Attribute Name	Attribute Type	Mandatory	Description
Assigned to Voucher	Voucher	yes	The Voucher the Restriction is assigned to.
Description	localized String	no	Description of the voucher.
Products	list of products	no	The list of products to be included or excluded (depending on the value of the **Valid** attribute)
Valid	Radio button	no	Specifies whether the current must be or must not be one of the product(s) listed in the **Products** attribute for the Restriction to be fulfilled.
Violation Message	localized String	no	A String to be displayed when the Restriction is violated (not fulfilled).

Restriction to Product Categories

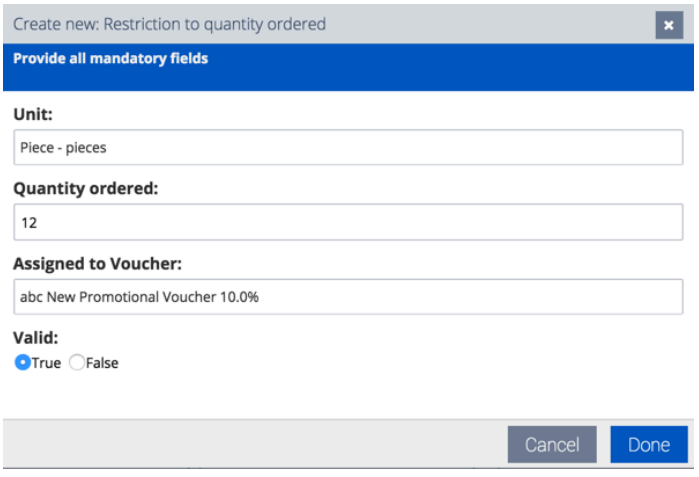
The product has to be in one of the specified categories.

Backoffice	

Attribute Name	Attribute Type	Mandatory	Description
Assigned to Voucher	Voucher	yes	The Voucher the Restriction is assigned to.
Description	localized String	no	Description of the voucher.
Product categories	list of categories	no	A list of categories which the product must be in or must not be in, depending on the **Valid** attribute.
Valid	Radio button	no	Specifies whether the product must be in the categories listed or not be in the categories listed.
Violation Message	localized String	no	A String to be displayed when the Restriction is violated (not fulfilled).

Restriction to Quantity Ordered

The ordered quantity of a product has to be equal to or less than the specified value or greater than the specified value, depending on the **Valid** attribute. This restriction limits the effect of the voucher to the number of items specified. For example, in the screenshot below, the voucher is effective for the first five items of the listed products. If a customer buys 20 items, the voucher is only applied to five items.

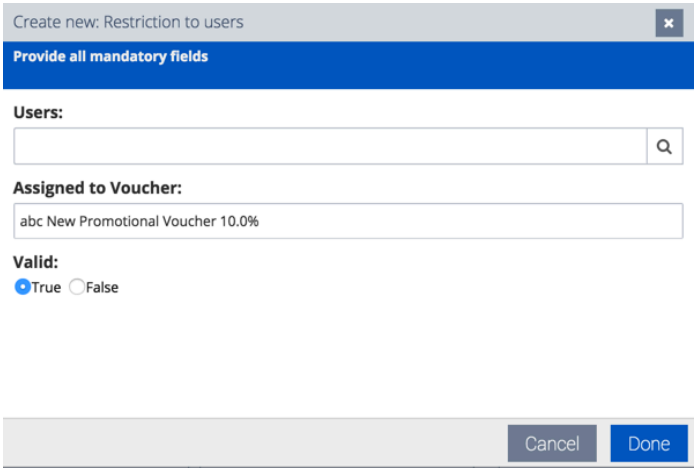
Backoffice	

Attribute Name	Attribute Type	Mandatory	Description
Assigned to Voucher	Voucher	yes	The Voucher the Restriction is assigned to.
Description	localized String	no	Description of the voucher.
Products	list	yes	Products affected by the restriction.
Quantity ordered	number value as String	yes	Text representation of the number of units that must at least or at most (depending on the **Valid** attribute) have been

Attribute Name	Attribute Type	Mandatory	Description
			ordered for the Restriction to fulfill.
Unit	a unit	yes	Unit of the **Quantity ordered** attribute.
Valid	Radio button	no	Specifies whether the number of units is given as a maximum or as a minimum.
Violation Message	localized String	no	A String to be displayed when the Restriction is violated (not fulfilled).

Restriction to Users

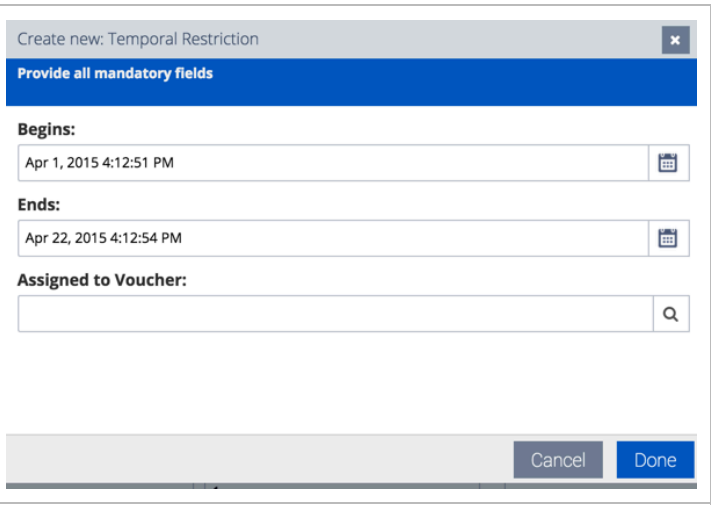
The user trying to redeem the voucher has to be on a specific list.

Backoffice	

Attribute Name	Attribute Type	Mandatory	Description
Assigned to Voucher	Voucher	yes	The Voucher the restriction is assigned to.
Description	localized String	no	Description of the Voucher.
Users	list of users	yes	A list of users the current user must be or must not be (depending on the **Valid** attribute) identical with for the Restriction to be fulfilled.
Valid	Radio button	no	Specifies whether the current user must be or must not be one of the users listed in the **Users** attribute for the Restriction to be fulfilled.
Violation Message	localized String	no	A String to be displayed when the Restriction is violated (not fulfilled).

Temporal Restriction

The current date has to be within the date range between **Begins** and **Ends**.

Backoffice	

Attribute Name	Attribute Type	Mandatory	Description
Assigned to Voucher	Voucher	yes	The Voucher the Restriction is assigned to.
Description	localized String	no	Description of the Voucher.
Begins	java.util.Date	yes	The beginning of the date range to which the Restriction applies.
Ends	Date	yes	The end of the date range to which the Restriction applies.
Violation Message	localized String	no	A String to be displayed when the Restriction is violated (not fulfilled).

Related Information

Voucher Technical Details

Voucher Technical Details

Learn about the technical details of the voucher types, how to create and manage them using the SAP Commerce API.

Generating Voucher Codes

To redeem a voucher, the voucher must have a code. The voucher class provides the generateVoucherCode() method, which is an implementation of the voucher code generator.

The generateVoucherCode() method generates hyphen-delimited voucher codes of 15 bytes according to the XXX-YYYY-YYYY-YYYY pattern (one group of three bytes and three groups of four bytes each).

Example: 123-H8BC-Y3D5-34AL.

ℹ Note

For more information on creating vouchers using the Backoffice Administration Cockpit, see Creating Vouchers Using the Backoffice Administration Cockpit.

The first three characters of the voucher codes generated by the generateVoucherCode() method are the **Promotion Code**, used as a prefix.

Example: Using the Vouchers **123**, **234** and **345** results in voucher codes such as **123-YYYY-YYYY-YYYY**, **234-YYYY-YYYY-YYYY** and **345-YYYY-YYYY-YYYY**.

The four byte groups (YYYY-YYYY-YYYY, such as **H8BC-Y3D5-34AL**) are generated basing on the **int** value provided by the getNextVoucherNumber(ctx) method. The generation mechanism does not involve a random generator function, so providing the same **int** for a Voucher always results in generating the same voucher code.

ℹ Note

The Backoffice do not allow entering more than three characters for the **code** attribute. At the API level, trying to create a voucher with more than three characters as value for the **code** attribute will result in the following exception:

```
For a voucher the maximum number of digits for the promotion code is 3.
```

Redeeming Vouchers

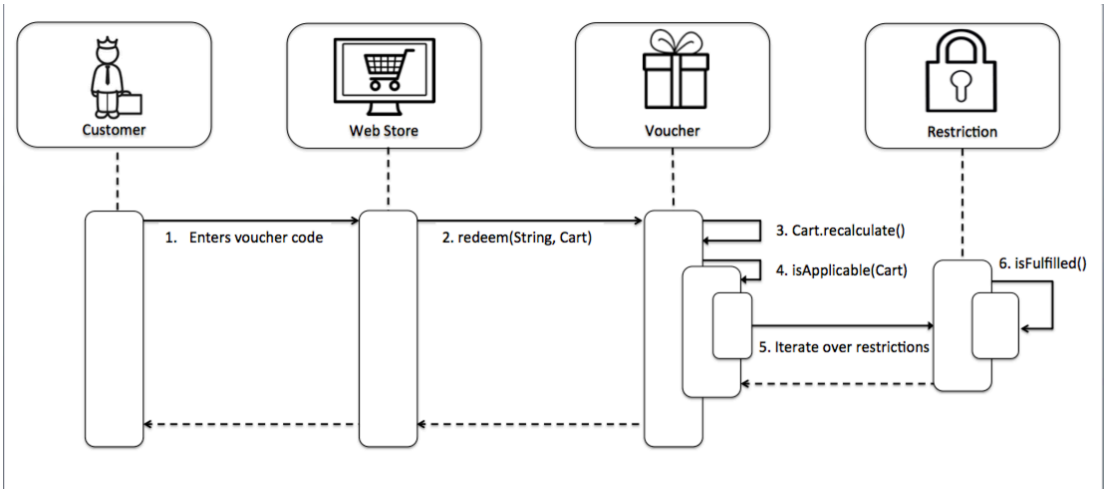
Redeeming a voucher is the process of activating the voucher for an order or a cart so that the customer receives a discount.

ℹ Note

For more information on releasing and redeeming vouchers using the Backoffice, seeRedeeming and Releasing Vouchers Using the Backoffice Administration Cockpit

The process is as follows:

1. The customer specifies a voucher code to the web store.



2. The voucher code is passed to the voucher class's `redeem( ...)` method.
3. The `redeem( ...)` method triggers the `recalculate()` method to make sure the cart is up to date. During the execution of the `recalculate()` method, the discounts (and, by consequence, vouchers) are calculated once more.

4. During voucher calculation, the `isApplicable(...)` method iterates over the voucher-related restrictions.
5. By calling each restrictions `isFulfilled()` method, the recalculation process checks whether or not a restriction denies the voucher.
6. If the `isFulfilled()` method returns `false` for one of the restrictions, the `isApplicable(...)` method also returns `false` and the voucher is not redeemed. For a voucher to be redeemed, every single voucher-related restriction must be fulfilled.

The voucher class offers two methods for voucher redemption, depending on if the voucher is to be redeemed for a cart or for an order:

- `redeem(String aVoucherCode, Cart aCart)`

  If both the `checkVoucherCode(String aVoucherCode)` and the `isReservable(String aVoucherCode, AbstractOrder abstractOrder)` methods return `true`, this method will add voucher to a Cart. Returns `true` if the redemption was successful, returns `false` if the redemption failed. Recalculates the Cart automatically.

  ```
  Voucher voucher = getVoucher( aVoucherCode );
  if( voucher != null )
  {
          return voucher.redeem( aVoucherCode, anOrder );
  }
  return null;
  ```

- `redeem(String aVoucherCode, Order anOrder)`

  Returns a `VoucherInvalidation` object if the redemption was successful. You will need to call the `recalculate()` method for the voucher redemption to take effect.

A `VoucherInvalidation` object establishes a connection between a Voucher, an Order and a User. If the `VoucherInvalidation` object exists for a combination of a voucher code, an Order, and a User, then the voucher code is redeemed for the User on the Order. Conversely, by removing the `VoucherInvalidation` object, the combination between a Voucher, an Order and a User is removed and the voucher no longer applies the voucher is released.

Releasing Vouchers

Releasing a voucher is the process of removing the voucher for an order or a cart so that the customer no longer receives a discount.

The voucher class offers two methods for voucher release, depending on whether the voucher is to be released for a cart or for an order:

- `release(String aVoucherCode, Cart aCart)`

  Removes the voucher with the code from the Cart. Recalculates the Cart automatically. Throws a JaloPriceFactoryException on errors.

  ```
  Voucher voucher = getVoucher( aVoucherCode );
  if( voucher != null )
  {
          voucher.release( aVoucherCode, anOrder );
  }
  return null;
  ```

- `release(String aVoucherCode, Order anOrder)`

Removes the voucher with the codefrom the Order. You will need to call the `recalculate()` method for the voucher release to take effect. Throws a ConsistencyCheckException on errors.

A `VoucherInvalidation` object establishes a connection between a Voucher, an Order and a User. If a`VoucherInvalidation` object exists for a combination of a voucher code, an Order, and a User, then the voucher code has been redeemed for the User on the Order. Conversely, by removing the `VoucherInvalidation` object, the combination between a Voucher, an Order and a User is removed and the voucher no longer applies the voucher is released.

Implementing Vouchers

To create custom voucher types, you will need to use one of the two pre-defined voucher-type implementations (`PromotionVoucher` and `SerialVoucher`), or create a non-abstract subtype of voucher.

To create a subtype of **Voucher**, you will need to implement the following three methods:

- `public boolean checkVoucherCode( String voucherCode)`

  Returns `true` if the String `voucherCode` passed as parameter is a valid voucher code for the voucher type. If `voucherCode` is not a valid voucher code, return `false`.

- `protected int getNextVoucherNumber( SessionContext ctx)`

  Must return an `int` that is used as a seed for the voucher code generation. Returning a fixed value (using `return 1;`, for example) causes the `generateVoucherCode()` method to always generate the same voucher code.

- `public boolean isReservable( String voucherCode, User user)`

  Return `true` if the voucher with the code `voucherCode` is eligible for the user `user`. The PromotionVoucher implementation of this method, for example, only returns `true` the voucher has been redeemed less than a given number of times overall and less than a given number of times for the individual user:

```
public boolean isReservable( String aVoucherCode, User user )
{
        return getInvalidations(aVoucherCode, user ).size() < getRedemptionQuantityLimitPerUs
                && getInvalidations( aVoucherCode ).size() < getRedemptionQuantityLimitAsPrim
}
```

Types and Attributes

This section describes type system-related information to create vouchers via the ImpEx extension, for example. The following tables provide the mandatory and some optional attributes of voucher-related types.

Unlike with almost all other types in the SAP Commerce, the length of the Voucher class `code` attribute (a promotion code) is technically limited to a maximum of three characters. The limitation to the three-character code is due to the fact that the promotion code is part of the voucher code by default.

ℹ Note

For more information on voucher types and attributes, see Voucher Types.

Serial Voucher

Unlike the `PromotionVoucher` type, the `SerialVoucher` type uses an internal counter which is increased on every voucher code generation. This counter allows up to 2^24 = 16.777.216 voucher codes per single `Serial Voucher.` To use more than this number of voucher codes, you will need to use different SerialVouchers.

The following list gives samples of SerialVoucher voucher codes as generated by the `generateVoucherCode()` method:

- 123-HYC5-A58D-DB4D
- 123-HYC5-AD45-575C
- 123-HYC5-A653-63A3
- 234-E452-GHWG-2E4W
- 234-E452-LWLR-WTL8
- 234-E452-L12W-TT5P
- 345-2TZG-SKEE-1WGZ
- 345-2TZG-S41W-1B1G
- 345-2TZG-SHZF-FFEF

Attribute Name	Attribute Type	Mandatory	Description
absolute	java.lang.Boolean	yes	Whether or not the `value` attribute specifies an absolute discount (reduction by a fixed value,`true`) or a relative discount (reduction by a certain percentage,`false`).
code	java.lang.String	yes	The identifier of the Voucher.
codes	MediaCollection	no	A list of medias containing the generated voucher codes.
description	localized:java.lang.String	no	The description of the voucher.
freeShipping	java.lang.Boolean	yes	Specifies if the order this voucher is applied to is shipped for free (`true`) or not (`false`). Defaults to `false`.
orders	OrderCollection	no	A list of all orders for which the voucher has been redeemed. Jalo-only attribute.
restrictions	VoucherRestrictionsRelationrestrictionsColl	no	References all restrictions that apply to the voucher.
valueString	java.lang.String	no	The value of this voucher to display. Jalo-only attribute.
value	java.lang.Double	yes	The numeric value of the voucher.

Promotion Voucher

`Promotion Vouchers` can optionally be redeemed more than once and / or more than once per customer. Whether or not a `PromotionVoucher` can be redeemed more than once depends on the values of the `redemptionQuantityLimit` and the `redemptionQuantityLimitPerUser` attributes, respectively.

The `PromotionVoucher` type always uses the same `int` value provided by the `getNextVoucherNumber( ctx)` method.Subsequently the voucher code for a `Promotion Voucher` never changes.

Attribute Name	Attribute Type	Mandatory	Description
absolute	java.lang.Boolean	yes	Whether or not the `value` attribute specifies an absolute discount (reduction by a fixed value, `true`) or a relative discount (reduction by a certain percentage, `false`).
code	java.lang.String	yes	The identifier of the Voucher.
description	localized:java.lang.String	no	The description of the voucher.
freeShipping	java.lang.Boolean	yes	Specifies if the order this voucher is applied to is shipped for free (`true`) or not (`false`). Defaults to `false`.
orders	OrderCollection	no	A list of all orders for which the voucher has been redeemed. Jalo-only attribute.
redemptionQuantityLimit	java.lang.Integer	yes	The maximum overall number of redemptions for this voucher.
redemptionQuantityLimitPerUser	java.lang.Integer	yes	The maximum number of redemptions for this voucher per user.
restrictions	VoucherRestrictionsRelationrestrictionsColl	no	References all `Restrictions` that apply to the voucher.
valueString	java.lang.String	no	The value of this voucher to display. Jalo-only attribute.
value	java.lang.Double	yes	The numeric value of the voucher.

Restrictions

DateRestriction

Unlike on the abstract Restriction type, the `positive` attribute is overridden to be optional.

Attribute Name	Attribute Type	Mandatory	Description
description	localized:java.lang.String	no	Description of the voucher.
endDate	java.util.Date	yes	The end of the date range to which the restriction applies.
positive	java.lang.Boolean	no	If set to `false`, the restriction is inverted. Defaults to `true`. The DateRestriction is fulfilled if • The current date is within the date range between `startDate` and `endDate`(`positive==true`) • The current date is outside the date range between `startDate` and `endDate`(`positive==false`)
restrictionType	localized:java.lang.String	no	Jalo-only attribute.
startDate	java.util.Date	yes	The beginning of the date range to which the restriction applies.
violationMessage	localized:java.lang.String	no	A string to be displayed when the restriction is violated (not fulfilled).
voucher	Voucher	no	Attribute created via the `VoucherRestrictionsRelation` relation.

NewCustomerRestriction

Attribute Name	Attribute Type	Mandatory	Description
description	localized:java.lang.String	no	Description of the voucher.
positive	java.lang.Boolean	yes	If set to `false`, the restriction is inverted. Defaults to `true`. The NewCustomerRestriction is fulfilled if • The current user account has no orders (`positive==true`) • The current user account has at least one order (`positive==false`).
restrictionType	localized:java.lang.String	no	Jalo-only attribute.

Attribute Name	Attribute Type	Mandatory	Description
violationMessage	localized:java.lang.String	no	A string to be displayed when the restriction is violated (not fulfilled).
voucher	Voucher	no	Attribute created using the `VoucherRestrictionsRelation` relation.

OrderRestriction

Attribute Name	Attribute Type	Mandatory	Description
currency	Currency	yes	The currency in which the value for the `total` attribute is specified
description	localized:java.lang.String	no	Description of the voucher.
net	java.lang.Boolean	yes	Specifies whether `total` is given as a net value (`true`) or as a gross value (`false`). Defaults to `true`.
positive	java.lang.Boolean	yes	If set to `false`, the restriction is inverted. Defaults to `true`. The OrderRestriction is fulfilled if: • the order has a higher total than the specified value (`positive==true`) • the order has a total equal to the specified value or less (`positive==false`).
restrictionType	localized:java.lang.String	no	Jalo-only attribute.
total	java.lang.Double	yes	The order must exceed this amount for the restriction to apply.
valueofgoodsonly	java.lang.Boolean	yes	Specifies whether the value of `total` refers to the total of the ordered articles only (`true`) or whether `total` refers to the ordered articles plus shipping etc (`false`). Defaults to `true`.
violationMessage	localized:java.lang.String	no	A String to be displayed when the restriction is violated (not fulfilled).
voucher	Voucher	no	Attribute created via the `VoucherRestrictionsRelation` relation.

ProductCategoryRestriction

Attribute Name	Attribute Type	Mandatory	Description
categories	CategoryCollection	yes	The categories the given product must be in or out of (depending on the value of the `positive` attribute).
description	localized:java.lang.String	no	Description of the voucher.
positive	java.lang.Boolean	yes	If set to `false`, the restriction is inverted. Defaults to `true`. The ProductCategoryRestriction is fulfilled if: • The product is in one of the specified categories (`positive==true`) • The product is in none of the specified categories (`positive==false`)
restrictionType	localized:java.lang.String	no	Jalo-only attribute.
violationMessage	localized:java.lang.String	no	A string to be displayed when the restriction is violated (not fulfilled).
voucher	Voucher	no	Attribute created using the `VoucherRestrictionsRelation` relation.

ProductQuantityRestriction

Attribute Name	Attribute Type	Mandatory	Description
description	localized:java.lang.String	no	Description of the voucher.
positive	java.lang.Boolean	yes	If set to `false`, the restriction is inverted. Defaults to `true`. The ProductQuantityRestriction is fulfilled if: • The ordered quantity of a product is equal to or less than the specified value (`positive==true`) • The ordered quantity of a product is greater than the specified value (`positive==false`)
quantity	java.lang.Long	yes	The order must exceed this amount of ordered items for the Restriction to apply.
restrictionType	localized:java.lang.String	no	Jalo-only attribute.

Attribute Name	Attribute Type	Mandatory	Description
unit	Unit	yes	The unit in which the value of the `quantity` attribute is specified.
violationMessage	localized:java.lang.String	no	A string to be displayed when the restriction is violated (not fulfilled).
voucher	Voucher	no	Attribute created via the `VoucherRestrictionsRelation` relation.

ProductRestriction

Attribute name	Attribute type	Mandatory	Description
description	localized:java.lang.String	no	Description of the voucher.
positive	java.lang.Boolean	yes	If set to `false`, the restriction is inverted. Defaults to `true`. The ProductRestriction is fulfilled if: • The product is a member of the specified list (`positive=true`) • The product is not a member of the specified list (`positive==false`)
restrictionType	localized:java.lang.String	no	Jalo-only attribute.
violationMessage	localized:java.lang.String	no	A string to be displayed when the restriction is violated (not fulfilled).
voucher	Voucher	no	Attribute created via the `VoucherRestrictionsRelation` relation.

RegularCustomerOrderQuantityRestriction

Attribute Name	Attribute Type	Mandatory	Description
description	localized:java.lang.String	no	Description of the voucher.
orderQuantity	java.lang.Integer	yes	The number of items the user needs to order for the restriction to be fulfilled. The effect depends on the `positive` attribute.
positive	java.lang.Boolean	no	If set to `false`, the Restriction is inverted. Defaults to `true`. The `RegularCustomerOrderQuantityRestriction` is fulfilled if the current user account has at least the specified number of orders.

Attribute Name	Attribute Type	Mandatory	Description
restrictionType	localized:java.lang.String	no	Jalo-only attribute.
violationMessage	localized:java.lang.String	no	A String to be displayed when the Restriction is violated (not fulfilled).
voucher	Voucher	no	Attribute created via the `VoucherRestrictionsRelation` relation.

RegularCustomerOrderTotalRestriction

Attribute Name	Attribute Type	Mandatory	Description
allOrdersTotal	java.lang.Double	yes	The total sum of all orders combined which the user must exceed for the restriction to be fulfilled.
currency	Currency	yes	The currency in which `allOrdersTotal` is specified.
description	localized:java.lang.String	no	Description of the voucher.
net	java.lang.Boolean	yes	Specifies whether `allOrdersTotal` is given as a net value (`true`) or as a gross value (`false`).
positive	java.lang.Boolean	no	If set to `false`, the restriction is inverted. Defaults to `true`. The RegularCustomerOrderTotalRestriction is fulfilled if the current user account has placed orders for a total of at least the specified value.
restrictionType	localized:java.lang.String	no	Jalo-only attribute.
valueofgoodsonly	java.lang.Boolean	yes	Specifies whether the value of `total` refers to the total of the ordered articles only (`true`) or whether `total` refers to the ordered articles plus shipping etc (`false`).
violationMessage	localized:java.lang.String	no	A String to be displayed when the restriction is violated (not fulfilled).
voucher	Voucher	no	Attribute created via the `VoucherRestrictionsRelation` relation.

UserRestriction

Attribute Name	Attribute Type	Mandatory	Description
description	localized:java.lang.String	no	Description of the voucher.

Attribute Name	Attribute Type	Mandatory	Description
positive	java.lang.Boolean	yes	If set to `false`, the Restriction is inverted. Defaults to `true`. The UserRestriction is fulfilled if: • The current user account is a member of the specified list (`positive==true`) • The current user account is not a member of the specified list (`positive==false`)
restrictionType	localized:java.lang.String	no	Jalo-only attribute.
users	PrincipalCollection	yes	The list of users of which the current user must (`positive == true` or must not (`positive == false`) be member for the restriction to be fulfilled.
violationMessage	localized:java.lang.String	no	A string to be displayed when the restriction is violated (not fulfilled).
voucher	Voucher	no	Attribute created via the `VoucherRestrictionsRelation` relation.

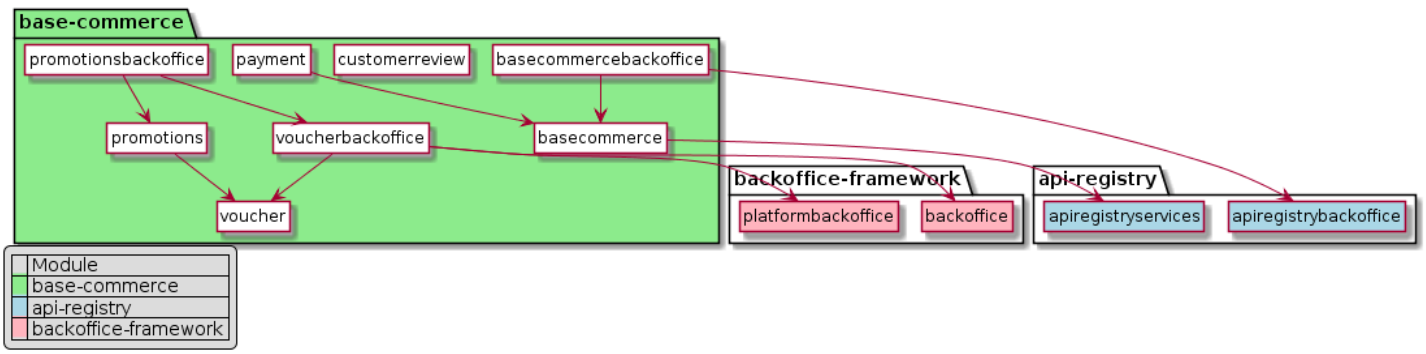
Related Information

voucher Extension
Voucher Types
Creating Vouchers Using the Backoffice Administration Cockpit
Redeeming and Releasing Vouchers Using the Backoffice Administration Cockpit
Next Generation Cockpit Framework Admin Area

Base Commerce Module Architecture

The Base Commerce module is a set of extensions providing a wide range of functions within SAP Commerce. These include payment, customer reviews, returns, refunds, store locator, stock service, and more.

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 Base Commerce module consists of the following extensions:

basecommerce Extension

The `basecommerce` extension provides a set of multi-purpose services used in SAP Commerce.

basecommercebackoffice Extension

The `basecommercebackoffice` extension provides settings for basecommerce features.

customerreview Extension

The `customerreview` extension provides functionality enabling front end users of a web shop based on the SAP Commerce StoreFoundation to give a rating and comment on offered products.

payment Extension

The `payment` extension constitutes the foundation for the payment framework in the SAP Commerce..

promotions Extension

The `promotions` extension consists of a code component that may be used to implement customer sales promotion functionality within web sites and UI components for creating and managing promotion data. The `promotions` uses the BaseStore type, which is a part of the basecommerce extension..

promotionsbackoffice Extension

The `promotionsbackoffice` extension depends on the `promotions` extension and adds additional configuration to the backoffice extension.

voucherbackoffice Extension

The `voucherbackoffice` extension depends on the `voucher` extension and adds additional configuration (related to vouchers) to the backoffice extension.

voucher Extension

The `voucher` extension enables you to create and manage vouchers redeemable by your customers.

basecommerce Extension

The `basecommerce` extension provides a set of multi-purpose services used in SAP Commerce.

About the Extension

Name	Directory	Related Module
`basecommerce`	`hybris/bin/modules/base-commerce/`	Base Commerce Module

Related Information

Customer Services
Deep Linking
Fraud Detection

basecommercebackoffice Extension

The `basecommercebackoffice` extension provides settings for basecommerce features.

About the Extension

Name	Directory	Related Module
`basecommercebackoffice`	`hybris/bin/modules/base-commerce/`	Base Commerce Module

customerreview Extension

The `customerreview` extension provides functionality enabling front end users of a web shop based on the SAP Commerce StoreFoundation to give a rating and comment on offered products.

About the Extension

Name	Directory	Related Module
`customerreview`	`hybris/bin/modules/base-commerce/`	Base Commerce Module

payment Extension

The `payment` extension constitutes the foundation for the payment framework in the SAP Commerce..

About the Extension

Name	Directory	Related Module
`payment`	`hybris/bin/modules/base-commerce/`	Base Commerce Module

promotions Extension

The `promotions` extension consists of a code component that may be used to implement customer sales promotion functionality within web sites and UI components for creating and managing promotion data. The `promotions` uses the BaseStore type, which is a part of the basecommerce extension..

About the Extension

Name	Directory	Related Module
promotions	hybris/bin/modules/base-commerce/	Promotions (Legacy)

Dependencies

The promotionsbackoffice extension adds additional configuration to the backoffice extension. It depends on the promotions extension.

promotionsbackoffice Extension

The promotionsbackoffice extension depends on the promotions extension and adds additional configuration to the backoffice extension.

About the Extension

Name	Directory	Related Module
promotionsbackoffice	hybris/bin/modules/base-commerce/	Promotions (Legacy)

Dependencies

The promotionsbackoffice extension depends on the promotions extension.

voucherbackoffice Extension

The voucherbackoffice extension depends on the voucher extension and adds additional configuration (related to vouchers) to the backoffice extension.

About the Extension

Name	Directory	Related Module
voucherbackoffice	hybris/bin/modules/base-commerce/	Vouchers (Legacy)

Dependencies

The voucherbackoffice extension depends on the voucher extension .

voucher Extension

The voucher extension enables you to create and manage vouchers redeemable by your customers.

About the Extension

Note

A new `voucherbackoffice` extension is available. This extension depends on the `voucher` extension and adds additional configuration to the backoffice extension.

Name	Directory	Related Module
`voucher`	`hybris/bin/modules/base-commerce/`	Vouchers (Legacy)