

OPP Development and Extension Guide



Document History

Version	Date	Change
1.0	2019-11-15	Initial version
1.1	2020-07-05	Minor changes in Formatting and version-related information

OPP Development and Extension Guide for CARAB 4.0

FP02/FP03

- Overview of Omnichannel Promotion Pricing
 - The Price and Promotion Repository (PPR)
 - The Promotion Pricing Service (PPS)
- Promotion Pricing Service Overview
 - Open Source Dependencies of the PPS
 - PPS Module Concept
 - Defining and Overriding Beans
 - PPS Context
 - PPS Validation
 - PPS-Specific Constraints
 - Adjusting Constraint Checks of the Standard Delivery
 - Enabling or Disabling Bean Validation within EclipseLink
 - Further information
 - PPS Module api
 - Overview
 - Beans
 - Required Beans
 - Configuration Properties
 - Dependencies
 - PPS Module client-interface
 - Overview
 - Extensibility via any Elements
 - XSD and Currencies
 - Beans
 - Configuration Properties
 - Dependencies
 - PPS Module core
 - Overview
 - PPS Application Context
 - PPS Context
 - Bean Validation
 - Beans
 - Configuration Properties
 - Dependencies
 - PPS Module dataaccess-interface
 - Overview
 - Beans
 - Configuration Properties
 - Dependencies
 - PPS Module jackson
 - Overview
 - Configuring Jackson (Client Side)
 - Request Logging
 - Beans
 - Configuration Properties
 - Dependencies
 - PPS Module restapi
 - Overview
 - Known Issues
 - Beans
 - Configuration Properties
 - Dependencies
 - PPS Module client-impl
 - Overview
 - Request Validation
 - Single vs Bulk Access for Regular Prices
 - Handling of Business Unit Type
 - Beans
 - Required Beans
 - Configuration Properties
 - Dependencies
 - PPS Module calcengine-gk
 - Overview
 - Beans
 - Default Settings and Properties
 - Required Beans
 - Configuration Properties
 - Dependencies
 - PPS Module dataaccess-common
 - Overview
 - Regular Price
 - Promotional Information
 - Object-Related Mapping Using Spring

- Multi-Step JPA Resource Mapping
 - Multi-Step JPA Property Definition
 - Support of JPA Entity Extensions
 - equals() and hashCode() for JPA Entities
 - Caching
 - Caching Regular Prices
 - Caching Promotional Information
 - Cache Keys
 - Prefetch of Price Derivation Rule Eligibility References
 - Support of Weaving
 - Support for Read-Only Transactions
 - Code Conversion
 - Handling of Currencies and Amounts
 - Handling Product IDs
 - Handling of Language-Specific Information
 - SAP Client and Logical System
- Beans
 - Required Beans
- Configuration Properties
- Dependencies
- PPS Module dataaccess-ddf
 - Overview
 - Attribute Converters
 - Boolean Values
 - Time Stamps
 - Beans
 - Configuration Properties
 - Dependencies
- PPS Module dataaccess-localdb
 - Overview
 - Indexes
 - Configuring the Data Access
 - Beans
 - Required Beans
 - Configuration Properties
 - Dependencies
- PPS Module idocinbound
 - Overview
 - Spring Integration Process Definition
 - Processing the IDoc Data
 - Conversion of the IDoc Payload to the Expected Java Types
 - Mapping Regular Prices
 - Mapping OPP Promotions
 - Validating Uploaded Data
 - Posting to the Database
 - Regular Prices
 - OPP Promotions
 - Beans
 - Required Beans
 - Configuration Properties
 - Dependencies
- PPS Performance Hints
 - Creating of the Offers
 - Distributing of the Data
 - Client Side (Price Calculation)
 - Client Side (Data Replication)
 - Server Side
 - Common Rules
 - Local-PPS-Specific
 - XSA-Based-PPS-Specific
 - Database Side
- PPS Logging and Tracing
- PPS Authentication
 - Enabling XSA Authentication
- Price and Promotion Repository
 - Overview
 - Modeling of OPP Promotions
 - Keys and Foreign Keys
 - Validity Period for the OPP Promotion
 - Database Tables
 - Handling of Amounts
 - Transformation from DDF offers into OPP Promotion
 - Technical Information
 - How We Transform DDF Offers into OPP Promotions
 - Transformation of Simple Discount Offers
 - Examples
 - Transformation of Offers with Transaction Discount
 - Transformation of Mix-and-Match Offers
 - Examples

- Transformation of Packaged Offers
 - Transformation of Offers with Incentives
 - Default Values
 - ItemPriceDerivationRule
 - Fields Only Relevant for Coupons
 - Fields Only Relevant for Loyalty Points
 - CouponPriceDerivationRule Eligibility
 - PromotionPriceDerivationRule
- Replication of the Price and Promotion Repository
 - Outbound Processing of IDocs via DRF
 - DRF Configuration
 - OPP Promotions
 - Outbound Implementation for Promotion-Centric Outbound Processing
 - Outbound Implementation for Location-Specific Outbound Processing
 - Filtering the OPP Promotions
 - Controlling the Target Locations
 - Generic Mapping of Customer Enhancement Segments
 - Transfer OPP Promotions Using the Global Object List
 - Location-Specific Outbound Processing Using the Global Object List
 - Cleanup of the Global Object List
 - Regular Prices
 - Outbound Implementation
 - Data Filtering
 - Handling of the Expected Data Volume
- OPP Extensibility
 - Extensibility of Demand Data Foundation (DDF)
 - Extensibility of DDF Offer Inbound API
 - Extensibility of DDF Regular Price Inbound API
 - Extensibility of the OPP Data Model (ABAP)
 - Extending SAP delivered ABAP domains
 - Extensibility of the OPP Business Logic (ABAP)
 - Extensibility of the Transformation from DDF Offer into OPP Promotion
 - Extensibility of the IDoc Outbound Processing (ABAP)
 - Extensibility of the OPP Data Model (Java)
 - Adding a Field to an Entity
 - Adding a Separate Entry
 - Adding an Attribute Converter to an Existing Attribute
 - Adding a Subentity to an Existing Entity
 - Adding a Specialization to an Existing Entity
 - Using Own Logic for Equals() and GetHashCode() of a JPA Entity
 - Extensibility of Client API (Java)
 - Extensibility of Enumerations
 - Extensibility of Content with User-Defined Attributes / Elements
 - Restrictions
 - Example: Enrich SaleForDelivery Entity with Address Information
 - Extending the PPS Business Logic (Java)
 - Plugin Concept
 - Calling the Plugins
 - Implementing a Plugin
 - Guaranteed Stability
 - Documented Stability
 - Your Choices for Extending the PPS Java Side
 - SAP Delivered Plugin Implementations
 - Structure of Your Extension Project
 - Installing your Extensions
 - Extensibility of the Promotion Calculation Engine (Java)
 - Extensibility of the sappspricing PPS Integration (Java)
 - Extensibility Examples
 - Integrating Custom Extensions into the XSA-Based-PPS
 - Setting Up the Development Environment
 - Creating Your Extension Projects
 - Adding Your Extension to the PPS
 - Extending the PPS-Based Price Calculation in SAP ERP and SAP S/4HANA Sales Documents
 - Extending via BAaIs
 - Enriching with Further Article Hierarchy Nodes
 - Extending the SAP ERP/ SAP S/4HANA PPS Client
 - Support for Mocking of the SAP ERP/ SAP S/4HANA PPS Client

Overview of Omnichannel Promotion Pricing

Omnichannel promotion pricing provides the **price and promotion repository (PPR)** as central storage of regular prices and price rules as well as the **promotion pricing service (PPS)** to calculate effective sales prices by applying promotional rules.

The Price and Promotion Repository (PPR)

The **PPR** is part of SAP Customer Activity Repository (SAP CAR) based on SAP HANA. This repository contains regular prices and price rules, so-called offers. Technically, this data is located in the Demand Data Foundation (DDF) software component. This data is also used by SAP Promotion Management (SAP PM), which is an optional add-on to SAP CAR.

For the maintenance of price rules, OPP reuses the existing options: SAP PM and SAP Fiori UI-based offer maintenance. For the import of price rules and regular prices from external systems, an RFC-based import via standard DDF interface can be used.



In an SAP environment, regular prices are typically imported from an SAP ERP or an SAP S/4HANA system. However, these systems are not mandatory components for the usage of OPP.

To make offers consumable for the promotion pricing service, they are transformed into a different data format based on the [ARTS promotion data model](#). An offer with this format is called **OPP promotion**. OPP promotions have similar, but not identical, features to define price rules compared to the DDF offer. If the offer can be transformed into an OPP promotion, an automatic compatibility check is performed before the offer is saved. The status of the offer determines whether the compatibility check and the transformation can be performed. Offers with status *In Process* are not considered for the transformation into OPP promotions.



You have to distinguish clearly between the repository view of a price rule (the DDF offer) and the runtime view (the OPP promotion).

Regular prices are consumed by the promotion pricing service using the database view.

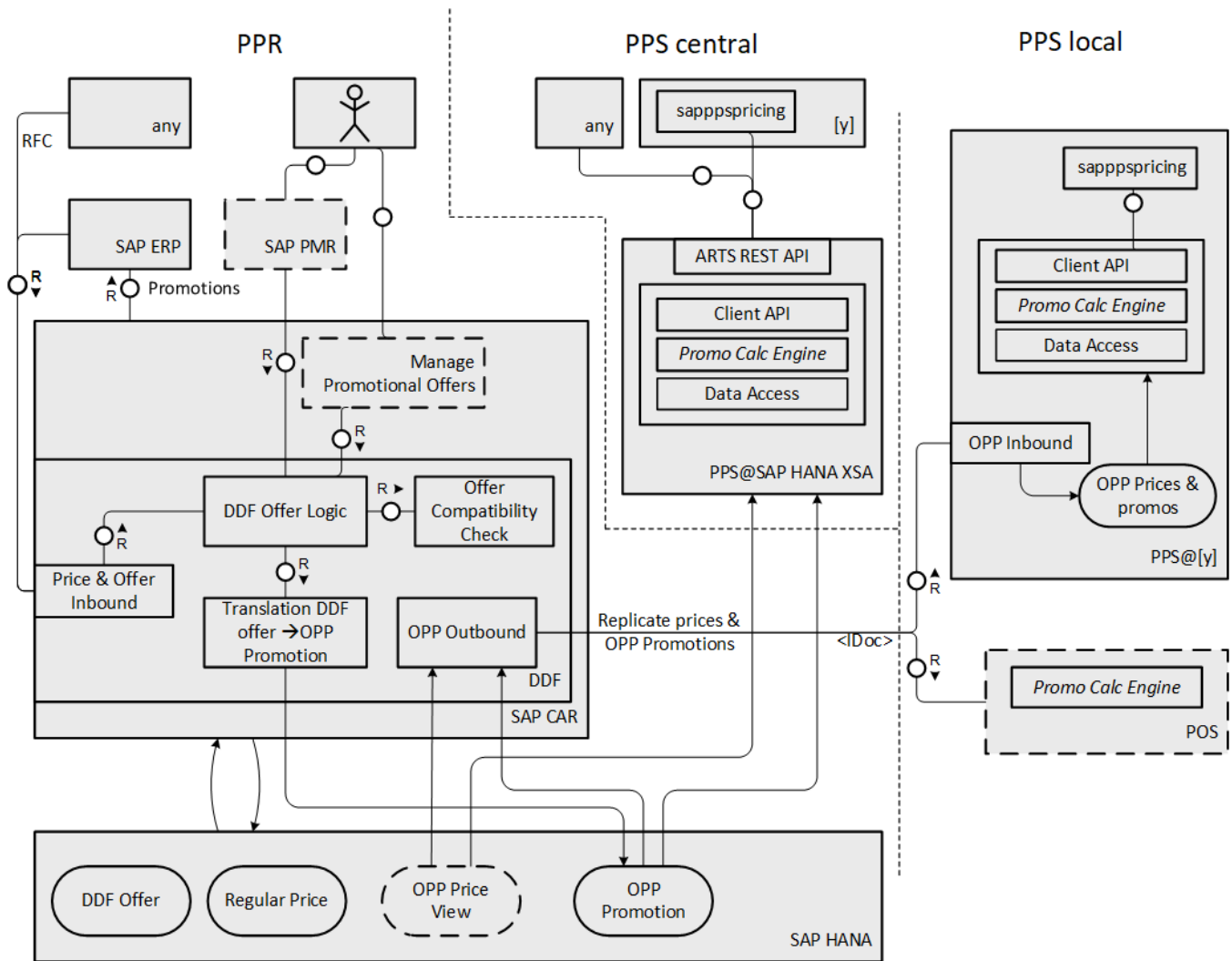
The Promotion Pricing Service (PPS)

The **PPS** uses a Java-based engine to calculate prices and promotions. This ensures high system performance and a flexible deployment. The service can be deployed centrally or locally:

- With a central deployment, the promotion pricing service runs on the central price and promotion repository, and is powered by SAP HANA XS advanced (XSA).
- With a local deployment, the promotion pricing service runs locally in the database of the corresponding sales channel application. Therefore, regular prices and OPP promotions are replicated from the central price and promotion repository to the database of the sales channel application, such as web applications and POS systems. The replication is done using the data replication framework (DRF), which is a central reuse component in SAP Business Suite. Delta transfer is also supported. In this way, the sales channel application can work without an additional remote system being continuously available.

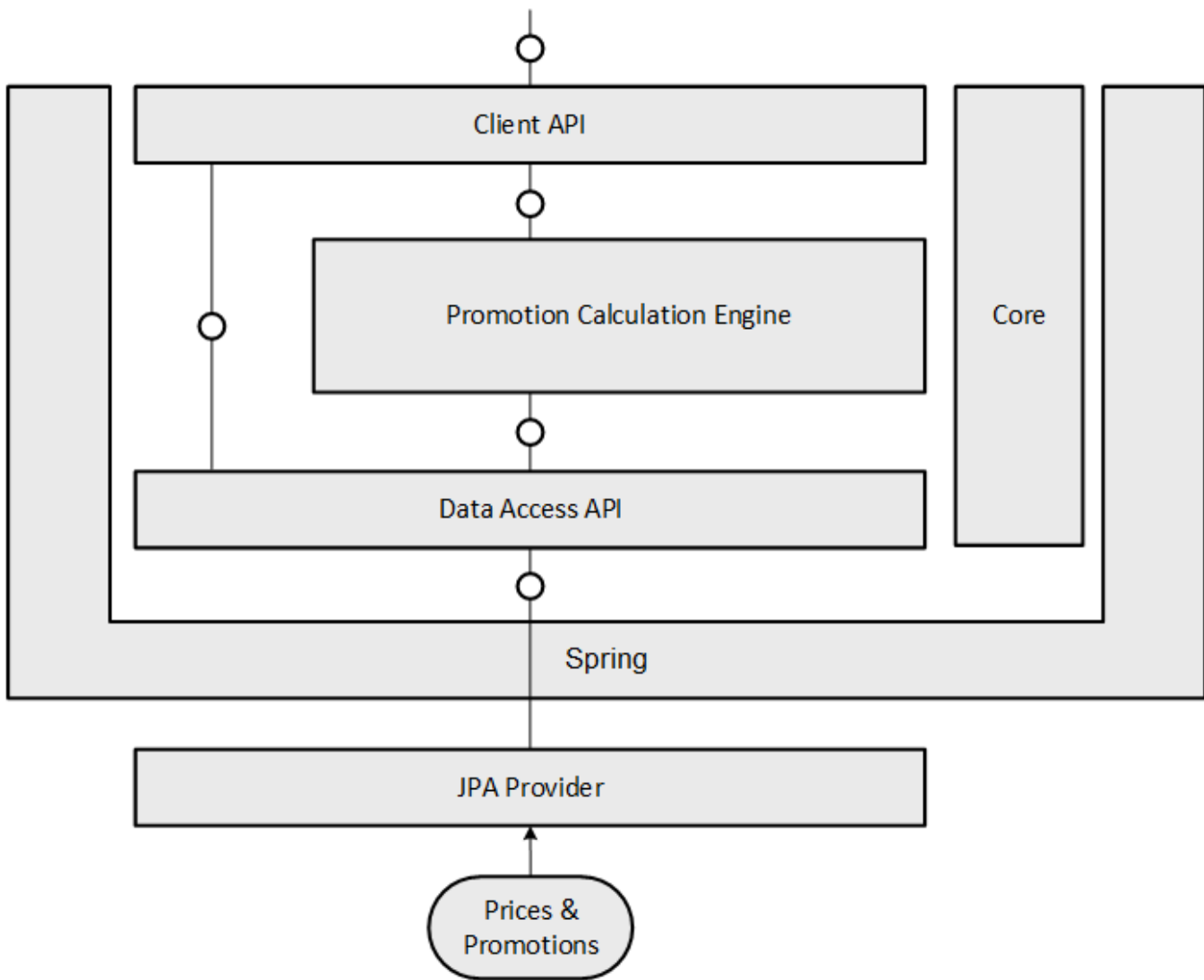
OPP provides the local and the central promotion pricing service for SAP Commerce and also the GK POS solution is based on the same concepts than the promotion calculation engine. The calculation is exposed using a stateless service, which is based on the ARTS Price Service Interface 1.0. This interface is consumed by SAP Commerce using an additional extension (**sappspricing**) but can be used by any other client. For a local deployment, embedded into SAP Commerce, the service uses the same database as SAP Commerce.

The following figure illustrates the OPP architecture:



Promotion Pricing Service Overview

This chapter describes the promotion pricing service (PPS), in particular its concepts and its structure. The figure below shows the inner structure of the PPS in more detail and the data flow of a price calculation request:



The PPS is an application that exposes an API based on the ARTS Pricing Service Interface 1.0. The structure of the requests is defined in the client API layer. The requests are forwarded to the client API implementation layer where the regular prices are determined using the data access API. Next, the request is forwarded to the promotion calculation engine that calculates discounts, and so on. The data access API is used again to read data from the persistence. The implementation based on Java Persistence API (JPA) calls the underlying database in which the regular prices and OPP promotions are stored. PPS core functions are available throughout the application. Spring framework is used extensively to assemble the different parts and configure the PPS.

Open Source Dependencies of the PPS

The PPS uses various open-source libraries. The following list only contains some of the used libraries as well as more information about what they are used for:

- Spring framework for dependency injection, transaction management, cache abstraction, and so on
- EclipseLink as a JPA implementation
- FasterXML Jackson for unmarshalling/marshalling HTTP requests (such as IDoc inbound processing)
- Woodstox as a Stax XML API implementation
- SLF4J as a logging facade
- Google Guava as a general purpose toolkit and, in particular, as a cache implementation (for named queries)
- Various parts of Apache Commons as a general purpose toolkit:
 - commons-lang
 - commons-lang3
 - commons-logging
 - commons-collections3
- Joda Time as an alternative for Java date and time classes

PPS Module Concept

The business logic of the PPS is implemented by Spring beans. To support extensibility, the PPS comes with its own lightweight module concept that uses Spring concepts. A PPS module is just a set of Spring beans, which are added to the application context during its initialization. From a business perspective, a PPS module should contain Spring beans that are used to implement a - potentially large - functional block.

A PPS module, such as M1, can have dependencies to other PPS modules, such as M2 and M3. In this case, the beans of module M1 are added to the Spring application context after the beans of modules M2 and M3. In this way, the M1 beans could hide or redefine the M2 or M3 beans. If you want to enhance business logic in a customer project, the corresponding Spring bean (in module sapABC, for example) can be hidden by a customer-specific bean. This can be done without modification of the standard shipment by adding a further module (custXYZ, for example) that depends on the module sapABC. A new bean with the same bean alias (see below) can be created in this module.

A PPS module is defined as follows.

1. Create the file `META-INF/<moduleName>-ppe-module-metadata.xml` on the PPS classpath. The following example shows the structure of this file type:

Module declaration

```
<module xmlns="http://www.sap.com/ppengine/core/module"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.sap.com/ppengine/core/module ppende-module-0.2.xsd">
  <name name="client-impl" vendor="sap" />
  <dependencies>
    <module name="dataaccess-interface" vendor="sap"/>
    <module name="client-interface" vendor="sap"/>
    <module name="core" vendor="sap"/>
  </dependencies>
</module>
```

The combination of the name `client-impl` with the vendor name is the PPS module name. This module depends on three further modules: core, client-interface, and dataaccess-interface, all with the vendor "sap". The purpose of the vendor attribute is to avoid name collisions. Modules delivered by SAP have the vendor "sap".

2. To enable schema validation, place the file `ppengine-module-0.2.xsd` in the same folder as the module metadata file.
3. Create an XML file with the name **(META-INF)<moduleName>-ppe-module-spring.xml** in the same folder as the metadata file. This contains the Spring beans. Below is an excerpt from the SAP bean definitions:

Spring beans of a module

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org
/schema/context"
  xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:tx="http://www.springframework.org/schema
/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema
/mvc/spring-mvc-4.1.xsd
  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
beans-4.1.xsd
  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.1.
xsd
  http://www.springframework.org/schema/context http://www.springframework.org/schema/context
/spring-context-4.1.xsd">

  <context:property-placeholder
    location="classpath:/META-INF/client-impl-ppe-module.properties"
    ignore-unresolvable="true" />

  <!-- Validator for a price calculation request -->
  <alias name="sapDefaultCalculateRequestValidator" alias="sapCalculateRequestValidator" />
  <bean id="sapDefaultCalculateRequestValidator" class="com.sap.ppengine.client.impl.
RequestValidatorImpl">
    <property name="objectFactory" ref="sapClientApiDtoFactory" />
  </bean>

  <!-- Further beans below -->

</beans>
```

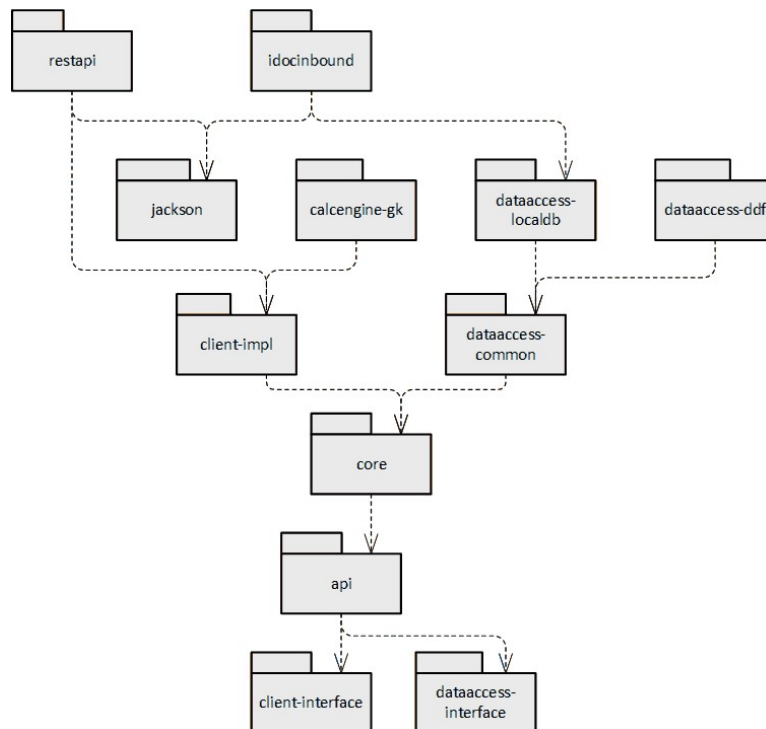
When the PPS is started, the PPS classpath is scanned for all modules. All the modules that are found are loaded automatically in the correct order.

A third, optional, part of a PPS module is a Java .properties file that holds default values for Spring properties used during the definition of the Spring beans. By convention, this is located in the same folder as the metadata and resource file of the module and has the name **META-INF/<moduleName>-ppe-module.properties**. It is loaded via the **<context:property-placeholder>** tag in the corresponding resource file. Defaults stored in this file can either be changed using one dedicated file **ppe-local.properties** on the Java classpath or by setting Java environment properties.

It is not possible to redefine the property values set in the .properties file of one module within the .properties file of another module.

In addition, if modules are added to the PPS application context in several steps, it is not possible to access the configuration properties of loaded modules during the addition of further modules. For example, if the PPS module **idocinbound** is added to a PPS application context that is already being used (as is the case for a local PPS within SAP Hybris Commerce), only the configuration properties of the module **idocinbound** may be used.

The PPS offers the following modules (dependencies are represented by arrows):



The modules **dataaccess-localdb** and **idocinbound** are part of SAP Hybris Commerce, integration package for SAP for Retail.

The modules **restapi** and **dataaccess-ddf** are part of the central promotion pricing service, which is part of SAP Customer Activity Repository.

Defining and Overriding Beans

By default, a Spring bean offered in the standard shipment is defined and used as follows:

Specifying ID and alias of a Spring bean

```
<alias name="sapDefaultClientApiDtoFactory" alias="sapClientApiDtoFactory" />
<bean id="sapDefaultClientApiDtoFactory" class="com.sap.ppengine.client.dto.ObjectFactory" />

<alias name="sapDefaultClientApiHelper" alias="sapClientApiHelper" />
<bean name="sapDefaultClientApiHelper" class="com.sap.ppengine.client.util.RequestHelperImpl">
  <property name="objectFactory" ref="sapClientApiDtoFactory" />
</bean>
```

Each bean has a unique ID (here specified in the name attribute). If the bean is to be enhanced using subclassing within your project, the ID of the original bean must be specified in the parent attribute of your bean. In addition, if the reference to the defined bean is to be injected into another bean, it is not usually necessary to specify the name/ID; instead, the bean with the corresponding "purpose" should be taken (be it delivered by SAP or created at the customer side). The "purpose" of a bean is represented by its Spring bean "alias". The majority of SAP beans have an additional alias. References to other beans usually make use of the bean alias. In the above example, the bean `sapDefaultClientApiHelper` uses the bean with the alias `sapClientApiDtoFactory`. In the standard shipment, this alias is provided by the bean `sapDefaultClientApiDtoFactory`. If this bean is to be replaced by a customer-specific bean, this could appear as follows:

Subclassing a bean

```
<!-- Hide old alias -->
<alias name="myClientApiHelper" alias="sapClientApiHelper" />
<!-- Define new bean subclassing existing one -->
<bean name="myClientApiHelper" parent="sapDefaultClientApiHelper" class="com.mycompany.MyHelper" />
```



The ID and alias of a bean provided by SAP always starts with "sap". The only exceptions are beans with a "magic name" expected by Spring, such as "cacheManager".

Note that technically it is also possible to completely hide a bean by choosing the same name (and not only the same alias). However, this is not usually recommended as this approach can lead to inconsistent class hierarchies if the parent attribute is used elsewhere in the bean definition.



Subclassing SAP beans offers a very flexible way to extend the application logic. However, it cannot be guaranteed that the SAP classes will be changed only in a compatible way over time. In other words, a method signature may change over time, making the subclass syntactically incorrect. The probability that an SAP object will be changed in an incompatible way increases from first to last entry in the following list:

- Spring bean ID/alias
- Java interface (methods may, however, be added)
- Signature of public method of a Java class
- Signature of protected method of a Java class
- Protected attribute of a Java class

When you redefine a Spring bean, SAP recommends the following:

- Let your custom class inherit from the SAP class. This makes sure that interface methods added by SAP are implemented.
- Define the Spring bean of the SAP class as the parent bean to your replacement Spring bean. This makes sure that additional bean properties added by SAP are set.
- Set the alias of your Spring bean to the alias of the parent (SAP) bean.
- If easily possible, reimplement the corresponding interface method(s).
- Otherwise (code duplication needed), consider redefining protected methods as well.

PPS Context

The PPS context (**com.sap.ppengine.core.Context**) offers a global container for arbitrary information that must be accessible at very different places of the application/call stack. The main use case for the PPS context is to store information that does not change for most customer installations during the time in which a price calculation request is processed. However, it can also be used as temporary global storage. Putting this information into the container leads to simpler methods with less parameters. Therefore, it is similar to the container offered by the **javax.servlet.ServletContext** interface, but can also be used outside of a servlet environment.

The PPS context is provided by a separate class implementing `com.sap.ppengengine.core.ContextProvider`. An implementation of the PPS context provider (`com.sap.ppengengine.core.impl.ThreadLocalContextProviderImpl`) is offered that holds separate contexts for each thread. This allows the easy plug-in of further PPS context initializers using a dedicated interface `ContextInitializer`. The PPS context is used to store and modify parameters within a request scope, assuming that context parameters are written and read within the same thread.

The following information is usually constant in the standard shipment:

- The SAP client (parameter `SAP_CLIENT`)
- The logical system for which external IDs are defined (parameter `SAP_LOGSYS`)
- The configuration of the promotion calculation engine (parameter `SAP_CALCENGINE_CONFIG`)

In addition, the following parameters are stored in the PPS context:

- The business unit type
- The requested language if provided

In addition to the parameters mentioned above, it is possible to store further data in the context.

✔ Context parameters that are provided by SAP have the prefix `SAP_`.

The class `com.sap.ppengengine.core.ContextParameters` contains all the context parameters used in the standard shipment.

For more information, see the documentation for the PPS Module Core in this guide.

i Starting with PPS 2.0, context parameters that are taken directly from the incoming request and do not need defaulting may also be offered as properties accessed using setter and getter methods. As an example, the requested language has been migrated from the parameter `SAP_LAN` to a regular attribute of the context.

PPS Validation

With the PPS validation concept, regular prices and OPP promotions that sent via IDocs can be validated for consistency.

As of PPS 4.0, a standardized way to validate data for consistency is supported. This validation contains more checks than the implementation available until PPS 3.0. For compatibility reasons, the original validation logic until PPS 3.0 remains default and is only replaced by the successor concept available as of PPS 4.0 if this is explicitly activated.

The PPS validation of objects is based on two concepts:

- The PPS plugin concept offering a plugin interface for validation
- The Java Bean Validation as defined in JSR-380 (see <https://beanvalidation.org/>)

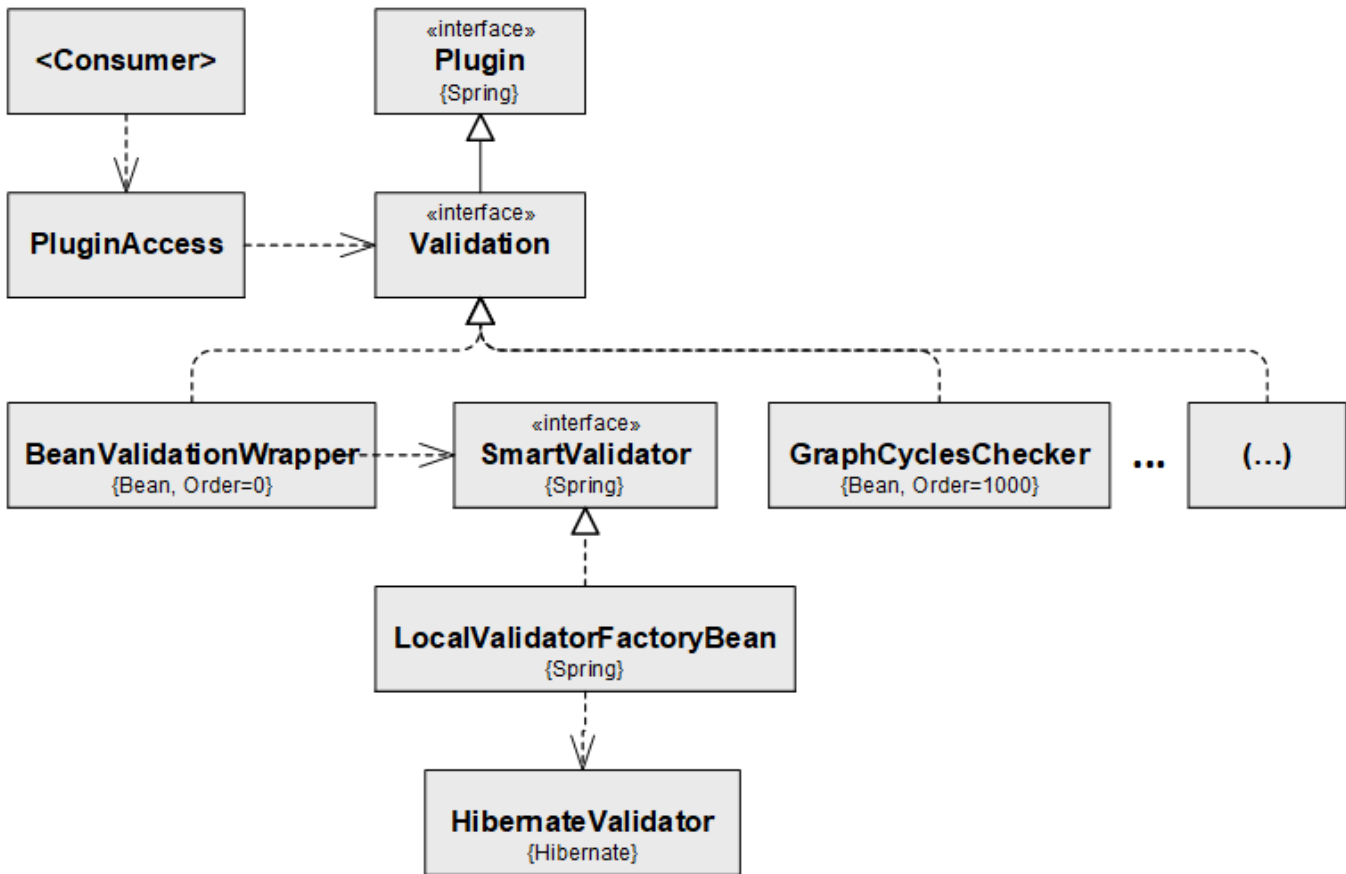
The Java Bean Validation is a well established concept and standard for validation of objects. The basic idea is to add annotations to the fields or classes to be validated defining certain constraints. When calling the validation, these annotations are evaluated and the corresponding constraints are verified. PPS uses Hibernate Validator as implementation of the API defined in JSR-380. Besides standard constraint annotations, JSR-380 allows the definition of additional annotations representing application-specific constraints. This is also done for PPS as described below in section *PPS-Specific Constraints*. For more information on JSR-380 and Hibernate validator, see <https://hibernate.org/validator/releases/6.0/>.

Bean validation focuses on constraints on single fields or single object instances via annotations which can easily be reused. However, validation may sometimes be very specific to certain classes with a low reuse so that the overhead to create annotations does not pay off. Also, validation may be required across several object instances. This isn't covered by JSR-380 in a straightforward way either. For this purpose, PPS offers a more general interface to invoke the validation of an object via the dedicated plugin interface `com.sap.ppengengine.api.plugin.Validation`. This supports the validation of single objects as well as list of objects. Besides that, it does not expose the objects `javax.validation` package but a Spring replacement to easily enable a plugin implementation of `com.sap.ppengengine.api.plugin.Validation` to signal a constraint validation. The invocation of the JSR-380 implementation is just one of several possible plugin implementations wrapped in a PPS-specific class (`BeanValidationWrapper`).

By default, a JSR-380 implementation is configured via the file `validation.xml` on the Java classpath. In an embedded context of the PPS, this may lead to issues if there is already such a file on the classpath. More than one `validation.xml` files are not supported. For this reason, the configuration and invocation of the JSR-380 implementation is not directly done but using the Spring offered wrapper `org.springframework.validation.beanvalidation.LocalValidatorFactoryBean`. This allows the configuration without `validation.xml`. Additionally, it exposes an alternative interface to `javax.validation.Validator` to call the validation providing the abstraction that is also desired when calling the PPS plugin interface.

After the validation implementation representing the bean validation wrapper (having order value 0), an additional implementation `GraphCyclesChecker` with order value 1000 is called. This implementation looks for forbidden cycles in the graphs of eligibilities or merchandise sets in an OPP promotion and performs further consistency checks.

The following diagram shows how validation is realized in PPS:



Invoking the validation for an object is then done as follows. For a list of objects it is similar.

Validate a single object

```

try {
    Class<?>[] validationGroups = (...) // could be empty, depends on the object type
    getPluginAccess().callAllWithFilterChecked(Validation.class, objectToBeValidated.getClass(),
        p -> { p.validateObject(objectToBeValidated, getContext(), validationGroups); });
} catch (final BindException be) {
    // ... do something
}
  
```

If you implement the validation interface for your own validation and want to report a constraint violation, you can proceed as follows:

Report a constraint violation

```

// Report an object level constraint violation
final BindException exc = new BindException(myObjectInstance, myObjectInstance.getClass().getName());
exc.addError(new ObjectError(myObjectInstance.getClass().getName(),
    new String[] { "myObjectErrorCode" }, null, "This object is not as it should be"));
throw exc;

// Report a field level constraint violation
final BindException exc = new BindException(myObjectInstance, myObjectInstance.getClass().getName());
exc.addError(new FieldError(myObjectInstance.getClass().getName(), "myFieldName",
    myObjectInstance.getMyFieldName(), true, new String[] { "myFieldErrorCode" }, null, "This field value is
wrong"));
throw exc;
  
```

PPS-Specific Constraints

In addition to the standard constraints defined via JSR-380 and those added by Hibernate validator, the following constraints are offered for the PPS:

- **Regular expression for timestamps:** The string representation of the annotated timestamp must match the specified pattern. This is used to reject regular prices with sub-daily validities or OPP promotions with sub-second precision effective dates or expiry dates. This constraint is offered via annotation `com.sap.ppengine.dataaccess.common.validation.TimestampPattern`.
- **Valid time range:** The effective date of the annotated class must occur before the expiry date. This constraint is offered via annotation `com.sap.ppengine.core.validation.ValidTimeRange`.
- **Conditionally mandatory properties:** If a certain property has the specified value, other properties must not be null and must not have the type-specific initial value. This constraint is offered via annotation `com.sap.ppengine.core.validation.ConditionallySet`.
- **Allowed fixed values:** The value of the annotated property must be defined as a constant in the specified class. This constraint is offered via annotation `com.sap.ppengine.core.validation.FixedValue`.
- **Valid currency with scale:** The annotated `com.sap.ppengine.dataaccess.promotion.common.entities.CurrencyWithScale` instance must have a valid content (scale > 0 and currency code filled). This constraint is offered via annotation `com.sap.ppengine.dataaccess.common.validation.ScaledCurrency`.

Adjusting Constraint Checks of the Standard Delivery

Constraint checks implemented as regular plugin implementations of `com.sap.ppengine.api.plugin.Validation` can be deactivated via the standard way described in [PPS Module Concept](#). Also, adding additional checks before or after SAP implementations is possible as described in [OPP Extensibility](#). However, constraints defined via annotations require a different approach. For this purpose, JSR-380 offers the possibility to maintain constraints in an additional XML file which can either be merged with the annotation based constraints or replace the annotation based constraints. For more information, see https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#section-mapping-xml-constraints. The following example shows how to adjust the standard delivery:

1. Create a custom constraints XML file, for example `META-INF/myown-constraints.xml`. For the JPA entity `com.sap.ppengine.dataaccess.promotion.common.entities.PromotionImpl`, we want to add a regular expression pattern for the allowed values of the origin property. In addition, for the property `logicalSystem` only values with pattern `XXXCLNTddd` shall be allowed, with X being a letter between A and Z and d a digit (0-9). This pattern shall replace the existing pattern for the logical system.

Additional validation constraints via XML file

```
<constraint-mappings xmlns="http://xmlns.jcp.org/xml/ns/validation/mapping" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/validation/mapping http://xmlns.jcp.org/xml/ns/validation/mapping/validation-mapping-2.0.xsd" version="2.0">
  <bean class="com.sap.ppengine.dataaccess.promotion.common.entities.PromotionImpl" ignore-annotations="false">
    <!-- Origin. ignore-annotations=false merges XML with annotations. -->
    <field name="origin" ignore-annotations="false">
      <constraint annotation="javax.validation.constraints.Pattern">
        <element name="regexp">^[A-Z]+$/element>
      </constraint>
    </field>
    <!-- External promotion ID. ignore-annotations=true deactivates all annotations for this field! -->
    <field name="logicalSystem" ignore-annotations="true">
      <!-- Define pattern for logical system -->
      <constraint annotation="javax.validation.constraints.Pattern">
        <element name="regexp">^[A-Z]{3}CLNT\d{3}$/element>
      </constraint>
    </field>
  </bean>
</constraint-mappings>
```

- Make this file known to the JSR-380 implementation by setting the configuration property `sap.core.beanvalidationconstraintmappinglocation`, either via `-D` or in `ppe-local.properties`:

```
# Make sure bean validation is active
sap.core.usebeanvalidation=true
# Make own constraints mapping file known to JSR-380 provider
sap.core.beanvalidationconstraintmappinglocation=classpath:META-INF/myown-constraints.xml
```



Such adjustments are required if you made a custom extension to the standard delivery, such as adjusting field lengths or the list of allowed values for certain fields (such as eligibility types).

Enabling or Disabling Bean Validation within Eclipselink

Technically, it is also possible to automatically perform a validation of the JSR-380 constraints when writing JPA entities to the database. By default, this option is not activated in the PPS, since it circumvents the additional checks of plugin `com.sap.ppengine.api.plugin.Validation`. Enabling or disabling this option is controlled via the JPA property `javax.persistence.validation.mode`. If set to `NONE` (SAP default), no bean validation takes place. Values `AUTO` or `CALLBACK` trigger the execution of the bean validation. Activating it within Eclipselink would call the bean validation twice when uploading IDocs, which is usually not desired.

Further information

When activating the Bean validation tests, keep the following in mind:

- The checks performed by the bean validation are much stricter than the very basic tests of the IDoc inbound processing done until PPS 3.0. This leads to an increased resource consumption. The upload of IDocs will be slower and more memory will be allocated.
- If you made extensions to the data model, e.g. by introducing new eligibility types, the IDocs might no longer be accepted. In this case, you can deactivate SAP standard constraints and add your own constraints via the custom constraints file as described above.
- SAP does not claim that the current checks are complete. If bean validation is active, SAP reserves the right to add further constraint checks to reveal further issues not covered so far. This will be announced but the additional checks will be active by default if bean validation is turned on.
- When running under a security manager, the bean validation requires further configuration. For more information, see https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#section-getting-started-security-manager.

PPS Module api

As of SAP Customer Activity Repository 3.0 FP2, this module provides the public API for extensions of the PPS.

Overview

As described in the chapter [Extensibility of the PPS Business Logic \(Java\)](#), the PPS provides stable extension points via Java plugin interfaces to be implemented on the customer side. The PPS module api provides these interfaces as well as the registry of all implementations found at runtime. PPS Java types referenced by the plugin interfaces (except for types from the PPS module `dataaccess-interface` and `client-interface`) are contained in this module as well - some (such as the interface for the PPS context) were moved from other modules into this module as well.

The annotations indicating the degree of stability of a Java object are also located here:

- `@ExtensionStable` - indicates that the annotated type can be extended safely on the customer side
- `@ConsumerStable` - indicates that the annotated type can be called safely on the customer side
- `@PlannedIncompatibleChange` - indicates if an incompatible change is planned for the annotated type; will be used for types annotated with `@ExtensionStable` or `@ConsumerStable`

Beans

ID	Alias	Description
<code>sapDefaultContextEnrichmentPluginRegistry</code>	<code>sapContextEnrichmentPluginRegistry</code>	Plugin registry of the ContextEnrichment interface
<code>sapDefaultRequestAdjustmentPluginRegistry</code>	<code>sapRequestAdjustmentPluginRegistry</code>	Plugin registry of the RequestAdjustment interface
<code>sapDefaultResponseAdjustmentPluginRegistry</code>	<code>sapResponseAdjustmentPluginRegistry</code>	Plugin registry of the ResponseAdjustment interface
<code>sapDefaultRequestValidationPluginRegistry</code>	<code>sapRequestValidationPluginRegistry</code>	Plugin registry of the RequestValidation interface
<code>sapDefaultQueryAdjustmentPluginRegistry</code>	<code>sapQueryAdjustmentPluginRegistry</code>	Plugin registry of the QueryAdjustment interface
<code>sapDefaultCustomEligibilityPluginRegistry</code>	<code>sapCustomEligibilityPluginRegistry</code>	Plugin registry of the CustomEligibility interface
<code>sapDefaultCustomPriceRulePluginRegistry</code>	<code>sapCustomPriceRulePluginRegistry</code>	Plugin registry of the CustomPriceRule interface
<code>sapDefaultPromotionServiceInitializationPluginRegistry</code>	<code>sapPromotionServiceInitializationPluginRegistry</code>	Plugin registry of the PromotionServiceInitialization interface
<code>sapDefaultFeatureCheckPluginRegistry</code>	<code>sapFeatureCheckPluginRegistry</code>	Plugin registry of the FeatureCheck interface

sapDefaultIdocInboundProcessingPluginRegistry	sapIdocInboundProcessingPluginRegistry	Plugin registry of the IdocInboundProcessing interface
sapDefaultPluginAccess	sapPluginAccess	Provides access to the PPS plugin interfaces via the corresponding plugin registries; PCE plugins are not accessible
sapDefaultNonUniqueBasePriceHandlingPluginRegistry	sapNonUniqueBasePriceHandlingPluginRegistry	Plugin registry of the NonUniqueBasePriceHandling interface
sapDefaultValidationPluginRegistry	sapValidationPluginRegistry	Plugin registry of the Validation interface. Available as of PPS 4.0.

Required Beans

This list contains only the additional beans to be provided if all dependencies of this module are resolved.

ID/Alias	Comment

Configuration Properties

Name	Description	Default Value	Comment
sap.client.impl.nonUniqueBasePriceHandling.strategy	This property refers to the NonUniqueBasePriceHandling interface. It enables you to switch the activated strategy based on the implemented interface method. By implementing these methods, you can implement a specific logic for processing non-unique regular prices. This Plugin is always called if a non-unique regular price is found. In this way, a specific logic can be implemented to process this situation.	SAP00	see SAP Note 2627591

Dependencies

This module depends on the following PPS modules:

- **dataaccess-interface**
- **client-interface**

PPS Module client-interface

This module provides the API of the PPS exposed to its clients. It contains the data transfer objects (DTOs) and the interface to trigger a price calculation.

Overview

This module is the outermost facade of the PPS. A client requesting a price calculation must be aware of the artifacts contained in this module. It does not contain any logic besides simple helpers to facilitate the creation of a price calculation request and evaluation of the corresponding response. If you want to call a central PPS, place at least this JAR onto the classpath of your client application.

For more information about the OPP client API, see the documentation on SAP Help Portal at <https://help.sap.com/viewer/p/CARAB> > <Version> > *Development* > *Client API for Omnichannel Promotion Pricing*.



The DTOs of the client interface are generated and use subclasses. SAP does not guarantee that the class hierarchy will remain stable over time. Therefore, we strongly recommend that you do **not** create subclasses of these DTOs on the customer side in case additional information is transported. Instead, use the predefined extension points via **any** elements realized as a List<Object>.

Extensibility via any Elements

As also described in the client API documentation, the request structure for the price calculation as well as the response structure offer extension points via **any** elements having no fixed structure. These allow arbitrary additional information to be transferred between the PPS and its client. To ensure that these **any** elements can be used in the same way for local and central deployments of the PPS, the way in which extension information is stored must be clearly defined.



The internal storage documented here is determined by the use of FasterXML Jackson. It uses the same XML (where **any** elements are effectively unwrapped lists) and JSON-based messages (where **any** elements are expected to be arrays) and should therefore also be used for local deployments where Jackson is not used.

The internal storage of an any element is a List<Object> (the only exception is the any element in LinItemChoiceDomainSpecific.java where it is only a simple object (Object)), as can be seen in the DTO for the ARTSHeader:

any-element in the ARTS Header DTO

```
public class ARTSCommonHeaderType {  
    // ...  
    @XmlElement(lax = true)  
    protected List<Object> any;  
    // ...  
}
```

What is the internal representation of the any elements and their content?

1. Each any element in an XML message or entry in the corresponding array of a JSON message corresponds to one entry in the **List<Object>**.
2. Each entry in the List, in other words the content of the any elements, is always a **Map<String,Object>**.
3. The value part of the Map entry can have the following types:
 - a. If the value corresponds to an elementary element in the XML/JSON message, this is a **String**.
 - b. If the value corresponds to a structured element in the XML/JSON message, this is a **Map<String,Object>**. The data definition of the value part is recursively defined applying rule 3.
 - c. If the value corresponds to an XML list/JSON array, this is a **List<Object>**. The element type of the list is recursively defined applying rule 3.

This is illustrated in the following example. The following is an excerpt of a request, showing only the ARTS header:

ARTS Header with any-elements - XML

```
<PriceCalculate xmlns="http://www.sap.com/IXRetail/namespace/" InternalMajorVersion="1" InternalMinorVersion="0"  
>  
<ARTSHeader ActionCode="Calculate" MessageType="Request">  
  <MessageID>9a89f2edfd1e413ea147e334b9c2ed4b</MessageID>  
  <DateTime>2250-01-13T04:48:30.427-05:00</DateTime>  
  <BusinessUnit TypeCode="RetailStore">FC01</BusinessUnit>  
  <any>Hello</any>  
  <any>  
    <foo>bar</foo>  
  </any>  
  <any>  
    <baz>17</baz>  
    <ext1>true</ext1>  
  </any>  
  <any>  
    <top>  
      <field1>value1</field1>  
      <myNode>  
        <field2>value2</field2>  
      </myNode>  
    </top>  
  </any>  
  <any>  
    <ele>one</ele>  
    <ele>two</ele>  
    <ele>  
      <a>b</a>  
    </ele>  
  </any>  
</ARTSHeader>
```

This is equivalent to the following JSON format:

ARTS Header with any-elements - JSON

```
{
  "ARTSHeader":
  {
    "MessageID":
    {
      "value": "9a89f2edfd1e413ea147e334b9c2ed4b"
    },
    "DateTime":
    [ {
      "value": "2250-01-13T04:48:30.427-05:00"
    } ],
    "BusinessUnit":
    [ {
      "value": "FC01",
      "TypeCode": "RetailStore"
    } ],
    "ActionCode": "Calculate",
    "MessageType": "Request",
    "any": [
      "Hello",
      { "foo": "bar" },
      { "baz": "17", "ext1": "true" },
      { "top": {
        "field1": "value1",
        "mynode": { "field2": "value2" } } },
      { "ele": [ "one", "two", { "a": "b" } ] }
    ]
  },
}
```

This leads to the following internal representation:

- The **any** attribute of the DTO **ARTSCommonHeaderType** is a list of length 5.
- List entry 0 is "Hello"
- List entry 1 is a Map with size 1.
 - This contains the entry
 - "foo" = "bar"
- List entry 2 is a Map with size 2.
 - This contains the entries
 - "baz" = "17"
 - "ext1" = "true"
- List entry 3 is a Map with size 1.
 - This contains the entry
 - "top" = <A Map with size 2>.
 - This contains the entries
 - "field1" = "value1"
 - "myNode" = <A Map with size 1>
 - This contains the entry
 - "field2" = "value2"
- List entry 4 is a Map with size 1.
 - This contains the entry
 - "ele" = <A List with size 3>
 - This contains the entries
 - "one"
 - "two"
 - <A Map with size 1>
 - This contains the entry
 - "a" = "b"

If you extend the PPS, you can base your coding on these rules when you process incoming requests. If you want to enhance the response to the client, you have to fill the **any** attributes of the DTOs accordingly. Vice versa, if you extend a client of the PPS, you have to fill the DTOs of the request sent to the PPS accordingly but you can rely on these rules when you process the response.

XSD and Currencies

The client interface module also contains the XSD (XML Schema Description) which is the base for the generated DTOs (Data Transfer Objects).



Until PPS 4.0, the currencies of the request and response must match a fixed list of values defined in the XSD (enumeration `CurrencyTypeCodeEnumeration`). This list does not correspond to ISO currency codes. If there are XSD validations implemented as customer extensions, a mismatch can cause issues and the calculation requests and responses might be rejected.

As of PPS 4.0, the XSD currencies of the price calculation requests and responses can have any value, either the values defined in the list or a string containing e.g. ISO currency codes.

Beans

ID	Alias	Description
sapDefaultClientApiDtoFactory	sapClientApiDtoFactory	Factory for creating the DTOs of the client API
sapDefaultClientApiHelper	sapClientApiHelper	Helper class to create a request skeleton, and so on

Configuration Properties

None

Dependencies

None

PPS Module core

This module provides basic functions that are used in the PPS.

Overview

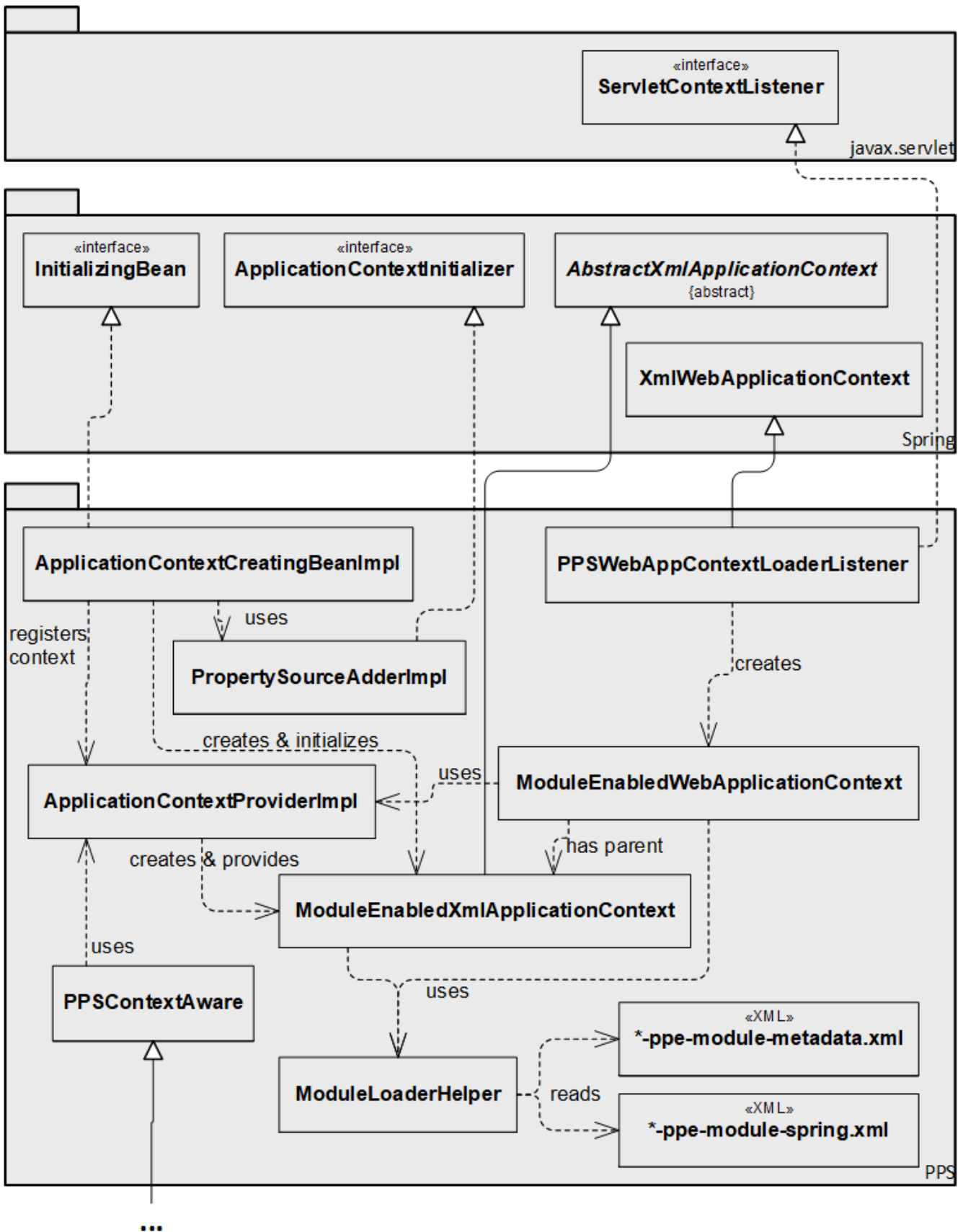
The core module offers the following functions:

- PPS application context supporting PPS modules
- PPS context
- Debug/profiling support

As described in [Promotion Pricing Service Overview](#), the PPS offers a lightweight module concept based on Spring application contexts that support modification-free extensibility. The classes enabling modularization via a PPS-specific application context are located here.

PPS Application Context

The following figure shows the most important classes that contribute to the PPS application contexts and how they interact:



ModuleEnabledXmlApplicationContext is the central class. This is a special Spring **AbstractXmlApplicationContext** that supports a distributed definition of Spring beans in separate files without a central file that explicitly includes the other resource files. Each file corresponds to a PPS module that has its metadata (name, dependencies to other modules) defined in a separate metadata file. By evaluating the defined dependencies, you can also control the order in which these beans are added to the Spring application context. The order in which Spring beans are added to an application context defines the beans that replace formerly added beans, allowing modification-free extensibility. The Spring application context that is represented by this class is also called the main PPS application context.

ModuleLoaderHelper locates the Spring bean definitions and module metadata, evaluates module dependencies, sorts the Spring bean definitions and modules according to their dependencies, and adds the Spring beans according to this sort sequence to the Spring application context. This class scans the classpath for the following file pairs located in the same directory :

- Spring bean definitions as an XML file following the resource pattern **classpath*:META-INF/**/*-ppe-module-spring.xml**
- Module metadata definitions as an XML file following the resource pattern **classpath*:META-INF/**/*-ppe-module-metadata.xml**

The following two options are available for the creation of the main PPS application context:

- Using the class **ApplicationContextProviderImpl**. This class does the following:
 - It offers a **getContext()** method that allows each caller to access the main PPS application context. To do so, it calls internally the constructor of the **ModuleEnabledXmlApplicationContext**. This option is sufficient if no external initialization of the main PPS application is required because, for example, all required Spring configuration properties are set.
 - It internally holds a reference to this application context once the main PPS application context has been created. In this way, subsequent calls of the **getContext()** method are very fast and the same application context is returned. There may be only one instance of a main PPS application context per classloader.
- Using the class **ApplicationContextCreatingBeanImpl**, it is easier to control the creation of the **ModuleEnabledXmlApplicationContext**. This class implements the Spring interface **InitializingBean**. If this class is defined as a Spring bean, the **afterPropertiesSet()** method is called automatically by the Spring framework during the implementation of this interface. The following happens within this method:
 - The main PPS application context is created.
 - All injected Spring **ApplicationContextInitializer** implementations are executed before the application context is refreshed. These initializers allow further initialization of the main PPS application context. In particular, it is easy to set Spring configuration properties during runtime via the class **PropertySourceAdderImpl**. This is helpful if, for example, you are running the PPS as a local deployment within a hosting application.
 - Finally, the main PPS application context is refreshed and registered in the class **ApplicationContextProviderImpl**, which makes it available in the application.

The main PPS application context is well suited if an application wants to call the PPS internally. Therefore, the logic to execute the price and promotion calculation should be located within this application context. However, it is not possible to have only this application context for the following reasons:

- Exposing servlets, such as the IDoc inbound, requires a web application. Spring requires a web application context as the root application context of a web application.
- The PPS relies on several open-source libraries. If these libraries are on the same classpath as a hosting application, this may lead to side effects. For example, if you use Jackson XML processes and the corresponding library is on the classpath, Spring automatically gives Jackson preference over the Jaxb2-based XML processes for the commonly used **RestTemplate**. Since Jaxb2 and Jackson are not 100% compatible with each other, this may lead to issues.

Therefore, a **ModuleEnabledWebApplicationContext** that extends **XmlWebApplicationContext** is offered in addition to the **ModuleEnabledXmlApplicationContext**. This class does the following:

- After creation, this context automatically tries to make itself the child of an existing **ModuleEnabledXmlApplicationContext** using the class **ApplicationContextProviderImpl**. Therefore, all PPS modules located in the main PPS application context are available in the Web application context.
- It scans the classpath of the corresponding Web application for PPS modules using the **ModuleLoaderHelper**. Modules that are not yet available in the main PPS application context will be added to the Web application context. Since the classpath of a Web application may be larger than the classpath of the main PPS application context, the issues mentioned above (first bullet point) are avoided.



A module located in the Web application context cannot be dependent only on a module located in the main PPS application context.

The Web application context needed for a Spring-based Web application must be loaded by a **ServletContextListener** that is registered in the web.xml of the corresponding Web application. The implementation that creates the **ModuleEnabledWebApplicationContext** is the class **PPSWebAppContextLoaderListener**. Once the servlet context is initialized, it automatically creates the PPS Web application context and stores in it the servlet context attribute **SAP_PPS_WEBAPPCONTEXT**.

Make the name of the servlet context attribute known to the following Spring **DispatcherServlet**:

```

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">
  <display-name>Price and Promotion Engine WebApp (central)</display-name>
  <!-- One dispatcher servlet for price calculation requests as well as iDoc
    inbound processing -->
  <servlet>
    <servlet-name>Dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <!-- Name of the servlet context attribute holding the PPS web app context -->
      <param-name>contextAttribute</param-name>
      <param-value>SAP_PPS_WEBAPPCONTEXT</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Dispatcher</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
  <!-- Create & initialize PPS web app context on startup -->
  <listener>
    <listener-class>com.sap.ppengine.core.spring.impl.PPSWebApplicationContextLoaderListener</listener-
class>
  </listener>
</web-app>

```

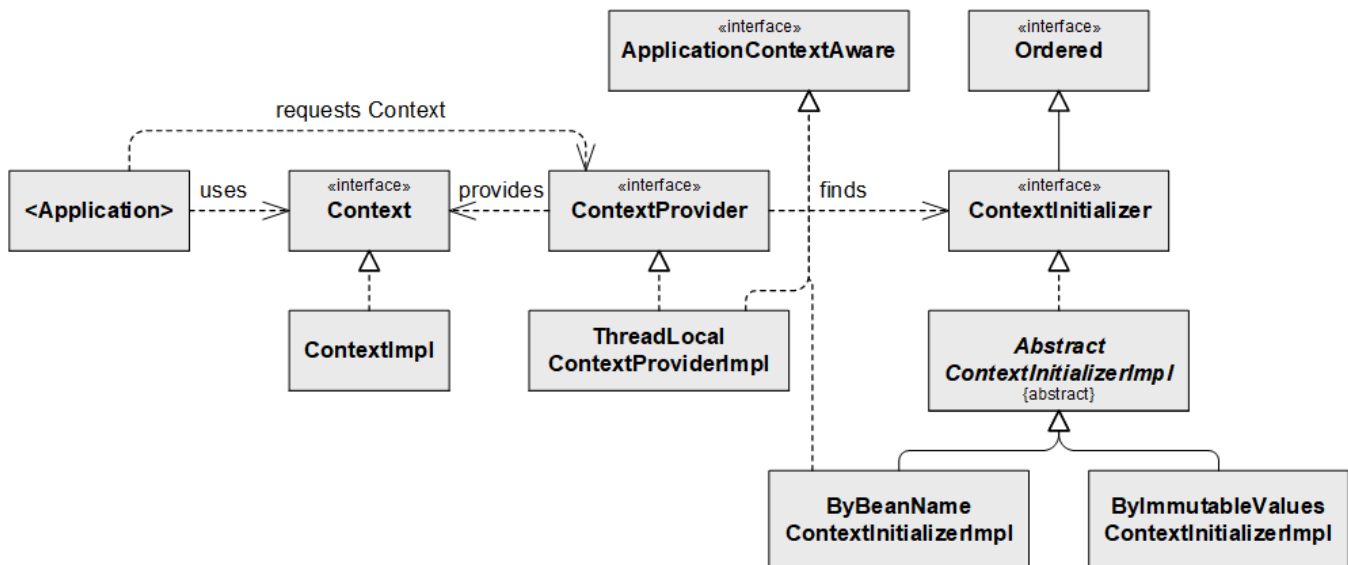
The class **PPSContextAware** is offered as a convenience class and possible root class for applications that need access to the main PPS application context. Regardless of how the subclass has been created (either explicitly or by a runtime container, such as another Spring application context), it offers internal access to the main PPS application context.

PPS Context

The PPS context (that is more related to a Spring servlet context than to a Spring application context) serves as a container that can be used to store data during request processing. This container can be seen by all parts of the application. Typically, these values should be constant. However, it is also possible to modify parameters that are stored in this context. For example, the requested language is information provided by the price calculation request. This information is not known by the promotion calculation engine. However, it is relevant for the underlying data access layer. This information can be extracted from the price calculation request using the PPS context and can be used later when reading promotional data.

The PPS context also allows (and requires) an initialization at the beginning of request processing. This resets all of its values to a defined initial state.

The following figure shows the main components of PPS context handling:



The **Context** is the container that holds the information on which the application works. It is implemented by the class **ContextImpl**, which does not contain additional logic besides the pure storage of data. To access the **Context**, the application requests it from a **ContextProvider** that is offered as a Spring bean. This Spring bean can be injected into the corresponding application bean. As of now, there is only one implementation of the **ContextProvider**, the **ThreadLocalContextProviderImpl**. This stores the **Context** in a **ThreadLocal** variable. Therefore, this implementation relies on the assumption that the processing of one request is realized by one Thread (which may be reused later on).

The initialization of the **Context** by the **ContextProvider** is delegated to a list of **ContextInitializer** instances. These are not injected into the **ContextProvider**. Instead, the **ContextProvider** searches in the current Spring application context for all Spring beans that implement the **ContextInitializer** interface. This interface extends the Spring **Ordered** interface and therefore allows you to control the order in which the **ContextInitializers** are processed. The following two implementations of **ContextInitializer** are offered that share common attributes and logic in the abstract **AbstractContextInitializerImpl** class:

- The class **ByImmutableValuesContextInitializerImpl** writes all the entries of the **Map** (that can be injected to this class) into the PPS **Context**.



This class should be used only if the values of the map entries are immutable, for example **String**, **Integer**, and so on.

The following excerpt shows how a Spring bean that uses this class could look:

```

<!-- Data access relevant initialization parameters of PPS context -->
<alias name="sapDefaultDbContextInitializer" alias="sapDbContextInitializer" />
<bean id="sapDefaultDbContextInitializer" class="com.sap.ppengine.core.impl.
ByImmutableValuesContextInitializerImpl">
    <property name="initValues">
        <map>
            <entry key="SAP_CLIENT" value="\${sap.dataaccess-common.db.client}" />
            <entry key="SAP_LOGSYS" value="\${sap.dataaccess-common.logSys}" />
            <entry key="SAP_BUTYPE" value="\${sap.dataaccess-common.defaultBuType}" />
        </map>
    </property>
</bean>

```

- The class **ByBeanNameContextInitializerImpl** writes the reference to a bean into the PPS context. This bean is specified by its bean name. This class should be used if the class of the references instance is not immutable, for example, a **Map** that might have been changed in previous request processing. As shown in the following example, changes can be undone using prototype scoped beans:

```

<!-- Calc engine relevant initialization parameters of PPS context -->
<alias name="sapDefaultCalcEngineContextInitializer" alias="sapCalcEngineContextInitializer" />
<bean id="sapDefaultCalcEngineContextInitializer"
class="com.sap.ppengine.core.impl.ByBeanNameContextInitializerImpl">
    <property name="paramName" value="SAP_CALCENGINE_CONFIG"></property>
    <property name="beanName" value="sapCalcEngineConfigCopy" />
</bean>
<!-- Prototype scoped bean! -->
<alias name="sapDefaultCalcEngineConfigCopy" alias="sapCalcEngineConfigCopy" />
<bean id="sapDefaultCalcEngineConfigCopy" factory-method="toProperties"
scope="prototype" class="org.apache.commons.collections.MapUtils">
    <constructor-arg>
        <ref bean="sapCalcEngineConfig" />
    </constructor-arg>
</bean>

```

Bean Validation

As of with PPS 4.0, a validation of objects based on Bean Validation (JSR-380) is supported. The required beans are defined in this PPS module in order to have the validation available in all PPS modules containing business logic.

For more information about this validation concept, see chapter *PPS Validation*.



Beans

ID	Alias	Description
sapDefaultTimeResolutionReducer	sapTimeResolutionReducer	Reduces the resolution of a time stamps. With this implementation, the resolution is reduced to a day-level. This is only used when regular prices are read. Adjust this bean if another resolution of regular price or OPP promotion validities is required.

sapDefaultStringifier	sapStringifier	Helps to create a string representation of a Java class if it does not offer a suitable toString() method. Used for creating debug messages, and so on.
sapDefaultTimerFactory	sapTimerFactory	Factory to create a timer to measure the duration of a price calculation. If the configuration parameter sap.core.requesttimer is set to true, a timer is created that stores measurements in a ThreadLocal container. Otherwise, a dummy implementation that records no measurements is created.
sapDefaultThreadLocalTimer	sapThreadLocalTimer	Timer created by the sapTimerFactory .
sapDefaultContextProvider	sapContextProvider	Bean that offers a PPS Context . The bean is implemented by default by a ThreadLocalContextProviderImpl .
sapDefaultEligibleCacheKeyGenerator	sapEligibleCacheKeyGenerator	Key generator used by Spring Cache abstraction. Intended for the eligibility references except for those referring to MerchandiseSet Eligibilities . Moved from the dataaccess-common PPS module into this module as of PPS 3.0.
sapDefaultMerchSetEligibleCacheKeyGenerator	sapMerchSetEligibleCacheKeyGenerator	Key generator used by Spring Cache abstraction. Intended for the eligibility references to MerchandiseSet Eligibilities . Available as of PPS 3.0.
sapDefaultPriceCacheKeyGenerator	sapPriceCacheKeyGenerator	Key generator used by Spring Cache abstraction. Intended for the regular prices. Moved from the dataaccess-common PPS module into this module as of PPS 3.0.
sapDefaultSystemProperties	sapSystemProperties	Java system properties exposed as a Spring bean. Available as of PPS 3.0.
sapDefaultPPSProperties	sapPPSProperties	Content of the ppe-local.properties merged with Java system properties as a Spring bean. System properties have precedence over the content of the ppe-local.properties file entries. Available as of PPS 3.0.
sapDefaultBeanValidator	sapBeanValidator	Available as of PPS 4.0. Spring framework wrapper around the JSR-380 implementation. This allows validation.xml-less configuration via Spring configuration properties. For more information, see sap.core.beanvalidationproviderclass , sap.core.beanvalidationconstraintmappinglocation and bean sapBeanValidationProperties . This bean should only be referenced if only bean validation must be done. If this is not the case, it is recommended to call the plugin <code>com.sap.ppengine.api.plugin.Validation</code> .
sapDefaultBeanValidationProperties	sapBeanValidationProperties	Available as of PPS 4.0. Bean holding properties to control the bean validation implementation. The properties are expected as .properties file specified via Spring configuration property sap.core.beanvalidationpropertieslocation .
sapDefaultBeanValidatorWrapper	sapBeanValidatorWrapper	Available as of PPS 4.0. Thin wrapper calling bean sapBeanValidator , implementing com.sap.ppengine.api.plugin.Validation . This evaluates the Spring configuration property sap.core.usebeanvalidation .

Configuration Properties

Name	Description	Default Value	Comment
sap.core.ppsconfiguration	Location of the PPS configuration file in Spring resource syntax	classpath:/ppe-local.properties	Since this property specifies the name of the configuration file, it cannot be specified in the configuration file itself. It must be set externally, for example, via a Java environment variable.
sap.core.requesttimer	Switch to activate the request timer	false	The request timer can be used to measure how long the processing of a price calculation request takes, broken down to certain parts of the process. Note that only server-side processing time without marshaling/unmarshaling is considered.
sap.core.jperftimer	Switch to activate jperf timer instead of default request timer	false	Available as of PPS 4.0. If activated in combination with sap.core.requesttimer , the default request timer is exchanged with the jperf timer which produces log file entries in the format used by JPerf. For more information, see the JPerf documentation under https://github.com/sovaa/jperf .
sap.core.usebeanvalidation	Evaluate the standard and SAP-specific JSR-380 annotations for the JPA entities during validation. This configuration property only controls the execution of the corresponding plugin implementation for plugin interface com.sap.ppengine.api.plugin.Validation . Other implementations of this interface are not affected by this property.	false	Available as of PPS version 4.0.

sap.core. beanvalidationproviderclass	Class representing the JSR-380 implementation to be used for bean validation.	org. hibernate. validator. Hibernate Validator	Available as of PPS 4.0.  It is not recommended to change this configuration property.
sap.core. beanvalidationpropertieslocation	Location of the properties controlling the chosen JSR-380 implementation.	classpath: META-INF /empty. properties	Available as of PPS 4.0.
sap.core. beanvalidationconstraintmappinglocation	Location of the XML file holding additional validation constraints on customer side. This can be used to add further constraints or deactivate constraints defined in the standard delivery.	classpath: META-INF /empty- constraints .xml	Available as of PPS 4.0.  There is no classpath: prefix.

Dependencies

This module depends on the following modules:

- **api (starting with PPS 3.0)**

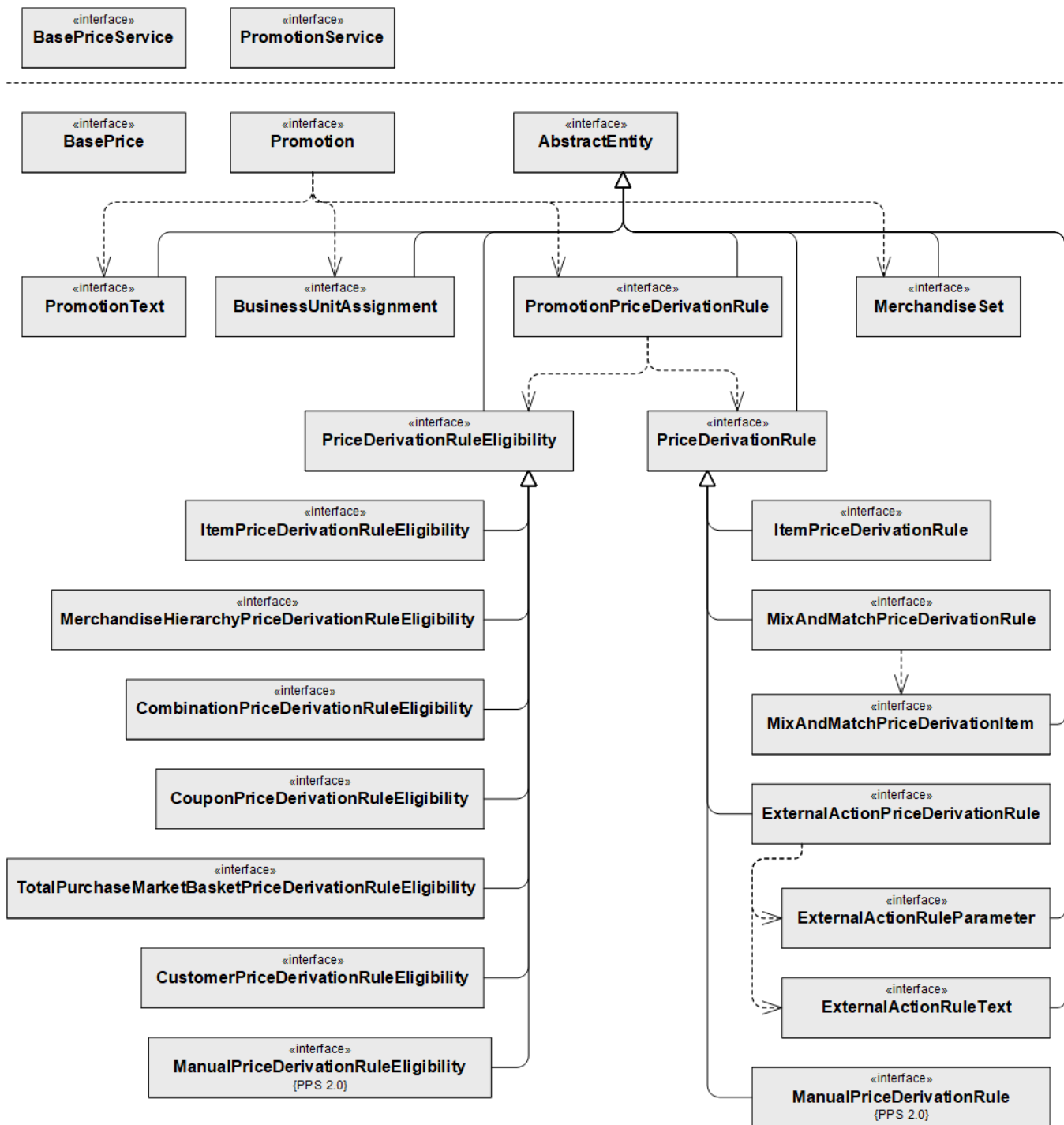
PPS Module dataaccess-interface

This module provides the abstraction layer for the read-only persistence services and the data retrieved by them.

Overview

The main purpose of this module is to shield the implementation details of the data access to other modules. It only offers interfaces and classes containing constants. Together with the module client-interface it offers the touch points between the promotion calculation engine and the rest of the PPS.

The following figure shows the most important objects in this module. The interface **AbstractEntity** offers a generic field extension to the promotion-related entities. For more information about the entities and services, see the Javadocs.



Although they are logically part of the key, the SAP client and the logical system are not part of the exposed entities. They are provided via the PPS context.

Beans

None

Configuration Properties

None

Dependencies

None

PPS Module jackson

This module provides a uniform configuration for a server and possible clients for the JSON- and XML-based message exchange.

Overview

The PPS uses Jackson for the conversion between request/response payload and its internal representation as Java classes. It is recommended that the PPS client does the same. In addition, to enable a smooth integration between the PPS and its possible clients, it is necessary to have the same data format, even using Jackson on both sides. This means that the conversion of the corresponding converters is the same on the server and client side. This includes the following aspects:

- Consideration of JAXB annotations in the Java classes for the DTOs
- Date format (yyyy-MM-dd'T'HH:mm:ss.SSS)
- (No) pretty printing
- Handling of empty and null value fields (to be ignored)
- Setting the time zone (to the corresponding default time zone). This is important because otherwise Jackson assumes that UTC is the time zone of the request, which usually differs from the JVM time zone. Since the date and time of the price calculation request calls is assumed to be in the local time zone of the PPS client and the price and promotion information is also stored in this schema, for example, without time zones, it is not necessary to convert the date and time. This is achieved by the following:
 - Setting the converter time zone to the JVM time zone (no automatic Jackson internal conversion)
 - Not expecting time zone information, at least for the price validity dates in the payload



Although the configuration-relevant classes of the JSON- and XML-based message exchange are located here, the configuration itself does not take place. This must be done explicitly in addition.

Configuring Jackson (Client Side)

Jackson is configured by declaring Spring beans. This is relevant on the server side as well as on the client side. Although technically not required, we recommend that you place this module (more precisely, the JAR containing this module) on the classpath of the PPS clients to allow for easy reuse. If this is the case, the configuration may look as follows (assuming the PPS client uses Spring as well):

JSON configuration on PPS client side

```
<alias name="myDefaultJacksonJsonConverter" alias="myJacksonJsonConverter" />
<bean id="myDefaultJacksonJsonConverter" factory-bean="myJacksonJsonConverterBuilder"
      factory-method="build" />
<alias name="myDefaultJacksonJsonConverterBuilder" alias="myJacksonJsonConverterBuilder" />
<bean id="myDefaultJacksonJsonConverterBuilder" class="com.sap.ppengine.jackson.
JacksonJsonConverterBuilder" />
```

The resulting bean implements `HttpMessageConverter` and can be easily used in, for example, a Spring `RestTemplate` (assuming `myJacksonJsonConverter` has been injected as a dependency):

```
protected RestTemplate getRestTemplate()
{
    final HttpMessageConverter<?> converter = getHttpMessageConverter();
    RestTemplate restTemplate = new RestTemplate(Collections.<HttpMessageConverter<?>> singletonList
(converter));
    return restTemplate;
}
```



Although this module declares a Maven dependency to `jackson-dataformat-xml`, this dependency is not required if only the data format JSON is used.

Request Logging

For debugging purposes, it is helpful to trace the incoming requests with their payload on the server side. This is enabled using the class `RequestToSlf4JLogger`. The requests are traced if its log level is set to `TRACE`. This class must be registered as a filter in `web.xml`:

Registering the request logger in web.xml

```
<!-- Filter to enable logging of ingoing requests via SLF4J - log level
of filter class must be set to TRACE to become effective -->
<filter>
  <filter-name>sapRequestLogger</filter-name>
  <filter-class>com.sap.ppengine.web.filter.RequestToSlf4JLogger</filter-class>
  <init-param>
    <param-name>maxPayloadLength</param-name>
    <param-value>10000</param-value>
  </init-param>
  <init-param>
    <param-name>includePayload</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>includeQueryString</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>beforeMessagePrefix</param-name>
    <param-value>REQUEST BEFORE PROCESSING---></param-value>
  </init-param>
  <init-param>
    <param-name>afterMessagePrefix</param-name>
    <param-value>REQUEST AFTER PROCESSING---></param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>sapRequestLogger</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

Beans

ID	Alias	Description

Configuration Properties

None

Dependencies

None

PPS Module restapi

This module exposes the price calculation as a RESTful service.

Overview

The price calculation is exposed as a RESTful service. Both XML as well as JSON format is supported. In both cases, the JAXB annotations of the client API DTOs are considered. The conversion between JSON/XML and the Java DTOs is done using FasterXML Jackson. The REST service is exposed using the relative path **/restapi**. It is realized using Spring MVC. The restapi is only a thin wrapper around the actual calculation logic. It delegates the calculation call to the corresponding PricingPromotionService instance and receives the response from it.

If exceptions are thrown during the processing of a price calculation request, these are not propagated back to the sender of the request. Instead they are recorded in the application log for security reasons.

The following HTTP response codes are possible:

- 200 (OK): This is returned if the price and promotion calculation was successful.
- 400 (Bad Request): This is returned if the request validation detected an error.

- 401 (Unauthorized): Authentication data is missing or incorrect.
- 403 (Forbidden): The authorization required to perform the calculation is missing for the authenticated user.
- 500 (Internal Server Error): This is returned if an unexpected error occurs.



Codes 401 and 403 are relevant only if the REST service is secured by authorization checks. These are not part of this PPS module.

Known Issues

FasterXML Jackson is used for mapping between the JSON/XML format used by external clients and the internal representation as Java classes. As of now, this library has the following known issues:

- Under some circumstances, attributes marked as optional are treated as mandatory. This is particularly true for the `BusinessUnitTypeCode`.
Workaround: Always specify the `BusinessUnitTypeCode` in the request.
- The elements of unwrapped lists with an XML payload must be co-located within the corresponding parent node, in other words no other element may be in between. For example, the following payload leads to the mapping of only part of the elements:

Illegal unwrapped list

```
<LineItem>
  <MerchandiseHierarchy ID="ID1" >hier1</MerchandiseHierarchy>
  <SequenceNumber>0</SequenceNumber>
  <MerchandiseHierarchy ID="ID2" >hier2</MerchandiseHierarchy>
</LineItem>
```

Workaround: Create the request accordingly.

Beans

ID	Alias	Description
sapDefaultPriceCalculateController	sapPriceCalculateController	Spring MVC controller that accepts the price calculation requests via HTTP POST
sapDefaultJacksonJsonConverterBuilder	sapJacksonJsonConverterBuilder	Factory bean: Builder for the org.springframework.http.converter.HttpMessageConverter that takes care of the conversion between the message/request body in JSON format and the internal representation as Java classes
sapDefaultJacksonJsonConverter	sapJacksonJsonConverter	HttpMessageConverter built by sapJacksonJsonConverterBuilder
sapDefaultJacksonXmlConverterBuilder	sapJacksonXmlConverterBuilder	HttpMessageConverter built using the bean sapJacksonXmlConverter
sapDefaultJacksonXmlConverter	sapJacksonXmlConverter	Factory for HttpMessageConverter that uses Jackson to convert from and to XML messages using a well-defined configuration; JAXB annotations are considered

Configuration Properties

None

Dependencies

This module depends on the following modules:

- client-impl
- jackson

PPS Module client-impl

This module provides the implementation of the client API for calculating sales prices and promotions.

Overview

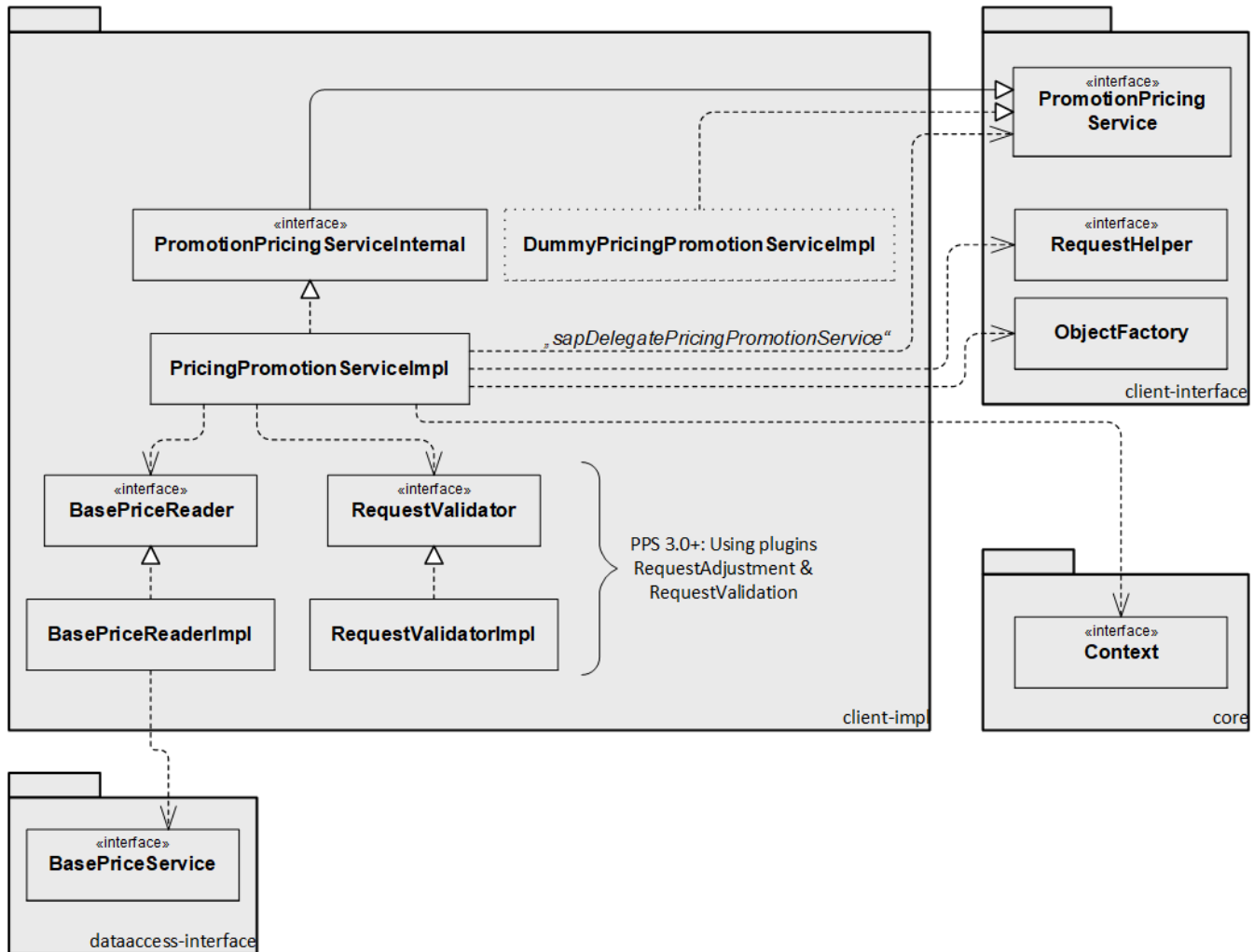
The PPS calculates a shopping cart as follows:

1. It determines the regular prices.
This step is optional since the regular prices can also be provided by the client.
2. It applies the relevant promotions based on the regular prices.

The first step is performed in this module. The regular prices are determined for all the items in the shopping cart for which prices have not been provided by the consumer of the service. Before this is done, the price calculation request is validated. This price validation checks if all the fields needed for the look-up of regular prices are filled.

The promotion calculation itself is delegated to a delegate that is referenced as another Spring bean.

The following figure shows the most important components of the **client-impl** module. Note that the class **DummyPricingPromotionServiceImpl** is not used productively, it is just used as a stub if the promotion calculation engine is not available in a test environment.



Request Validation

This section describes the request validation in the **client-impl** layer. There are also other validations inside the promotion calculation engine. For a complete list of the possible error codes, see the documentation for the OPP client API for your local or central promotion pricing service. The request validation on the **client-impl** layer fails in at least one of the following cases:

- **InternalMajorVersion** is missing in the request
- **Invalid InternalMajorVersion** and/or **InternalMinorVersion**
- There is no **ARTSHeader** in the calculation request
- Unsupported **actionCode** in **ARTSHeader**
- Unsupported **messageType** in **ARTSHeader**
- The **RequestedLanguage** and **RequestedMultiLanguage** fields are together in the calculation request (for PPS requests as of client API version 2.0)
- Wrong number of element **PriceCalculateBody** in the calculation request (only one supported)
- **DateTime** is missing in **PriceCalculateBody**
- Invalid **BusinessUnit**
- **BusinessUnit** is longer than 60 characters (for PPS requests as of client API version 2.0)
- Invalid number of **BusinessUnits** in the calculation request (only one supported)
- There is no **ShoppingBasket** in the calculation request
- Invalid number of quantity elements in the calculation request (only one per line item supported)

- There is no **LineItem** in the calculation request
- **The number of line items exceeds the defined threshold**
- Invalid **ItemID**
- **ItemID** is longer than 60 characters (for PPS requests as of client API version 2.0)
- Invalid number of **ItemIDs** in the calculation request (only one per line item supported)
- Invalid **UnitOfMeasure**
- **RegularSalesUnitPrice** is missing although **FixedPriceFlag** is set to **true**
- Invalid number of regular prices retrieved for a **LineItem**
- **More than two different merchandise group hierarchy identifier qualifiers for a request (depends if merchandise sets are enabled)**

Once an error is detected, the validation stops and errors are not collected. In the case of a validation error, the response code is set to **REJECTED**. In addition, the **ARTSHeader** of the response is filled with a **BusinessError** element that describes the error using an SAP error code. Documentation for the SAP error codes can be found in the Javadoc for the class **com.sap.ppengine.client.impl.PriceCalculateConstants**.

If all validations are successful and the regular prices (if needed) have been read, the request is forwarded to the promotion calculation engine for further processing (applies the relevant promotions).

Single vs Bulk Access for Regular Prices

A main task of this module is to determine the regular prices of items if they have not yet been provided by the consumer of the service. This is done using the **BasePriceService** of the **dataaccess-interface** module. To achieve the best performance and ensure consistent results, prices that have been determined for the corresponding shopping cart in former price calculation requests should be remembered on the client side and sent as part of the next request (with **fixedPriceFlag** set). As a result, a regular price should not have been determined yet for only a very limited number of items (ideally only one). The price for the remaining item can be determined by a single price look-up that is cached in the data access layer. However, this is not automatically ensured but determined by the consumer of the PPS. Therefore, it is also possible that the regular price has to be determined for several items (in some cases all items). In this case, a single look-up for each article is not feasible if the corresponding prices are not within the cache.

Therefore, the following strategy is applied:

- If the number of regular prices to be determined is below a fixed threshold, a single access is done for each price, considering application built-in caches.
- If the number of regular prices to be determined is greater than or equal to the set threshold, one bulk access is performed. With PPS 1.0, this access bypassed the cache for regular prices. As of PPS 2.0, this access also considers and updates prices that are not provided by the client but are already in the cache.



Due to the restrictions of the existing interface, the set of prices may be larger than needed.

The threshold for the number of items without provided prices can be specified using configuration property **sap.client-impl.basepricebulkaccessitemthreshold**.

Handling of Business Unit Type

The business unit type is externally provided information within the **ARTSHeader**. Its handling differs from the business unit ID. This is due to a difference in the data model of the promotion calculation engine and the corresponding data model of a DDF location:

- In the case of the DDF location, the location has an external compound key consisting of the location ID and the location type code.
- In the case of the ARTS data model, the business unit type is a simple attribute of the business unit. Therefore, the business unit type is not considered within the promotion calculation engine. The engine does not supply the information about the business unit type when it requests data from the data access layer.

To provide the business unit type to the data access layer, which needs it to access the database tables, this information is stored for the corresponding price calculation request within the PPS context as the parameter **SAP_BUTYPE**. This is done within the **client-impl** module.

Beans

ID	Alias	Description
sapDefaultCalculateRequestValidator	sapCalculateRequestValidator	Validator for a price calculation request For more information, see <i>Request Validation</i> .
sapDefaultBasePriceReader	sapBasePriceReader	Reader for regular prices
sapDefaultPricingPromotionService	sapPricingPromotionService	The main entry point of the PPS on Java level. This delegates the work internally to the validation, the reading of regular prices, and the calculation of promotions.
sapDummyPricingPromotionService	sapDelegatePricingPromotionService	Dummy implementation for the promotion calculation. The bean with this alias is to be replaced by the "real" implementation, as described in <i>PPS Module calcengine-gk</i> .
sapCalculateRequestValidation	sapDefaultCalculateRequestValidation	Plugin implementation performing the actual request validation. Available as of PPS 3.0.

sapAddBasePricesToRequest	sapDefaultAddBasePricesToRequest	Plugin implementation adding the regular price to the calculation request. Available as of PPS 3.0.
---------------------------	----------------------------------	---

Required Beans

This list contains only those additional beans to be provided if all the dependencies of this module are resolved.

ID/Alias	Comment
sapBasePriceService	Reads the regular prices
sapPromotionService	Reads the promotions
sapTransactionManager	Manages the (read) transactions
(sapDelegatePricingPromotionService)	Does the real promotion calculation; by default a stub is used doing nothing

Configuration Properties

Name	Description	Default Value	Comment
sap.client-impl.basepricebulkaccesstentthreshold	Minimum number of line items without prices leading to a bulk access instead of single read accesses	10	A bulk access to prices is done if the number of items without prices provided by the client is greater than or equal to this threshold. Setting this property to 0 will always lead to a bulk access.
sap.client-impl.maxnumberoflineitems	Maximum number of line items that may be within a price calculation request	200	Set to 0 if you do not want to set a threshold.
sap.client-impl.maxcalculationretries	Maximum number of price calculation retries	10	A price calculation retry takes place when invalid cache entries are detected. In this case, the invalid entries are evicted from the cache and the whole calculation is restarted.

Dependencies

This module depends on the following modules:

- core
- client-interface
- dataaccess-interface

PPS Module calcengine-gk

This module provides the promotion calculation engine.

Overview

The PPS application context is not known by the promotion calculation engine. Its internal functions are described in the technical documentation for the promotion calculation engine (SDK Promotion Calculation Engine). This module serves only as a wrapper to include the promotion calculation engine in the PPS application context. In addition, it contains the default settings of the configuration parameters for the promotion calculation engine.

Beans

ID	Alias	Description
sapDefaultCalcEngineConfig	sapCalcEngineConfig	Default configuration for the promotion calculation engine as Java Properties. No write access from the application.
sapDefaultCalcEngineConfigMap	sapContextParametersEngine	Maps the wrapping of the default configuration so that the whole properties are stored in one map entry for value SAP_CALCENGINE_CONFIG . This entry is automatically added to the default PPS context. This bean has prototype scope.
sapDefaultCalcEngineContextInitializer	sapCalcEngineContextInitializer	Initializer for the PPS context that fills the promotion calculation engine configuration parameters.
pricingPromotionService	sapDelegatePricingPromotionService	This is the main bean for performing the promotion calculation. Here it is wired to the delegate that is defaulted to a dummy implementation by the client-impl module.

sapDefaultFeatureCheck	sapFeatureCheck	This is the bean that checks if special features of the PPS are active (for example, offers on product groups).
(sapPromotionService)	promotionServiceSAP	The existing alias sapPromotionService is also offered as promotionServiceSAP , which represents the dataaccess service required by the promotion calculation engine.
<many more>	<many more>	The promotion calculation engine consists of many more Spring beans that are available on SAP Help Portal at https://help.sap.com/viewer/p/CARAB <Version> Development> <i>SDK Promotion Calculation Engine</i> .

Default Settings and Properties

The promotion calculation engine supports a lot of configuration properties that are set to default values in the PPS standard delivery. The use of product groups via merchandise sets was introduced with PPS 3.0 and is not active by default. The corresponding property **merchandiseSetsEnabled** must be set to *true* to use this feature. Each property of the promotion calculation engine can be set either as JVM environment property specified via the **-D** option (in the case of central deployment using XSA) or as local property located in the **ppe-local.properties** file. The complete list of PCE config properties and is described in the SDK Promotion Calculation Engine on SAP Help Portal at <https://help.sap.com/viewer/p/CARAB> <Version> Development> *SDK Promotion Calculation Engine*.

Required Beans

This list contains only those additional beans to be provided if all the dependencies of this module are resolved.

ID/Alias	Comment
sapPromotionService	Reads the promotions
sapTransactionManager	Manages the (read) transactions

Configuration Properties

Name	Description	Default value	Comment
sap.calcengine-gk.configproplslocation	Location of the default promotion calculation engine configuration properties in Spring resource syntax	classpath:META-INF/calcengine-gk-config.properties	The list of configuration properties is part of the promotion calculation engine documentation that is available on SAP Help Portal at https://help.sap.com/viewer/p/CARAB <Version> Development> <i>SDK Promotion Calculation Engine</i> .



The default values of the configuration properties differ from the description in the *OPP Functional Guide for the Promotion Calculation Engine*. The functional guide describes the default values for properties that are not set via the file specified with **sap.calcengine-gk.configproplslocation**.

Dependencies

This module depends on the following PPS modules:

- **client-impl**
- **dataaccess-interface** (transitive dependency as of PPS 3.0)

PPS Module dataaccess-common

This module provides the implementation of the data access layer, independent of the underlying database.

Overview

The module **dataaccess-common** is the main module that is needed to provide access to the persistence (OPP promotions and regular prices). It provides the implementations of the entity interfaces and of the interfaces to access the entities offered via the module **dataaccess-interface**. The implementation is based on Java Persistence API (JPA) 2.1. [EclipseLink](#) is used as the JPA provider. The dataaccess-common module contains database-independent information and, for example, **module dataaccess-localdb** to isolate database specifics so that enhancements of the database access can be reused on different databases. These common artifacts are stored in **dataaccess-common**, whereas **dataaccess-localdb** contains artifacts specific to a local, row-oriented relational database.

Regular Price

The regular price is modeled as JPA entity **BasePriceImpl**. It is accessed using the class **BasePriceServiceImpl**. The regular price is held at the level of *Article ID/Unit of Measure Code/Price Class (Net/Gross)/Business Unit/Business Unit Type/SAP Client/Logical System* and *Valid From*.

Promotional Information

The promotion is stored in several entities located in the package `com.sap.ppengine.dataaccess.promotion.common.entities`. The root JPA entity is `PromotionImpl`. The class hierarchy follows the interface hierarchy shown in the section `dataaccess-interface`. Promotional information is accessed using `PromotionServiceImpl`.

Object-Related Mapping Using Spring

The following Spring enhancements are used for object-related mapping:

- The class `org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean` is used as the entity manager factory. This allows an easy configuration using Spring properties instead of having to maintain a `persistence.xml` file. In addition, it also supports the easy configuration of JPA properties by reusing existing Spring concepts, such as maps stored as properties files. Setting the list of packages to scan for JPA entities can be just a matter of Spring property configuration.
- The class `org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor` enables automatic injection of an entity manager using the annotation `@PersistenceContext`. In addition, Spring automatically provides a thread-safe entity manager so that concurrent requests can be handled without further precautions.
- The class `org.springframework.orm.jpa.JpaTransactionManager` implements JPA transaction handling.

Multi-Step JPA Resource Mapping

The aim of the data access is to have the JPA entities independent of the underlying database. This will become important if several deployment options are offered. This module provides the entity implementations, making some abstractions from specific database table design details, such as indexes. These specifics are added using the module `dataaccess-localdb`.

Since JPA entities are subject to extensibility, the following strategy is used for their definition:

- The properties of a JPA entity expected to be common to all deployment options are specified via annotations as an integral part of the Java class for the JPA entity.
- Properties specific to one standard deployment option, such as local Java DB, are added by XML file-based mapping (`orm.xml`), potentially overruling annotations on class level. For example, specific attribute converters or database indexes may be added in this way.
- Properties specific to a specific (customer) installation are expected in the file `ppe-schema-orm.xml`. In particular, this may contain the database schema if not yet specified in the database connection URL. In the case of a local deployment, this file is not relevant.
- Properties specific to customer extensions are expected in additional `orm` files that are specified using the configuration property `sap.dataaccess-common.custmappingresources`.

Multi-Step JPA Property Definition

The JPA properties, such as the configuration properties for the JPA provider, are treated in a similar way as the definition of JPA entities. They are expected in the following three files:

- A file for JPA properties independent of the deployment (SAP owned)
- A file for JPA properties dependent on the deployment (SAP owned)
- A file for customer-specific configuration (empty in SAP shipment)

These files are specified using Spring configuration properties (see below). If a parameter appears in more than one file, the standard Spring logic is executed to merge properties using the `<util:properties>` tag.

Support of JPA Entity Extensions

The entities provided by SAP support the addition of fields to existing JPA entities without replacing or extending the corresponding Java classes. This is achieved using the concept of virtual properties offered by `EclipseLink`. Technically, additional attributes or relations of the JPA entity are stored in a map that can be accessed by dedicated set- and get-methods. Both the attribute name and the property that it is a virtual attribute are specified externally in an `orm.xml` file. Therefore, you can use customer-specific mapping files, such as `ppe-local-orm.xml`.

Consider the following example that introduces the attribute `zzUpSellingCode` as another database column `ZZUP_SELL_TCD`.

Adding a virtual attribute via `ppe-local-orm.xml`

```
<entity-mappings
  xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.eclipse.org/eclipselink/xsds/persistence/orm
http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_4.xsd"
  version="2.4">

  <!-- ... -->
  <entity
    class="com.sap.ppengine.dataaccess.promotion.common.entities.PromotionImpl">
    <attributes>
      <!-- Attribute name is zzUpSellingCode, type is String. Access is virtual -->
      <basic name="zzUpSellingCode" attribute-type="String" access="VIRTUAL">
        <!-- This maps to ordinary column name -->
        <column name="ZZUP_SELL_TCD" />
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```

```

        <!-- Name of setter and getter method for this attribute -->
        <access-methods get-method="get" set-method="set" />
    </basic>
</attributes>
</entity>
</entity-mappings>

```

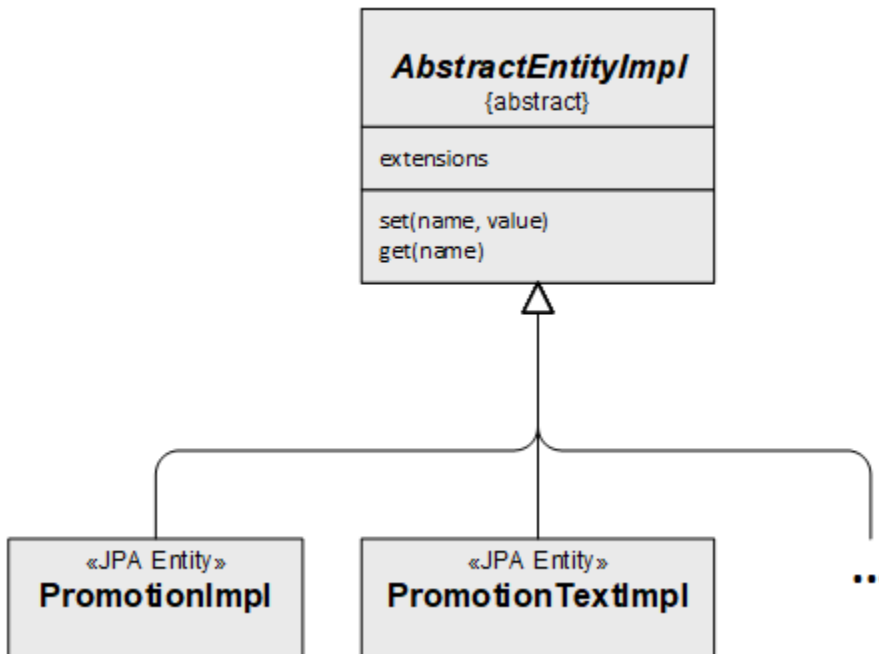


This attribute is an ordinary column in the database. For example, you can define database indexes on it, as for any other column. Furthermore, it is possible to use the virtual attribute like any other attribute in named queries, and so on.



How you add this column to the database depends on the deployment scenario. The PPS takes care of the creation of this column in a local deployment (such as in SAP Hybris Commerce). In a central deployment that runs on an ABAP-owned database, the field must be created explicitly using the ABAP Data Dictionary (SE11). This is controlled in the corresponding deployment specific modules, such as `dataaccess-localdb` or `dataaccess-ddf`.

The virtual attributes are inherited by the common base class **AbstractEntityImpl**.



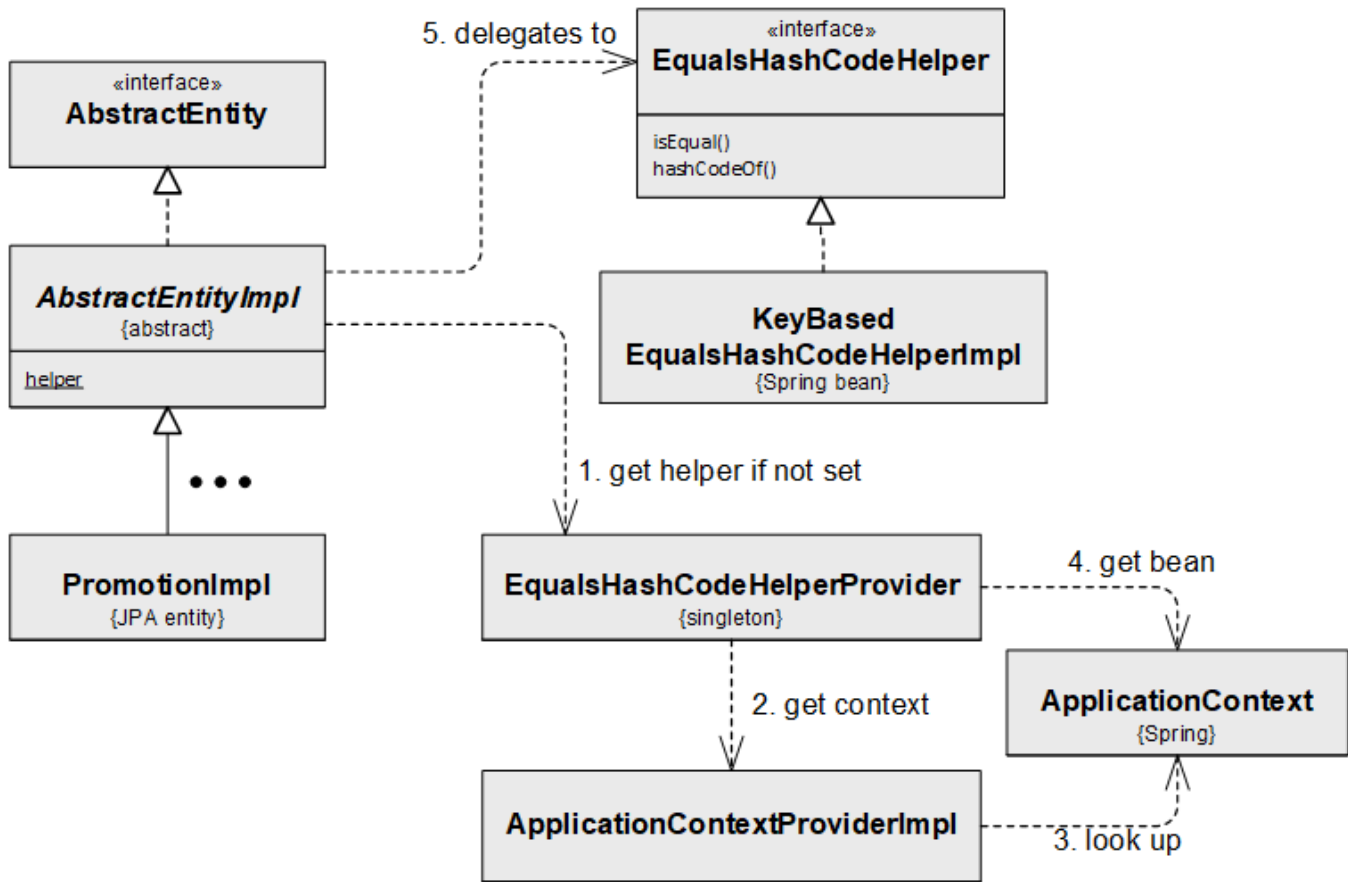
More examples of how to extend entities using this concept are given in the chapter *OPP Extensibility*.

equals() and hashCode() for JPA Entities

The **equals()** and **hashCode()** methods are used in many places in a Java application. By default, the provided JPA entities implement the following behavior:

- Two entities are equal if they have the same type and if they have equal keys.
- Two entity keys are equal if they have the same type and all their components are equal.
- The hashCode of a JPA entity is the hashCode of its key.
- The hashCode of a JPA entity (compound) key is calculated from the hashCodes of its components.

However, you might want to have a different logic in these methods. Since there are no plans to replace the provided SAP JPA entities on the customer side, not even using subclasses, it is not possible to reimplement the standard logic by overriding the methods within a subclass of the corresponding entity. To allow extensions of the standard logic, these methods are implemented as follows:



Each JPA entity inheriting from **AbstractEntityImpl** has a (shared) static attribute "helper" of the type **EqualsHashCodeHelper**. Within **AbstractEntityImpl**, the **equals()** and **hashCode()** methods simply delegate the work to this helper. Since the helper attribute is not managed by JPA, it is determined using the class **EqualsHashCodeHelperProvider** if not yet set. This is a wrapper that gets the PPS Spring application context and retrieves the Spring bean with the fixed name **sapJpaEqualsHashCodeHelper**. In the default shipment this is a class of type **KeyBasedEqualsHashCodeHelperImpl**.

Therefore, the **equals()** and **hashCode()** logic can be redefined by replacing a Spring bean.

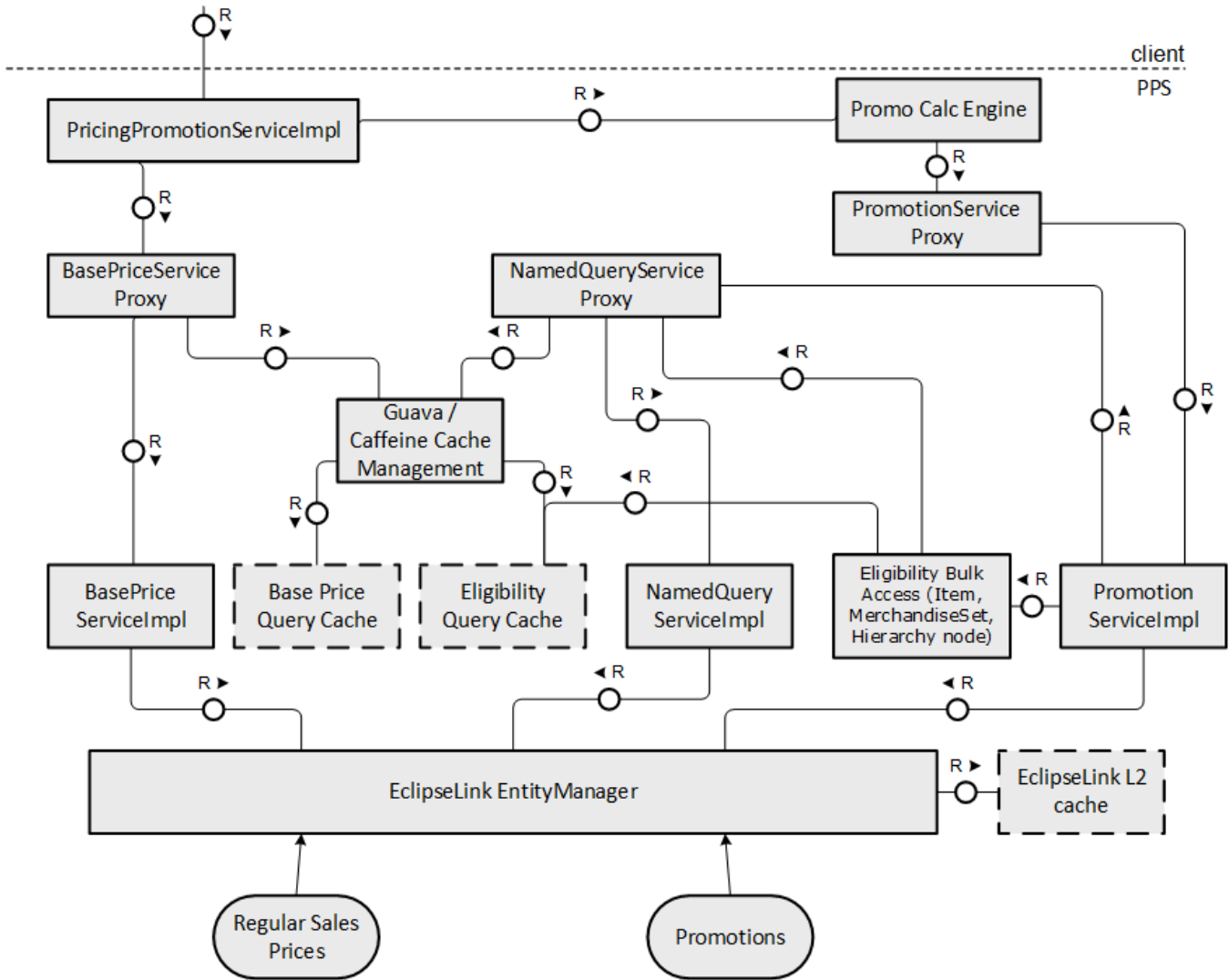
Caching

Regular prices and OPP promotion have to be cached in order to achieve good performance. During the processing of a price calculation request, the number of accesses to information about regular prices and OPP promotions can be high. To avoid cross-system communication and to free the database server from additional load, caching is done in the application (for each Spring application context).

The following requirements were considered for the caching strategy:

- It must use well-proven, fast technology.
- It must provide consistent results during the processing of one price calculation request. This is particularly relevant for the promotion as a complex object stored in many entries of several database tables.
- It must be easily configurable to support installation-specific needs.
- It should be possible to replace the cache provider.

The following figure illustrates how regular prices and OPP promotions are cached on the server side. The **ItemPriceDerivationRuleEligibilityCacheAwareBulkAccessorImpl** that handles the bulk access of **ItemPriceDerivationRuleEligibilities** is introduced with PPS 3.0.



The following three types of caches are used:

- The EclipseLink level 1 cache that is simply the persistence context bound to the transaction created for each price calculation request. The persistence context is attached to the entity manager.
- The EclipseLink level 2 cache that holds the JPA entities of a complete OPP promotion (apart from assigned business units because they are not needed to calculate the OPP promotions once it is known that they are relevant). This is attached to the entity manager factory that exists once in a PPS application context.
- Caches holding the results of named queries. These are defined using Spring cache abstraction (see <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html>). In the standard shipment, Google Guava Cache (see <https://github.com/google/guava/wiki/CachesExplained>) is used as an implementation. Two separate cache regions are offered:
 - A cache region to hold the results of a single price lookup
 - A cache region to hold the results of search queries for eligibilities based on information about the corresponding shopping cart

Caching Regular Prices

Single record look-up results are cached using Spring cache abstraction. This is done by adding the corresponding annotation:

Caching the result of single price look-up

```
public class BasePriceServiceImpl implements BasePriceService {

    @Cacheable(value = DefaultCacheSettings.CACHE_REGION_BASEPRICE)
    public List<BasePrice> getBasePriceForProduct(final String itemId,
        final String businessUnitId, final String businessUnitType,
        final boolean isNet, final String uomCode, final Timestamp timestamp) {
        // ...
    }
}
```

This leads to the creation of a Spring-managed proxy class (**BasePriceServiceProxy**) during the creation of the application context. This class delegates the price look-ups to the cache manager. The logic of the class **BasePriceServiceImpl** is called only if a cached entry cannot be found. Likewise, the result of the price look-up is automatically placed into the Spring-managed cache in the event of a cache miss. The cache implementations (for example, the classes responsible for offering and updating the cache) are defined using Spring beans:

Defining caches in Spring

```
<!-- Cache for regular prices - aware of the underlying cache provider -->
<alias name="sapDefaultBasePriceCache" alias="sapBasePriceCache" />
<bean id="sapDefaultBasePriceCache" class="org.springframework.cache.guava.GuavaCache">
  <constructor-arg
    value="#{T(com.sap.pengine.dataaccess.promotion.common.entities.DefaultCacheSettings).
CACHE_REGION_BASEPRICE}" />
  <constructor-arg>
    <bean factory-bean="sapBasePriceCacheBuilder" factory-method="build" />
  </constructor-arg>
</bean>

<!-- Cache for eligibility references - omitted here -->

<!-- Cache for named queries. Currently all named queries share a common result cache -->
<bean id="cacheManager"
class="com.sap.pengine.core.spring.impl.SwitchableCacheManager">
  <constructor-arg value="{sap.dataaccess-common.cachenamedqueries}" />
  <property name="caches">
    <set>
      <ref bean="sapBasePriceCache" />
      <ref bean="sapEligibilityCache" />
    </set>
  </property>
</bean>

<!-- Builder for cache of promotional information omitted here -->

<!-- Builder for cache of base prices: Create google guava cache with dedicated
spec -->
<alias name="sapDefaultBasePriceCacheBuilder" alias="sapBasePriceCacheBuilder" />
<bean id="sapDefaultBasePriceCacheBuilder" class="com.google.common.cache.CacheBuilder"
  factory-method="from">
  <constructor-arg value="{sap.dataaccess-common.basepricecachespec}" />
</bean>
```

The result of a price look-up is not a managed entity. Inconsistencies between several calls within one price calculation request are avoided by reading each price only once. This is ensured by the class **BasePriceReaderImpl**.



Bulk accesses for regular prices are also offered. However, the behavior slightly differs between the PPS releases: With PPS 1.0, the query completely bypasses the cache for regular prices. As of PPS 2.0, the database query contains only those products/uom codes that are not yet in the cache. In addition, the query result is added to the cache, leading to faster processing in the event of cache misses and therefore more robust behavior.

Caching Promotional Information

The caching of promotional information is more complex than the caching of regular prices, since different parts of the OPP promotion are retrieved by the promotion calculation engine in several steps. Once one part of the OPP promotion is read, it must be ensured that the other parts of the OPP promotion are consistent. At the beginning, a search is made for OPP promotions with the requested eligibilities. Although the class **PromotionServiceImpl** offers methods to find the corresponding eligibilities, it simply delegates the work to the class **NamedQueryServiceImpl**, which is located behind the **NamedQueryServiceProxy**. Therefore, the same approach is applied as for reading the regular prices. With this approach, the result of this search is stored in the eligibility query cache. This is configured via Spring cache abstraction and implemented by Google Guava, (starting with PPS 4.0 Caffeine Cache) as for the regular price cache. However, only a very limited amount of information is read in this case, not the full eligibility. All the OPP promotions for the eligibilities found are read by key (apart from the assigned business units). The assumption is that if an eligibility is found for an OPP promotion, the OPP promotion will become effective soon. This means not necessarily within this price calculation request but in one of the following ones if all eligibilities for the OPP promotion are met. When the OPP promotion is read, the JPA L2 cache is automatically used by the JPA provider. Its content is shared by several price calculation requests. If the promotion to be read is already in the L2 cache, the database is not accessed. In the case of a database access, the L2 cache is updated automatically.

Since the results of the eligibility search and the OPP promotions are stored in different caches and due to the possibility of cache eviction in the L2 cache, it must be ensured that data inconsistencies are detected and resolved. This is done by storing the time stamp of the last write access to any part of the OPP promotion both on promotion level, such as in the L2 cache, and eligibility level, such as in the query cache. This time stamp is introduced by the class **ChangeAwareEntity**, which is a super class to all promotional entities except for **BusinessUnitAssignmentImpl**.

The following logic is implemented:

Case #	Description	Action
1	The time stamp of the eligibility is more recent than the time stamp of the OPP promotion (in cache) or the time stamps of OPP promotion subentities are inconsistent.	The promotion is read again from the database
2	The OPP promotion referred to by the eligibility does not exist or is not active (any more).	Eligibility is skipped
3	The time stamp of the eligibility is the same or older than the time stamp of the OPP promotion.	Eligibility of the OPP promotion is returned if it still exists and has the expected type

The action for *case #3* may look wrong if the time stamp differs, since the returned eligibility might not have the requested content any more. However, existing eligibilities are not reused when a DDF offer is transformed into an OPP promotion. Instead, new eligibilities with new keys are created. Therefore, existing eligibilities are not updated.

Activating the JPA L2 cache for the OPP promotion is done explicitly because the shared cache mode is set to **ENABLE_SELECTIVE** by default:

```
sap.dataaccess-common.sharedcachemode=ENABLE_SELECTIVE
```

All promotional entities except for the business unit assignment are defined as cacheable, using the default settings. The excerpt below shows how this is done for the promotion header:

Making PromotionImpl cacheable

```
@Entity
@Table(name = DBTables.PROMOTION)
@Cacheable
@Cache
public class PromotionImpl extends ChangeAwareEntity implements Promotion {
    // the attributes etc.
}
```

It is not usually necessary to make changes here. However, it is possible to change the settings of the L2 cache by setting the corresponding JPA properties (see Spring property **sap.dataaccess-common.custjpapropertieslocation**):

Adjust cache settings in the JPA properties file

```
# Default cache type & size
eclipselink.cache.type.default=SoftWeak
eclipselink.cache.size.default=1000
# Do it differently for promotion header - just as an example!
eclipselink.cache.type.com.sap.ppengine.dataaccess.promotion.common.entities.PromotionImpl=Soft
eclipselink.cache.size.com.sap.ppengine.dataaccess.promotion.common.entities.PromotionImpl=10000
```



The **PromotionServiceProxy** is created by Spring in order to handle transactions automatically.

Cache Keys

The following keys are used for the various objects stored in the Spring-managed cache:

- Regular price: Business Unit ID, Business Unit Type Code, Timestamp, Unit Of Measure Code, Item ID, Net Flag
- Common to all eligibility references: Business Unit ID, Business Unit Type Code, Timestamp, Status Code, Lineitem Mode, Identification of Eligibility Type (currently: name of cached method). Note that the business unit type code is only cached from PPS 3.0.
- In addition for TotalPurchaseMarketBasket Eligibility: ./.
- In addition for Coupon Eligibility: Coupon ID
- In addition for Customer Group Eligibility: Customer Group ID
- In addition for Item Eligibility: Item ID, Unit Of Measure Code
- In addition for Merchandise Hierarchy Eligibility: Node ID, Node ID Qualifier
- In addition for the Manual Eligibility: Trigger Type, Trigger Value
- In addition for Merchandise Set Eligibility: Item ID - and not the list of assigned merchandise hierarchy nodes(!). It is assumed that the PPS clients have access to the same master data and provide identical hierarchy node assignments in the price calculation request.
- In addition for the OtherEligibility extension: Eligibility Type, additional parameters



The following values are not part of the cache key since they are expected to be constant for each PPS installation: SAP Client, Logical System.

If you want to adjust the cache key, you can replace the standard Spring beans for cache key generation. For example, in a custom key generator it would be possible to access additional information stored in the PPS context, such as the logical system.

Prefetch of Price Derivation Rule Eligibility References

As of PPS 3.0, you can prefetch eligibility references. Depending on the number of line items in the price calculation cart, it may make sense to prefetch (via a bulk access) certain eligibility references at the very beginning of the price calculation. If the eligibility reference cache is not completely filled, this avoids many single selects, leading to improved performance. In particular, this helps for eligibilities that are based on the individual item ID, such as the Item eligibilities and the **MerchandiseSet** eligibilities. For other eligibility types, such as the **MerchandiseHierarchy** or the **CouponEligibility**, the number of possible distinct values across several requests (in other words, the number of different Merchandise Hierarchy nodes or coupon IDs) is much lower, leading to a much faster population of the caches. Therefore, such eligibilities are not considered.

However, the bulk selection is not as specific as the corresponding single selects. For the Item eligibilities, many different UOM codes reduce the selectiveness of the database queries. For **MerchandiseSet** eligibilities, the individual assignment of an item to merchandise hierarchy nodes gets lost. In rare cases, this can lead to a deterioration in performance. Therefore, this optimization can be turned on or off using a threshold. The threshold for this prefetch is defined by the configuration property **sap.dataaccess-common.bulkitemelithreshold** for Item eligibilities and **sap.dataaccess-common.bulkmerchsetelithreshold** for **MerchandiseSet** eligibilities. When this threshold is reached, the corresponding eligibilities that are not already cached are read with one database query (for each type).

Afterwards, the results are added to the cache in a way that they can be retrieved using single access from the cache later on during price calculation.

In addition to the threshold configuration property, this prefetch also needs the property **sap.dataaccess-common.cachenamedqueries** to be set to true.

The bulk access of eligibility references is realized as separate plugin implementations for plugin interface **com.sap.pengine.api.plugin.PromotionServiceInitialization**.

Support of Weaving

As of PPS 2.0, weaving is supported. The EclipseLink feature of weaving the JPA entities (load-time weaving) leads to an improved performance, for example, by reading the promotional entities in a more efficient manner from the database. This approach performs the weaving of entities during startup of the application. As a consequence, possible customer extensions automatically benefit from weaving and it is not necessary to recompile JARs. Weaving is enabled using the Spring profile **sapweaving**. This must be set as an environment variable when the corresponding (Web) application is started:

Enabling weaving

```
-Dspring.profiles.active=sapweaving
```

Load-time weaving has some requirements of the runtime environment. This environment is prepared by the Spring class **org.springframework.context.weaving.DefaultContextLoadTimeWeaver**. This automatically checks whether the classloader supports load-time weaving and supports recent versions of Tomcat 8 or later (necessary for the XSA-based PPS).



The following issues are known to occur with weaving:

- If weaving is enabled, all weaving features are activated by default. As a result, potentially existing constructors or field initializers of JPA entities (that are not recommended) are no longer called. If this leads to issues, it is possible to selectively disable specific weaving features by setting the corresponding JPA properties. For instance, setting `eclipselink.weaving.internal=false` can help to reduce these types of issues.
- In general, weaving supports the extension of JPA entities using virtual access methods. However, according to the EclipseLink documentation, weaving is not supported when virtual access methods are used with `OneToOne` mappings.
- Load-time weaving of a JPA entity takes place when the corresponding Java class is loaded. If it has been loaded before load-time weaving is activated (for example, within a JUnit test), it cannot be woven anymore. As a result, **NoSuchMethodError** exceptions due to incomplete weaving will be thrown, causing the application to stop working.

As a result of the introduction of weaving, how the promotional entities are read has changed with PPS 2.0. In PPS 1.0, the promotion for the corresponding eligibility was read using the fetch type **EAGER**. In PPS 2.0, the fetch type **LAZY** is used.

For more information about weaving, see <http://www.eclipse.org/eclipselink/documentation/2.6/solutions/testingjpa004.htm>.

Support for Read-Only Transactions

As of PPS 2.0, promotional information can also be read in a read-only mode. In this mode, no change tracking of the JPA entities is done. This results in an optimized resource consumption. Consequently, no changes to the read JPA entities are saved to the database once the transaction is committed. The request of a read-only transaction is controlled using a dedicated attribute of the PPS context. This is set to TRUE in the case of a price calculation request.

Depending on the cache isolation set for the corresponding entities, read-only transactions may have no "working copy" of the entity in the persistence context during the price calculation. This can lead to consistency issues in the case of concurrent read and write operations on the same entity. If you create custom subentities of an OPP promotion, they must fulfill the following constraint:



The subtentities of an OPP promotion must be immutable objects if they are critical to the correct price calculation. If a new version of an OPP promotion is created, the subtentities must be new objects with new keys replacing the subtentities of the former version.

The following subtentities of an OPP promotion are not considered as critical:

- The promotion texts (because they do not contain information that influences the price itself)
- The assigned business units (because they are not taken into account once a promotion is considered as relevant and because they contain only key fields)

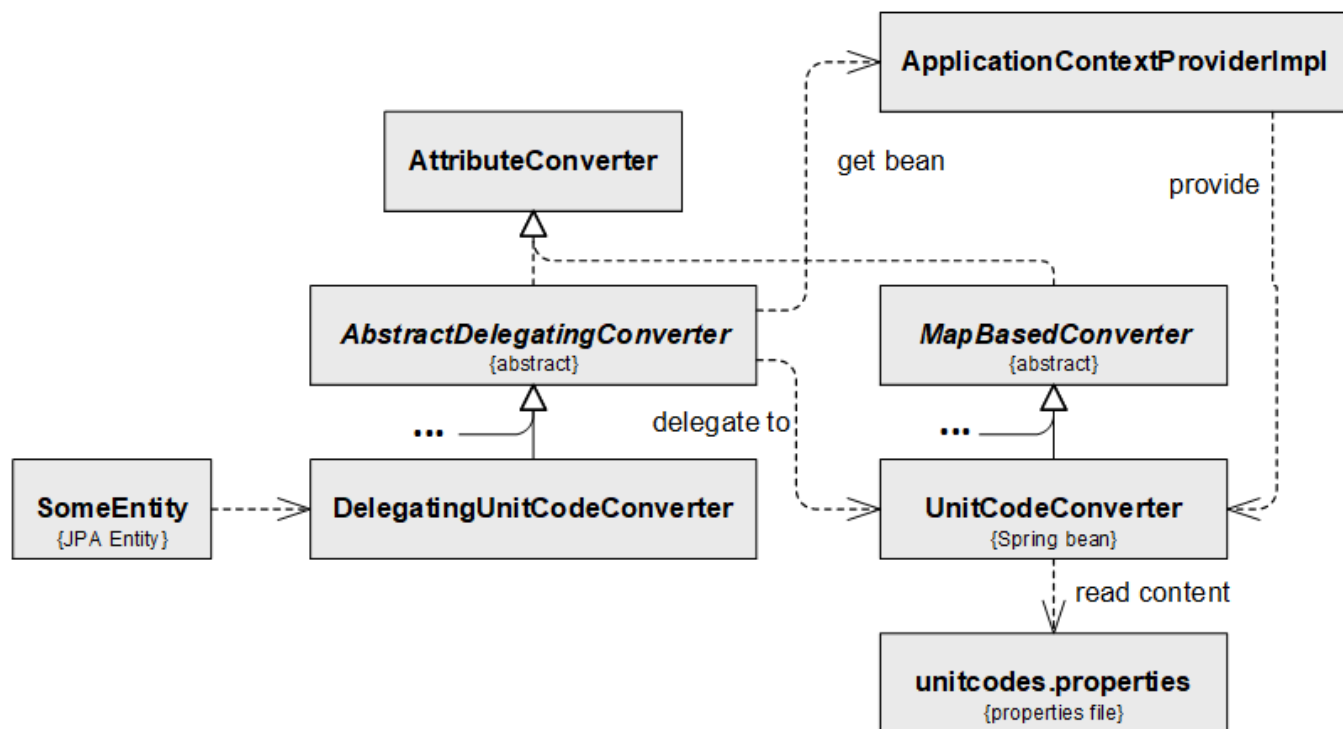
There is no change of system behavior for regular prices. They are not cached in the JPA L2 cache.

Code Conversion

Following SAP standards, *amounts*, *quantities* and *texts* refer in the database to the SAP internal code lists. This is also true for the encoding of the business unit type. The following values are allowed for the database representation of these codes:

Code	Same Table Field/ABAP Domain
Language	T002-SPRAS
Currency and Decimals	TCURC-WAERS, TCURX-CURRDEC
Unit of Measure	T006-MSEHI
Business Unit Type	Domain /DMF/LOCATION_TYPE_CODE

This schema is not generally known to an external client. Therefore, the PPS works from the client API down to the JPA entities with the ISO schema of these codes. Since there is no ISO representation for the business unit type, the ARTS schema is used. The translation between the database representation and the JPA entity representation is realized using JPA **AttributeConverter** implementations. However, it might be desirable to configure this mapping, particularly in view of the unit of measure codes. The following figure shows how to configure this mapping for unit codes:



The class **DelegatingUnitCodeConverter** is a JPA attribute converter. The lifecycle of an **AttributeConverter** is now managed by the JPA provider not by Spring. In particular, only a default constructor is supported and does not offer further configuration. Therefore, this converter simply delegates the actual work to a Spring bean with a fixed alias. The Spring bean is retrieved using the class **ApplicationContextProviderImpl**. This delegate bean also implements **AttributeConverter** but has the full support of Spring offerings, such as configuration parameters, support of properties files, and so on. Hence, the code mapping can be specified using a properties file whose location can be specified by a Spring configuration property.

At first sight, it might look surprising that the attribute converter does not simply access the content of the corresponding customizing tables and instead reads the content of a properties file, leading to double maintenance if additional unit codes are introduced. However, reading another database table within an attribute converter again requires access to a JPA entity manager, making the implementation of the attribute converter much more difficult and its performance likely worse.

Handling of Currencies and Amounts

- When the JPA entity is converted to the database representation, leading zeros are added up to a length of 60 characters.
- When the database is converted to the JPA entity representation, the prefix of the ID is removed so that the result has a fixed number of digits that is configurable via property **sap.dataaccess-common.fixednumberofplacesinproductid**.
- If this parameter is zero, all leading zeros are removed, not considering a fixed length.

Handling of Language-Specific Information

The client API for price calculation allows the specification of a requested language in which language-dependent information (promotion descriptions, external action price rule texts) is returned to the caller. This information is stored in the PPS context (bean **sapContext**) and evaluated once the corresponding parent object, such as the Promotion or the **ExternalActionPriceRule**, is requested. If language codes are specified and the resulting set of language-dependent information differs from the original set, the caller gets a detached copy of the parent object for each method call. This copy contains only the requested information.



Requesting and returning language-specific information is supported as of PPS 2.0.

SAP Client and Logical System


An SAP client and logical system must be specified in order to uniquely identify which information is to be retrieved from the database. This is particularly true when regular prices are read with a given external product and business unit (location) ID. However, this information is not provided externally as part of the request for the price calculation. Therefore, this information must be provided via Spring configuration properties. They are stored in the PPS context that is globally visible with request processing via the bean **sapContext**.



If you need to support several values for a logical system, SAP clients or business unit types for each installation, you also need to adjust the cache key generators since this information is not considered to be part of the cache key for named queries by default.

Beans

ID	Alias	Description
sapDefaultPersistenceAnnotationBeanPostProcessor	sapPersistenceAnnotationBeanPostProcessor	Spring postprocessor enabling automatic transaction management via annotation @Transactional
sapDefaultJpaProperties	sapJpaProperties	Properties bean holding the JPA properties Refers to configuration properties: <ul style="list-style-type: none"> • sap.dataaccess-common.defaultjpapropertieslocation • sap.dataaccess-common.jpapropertieslocation • sap.dataaccess-common.custjpapropertieslocation
sapDefaultEntityManagerFactory	sapEntityManagerFactory	Spring-based entity manager factory, configurable via properties instead of a single persistencexml file
sapDefaultTransactionManager	sapTransactionManager	Spring-based transaction manager
sapDefaultJpaDialect	sapJpaDialect	Spring JPA Dialect "EclipseLink" to be used for transaction manager
sapDefaultJpaVendorAdapter	sapJpaVendorAdapter	Registers EclipseLink as a JPA provider
sapAbstractPersistenceService	./.	Base class for all persistence services
sapDefaultPromotionService	sapPromotionService	Central service for accessing promotional information from database
sapDefaultNamedQueryService	sapNamedQueryService	Service for reading promotional information from database via named queries. Used by sapPromotionService .
sapDefaultBasePriceService	sapBasePriceService	Central service for accessing regular price information from database
sapDefaultDbContextInitializer	sapDbContextInitializer	Initializer of the PPS context adding parameters relevant for accessing the database (client, business unit type, logical system)
sapDefaultLanguageCodes	sapLanguageCodes	Default mapping to translate between SAP and ISO language codes
sapDefaultCurrencyCodes	sapCurrencyCodes	Default mapping to translate between SAP and ISO currency codes

sapDefaultUnitCodes	sapUnitCodes	Default mapping to translate between SAP and ISO unit of measure codes
sapDefaultCurrencyDecimals	sapCurrencyDecimals	Default decimals of SAP currencies
sapDefaultCurrencyMappingFactory	sapCurrencyMappingFactory	Default factory to create mapping from SAP currency codes to tuple <ISO code + scaling factor>. Uses beans sapDefaultCurrencyCodes and sapDefaultCurrencyDecimals.
sapDefaultCurrencyCodesWithScale	sapCurrencyCodesWithScale	Default mapping between SAP currency codes and CurrencyWithDecimals. Created by the bean sapCurrencyMappingFactory.
sapDefaultBusinessUnitLocationTypes	sapBusinessUnitLocationTypes	Default mapping file to translate between SAP and ARTS business unit type codes
sapDefaultLanguageCodeConverter	sapLanguageCodeConverter	Converter between SAP and ISO language codes accessing the mapping file Called by corresponding JPA attribute converter
sapDefaultCurrencyCodeConverter	sapCurrencyCodeConverter	Converter between SAP and ISO currency codes accessing the mapping file Called by corresponding JPA attribute converter. Only to be used if scaling of amounts is not needed.
sapDefaultCurrencyWithScaleConverter	sapCurrencyWithScaleConverter	Converter between SAP currency and CurrencyWithDecimals. Uses sapCurrencyCodesWithScale.
sapDefaultUnitCodeConverter	sapUnitCodeConverter	Converter between SAP and ISO unit of measure codes accessing the mapping file Called by corresponding JPA attribute converter
sapDefaultBusinessUnitLocationTypeConverter	sapBusinessUnitLocationTypeConverter	Converter between SAP and ARTS business unit type codes accessing the mapping file Called by corresponding JPA attribute converter
sapDefaultInternalProductIDConverter	sapInternalProductIDConverter	Converter between SAP CAR internal representation of numeric product IDs and their (internal) representation used by the PPS client
cacheManager	./.	Spring cache manager introducing caches for promotional information (OPP promotion eligibilities) and single accesses for regular prices
sapDefaultPromoCacheBuilder	sapPromoCacheBuilder	Cache builder for storing OPP promotion eligibility keys using Google Guava as cache implementation
sapDefaultBasePriceCacheBuilder	sapBasePriceCacheBuilder	Cache builder for regular prices using Google Guava as cache implementation
sapDefaultEligibilityCache	sapEligibilityCache	Spring wrapper for the cache for regular prices Before PPS 2.0 this was an anonymous Spring bean
sapDefaultBasePriceCache	sapBasePriceCache	Spring wrapper for the cache for promotional information (eligibility references) Before PPS 2.0 this was an anonymous Spring bean
sapDefaultEliCacheKeyGenerator	sapEliCacheKeyGenerator	Enhancement of the default Spring cache key generator considering the name of the method in addition to the provided arguments. Necessary if different read methods have the same arguments but should provide different results.  As of PPS 3.0, this bean is contained in the core module.
sapDefaultPriceCacheKeyGenerator	sapPriceCacheKeyGenerator	Cache key generator used when caching regular prices.  As of PPS 3.0, this bean is contained in the core module.
sapDefaultItemEligibilityBulkAccessor	sapItemEligibilityBulkAccessor	Default implementation for the item price derivation rule eligibility bulk access (as of PPS 3.0)
sapDefaultMSetEligibilityBulkAccessor	sapMSetEligibilityBulkAccessor	Default implementation for the MerchandiseSet price derivation rule eligibility bulk access (as of PPS 3.0)
sapDefaultJpaEqualsHashCodeHelper	sapJpaEqualsHashCodeHelper	Default implementation of equals() and hashCode() for JPA entities
sapDefaultLoadTimeWeaver	sapLoadTimeWeaver	If weaving is active (as of PPS 2.0): provides Environment for EclipseLink weaving

r		
sapDefaultSelectionIntervalCreator	sapSelectionIntervalCreator	<p>As of PPS 2.0:</p> <p>Converter of a provided time stamp for price calculation into an interval. This is used for searching eligibilities only. In order to be found, the corresponding promotion must intersect with this interval. By default, the whole day of the given time stamp is considered as an interval.</p> <p>See also bean sapTimeResolutionReducer in the core module.</p> <p><u>Example:</u> 2017-30-03 12:34:56 is converted into interval 2017-30-03 00:00:00 (inclusive) until 2017-03-31 00:00:00 (exclusive)</p>
sapDefaultPromoCyclesChecker	sapPromoCyclesChecker	<p>Available as of PPS 4.0.</p> <p>Checks an OPP promotion for cycles in the eligibilities or merchandise sets. Implementation of com.sap.ppengine.api.plugin.Validation</p>


Required Beans

ID/Alias	Comment
sapDataSource	Provides database connection

Configuration Properties

Name	Description	Default Value	Comment
sap.dataaccess-common.persistence.unitname	Name of the JPA persistence unit to be used for reading and writing OPP promotion and regular price information	<ul style="list-style-type: none"> SAPDefaultPU 	
sap.dataaccess-common.sharedcachemode	Defines the JPA entities for which a JPA Level2 cache is to be used	<ul style="list-style-type: none"> ENABLE_SELECTIVE 	Explicit enablement of caching per entity
sap.dataaccess-common.defaultjapropertieslocation	Location of the default JPA properties (to be used independent of the underlying database) in Spring resource syntax	<ul style="list-style-type: none"> classpath: META-INF/defaultjaproperties 	
sap.dataaccess-common.weavingdefaultjapropertieslocation	Location of the default JPA properties (to be used independent of the underlying database) in Spring resource syntax. Used if weaving is active.	<ul style="list-style-type: none"> classpath: META-INF/weavingdefaultjaproperties 	Weaving available with PPS 2.0 or later
sap.dataaccess-common.custjapropertieslocation	Location of customer-specific JPA properties in Spring resource syntax	<ul style="list-style-type: none"> classpath: META-INF/empty.properties 	Example value: classpath: META-INF/myjapros.properties
sap.dataaccess-common.packagestoscan	Comma-separated list of package names to be scanned for JPA entities or attribute converters	<ul style="list-style-type: none"> com.sap.ppengine.dataacce 	Should not be changed

		<ul style="list-style-type: none"> ss.promotion.common.entities,com.sap.pengine. dataaccess.converter.common,com.sap.pengine. dataaccess.baseprice.common.entities 	
sap.dataaccess-common.custpackagestoscan	Comma-separated list of additional package names to be scanned for JPA entities or attribute converters	<ul style="list-style-type: none"> <empty> 	Example value: ,com.mycompany.myentities (note the leading comma)
sap.dataaccess-common.mappingresources	Comma-separated list of mapping resource files overruling/adding to annotations defined in the classes for the JPA entities. Must be on the Java classpath.	<ul style="list-style-type: none"> META-INF/orm.xml,ppe-schema-orm.xml 	Should not be changed
sap.dataaccess-common.custumappingsresources	Comma-separated list of <u>additional</u> mapping resource file intended for customer-specific extensions	<ul style="list-style-type: none"> <empty> 	Example value: ,ppe-local-orm.xml (note the leading comma)
sap.dataaccess-common.cachenamequeries	Switch for caching the result of named queries	<ul style="list-style-type: none"> true 	Set this to false, if you always want to access updated regular prices and OPP promotions. Disabling the L2 cache for OPP promotions should not be needed.
sap.dataaccess-common.promocachespec	Cache specification of the cache for promotional information read via named queries as defined by Google Guava	<ul style="list-style-type: none"> maximum Size = 10000, expireAfterAccess = 10m, expireAfterWrite = 20m, initialCapacity = 100 	Only relevant if sap.dataaccess-common.cachenamequeries = true With PPS release 4.0, the new parameter initialCapacity defines the initial capacity value for the caffeine promotion cache.
sap.dataaccess-common.basepricecachespec	Cache specification of the cache for single records of regular prices read via named queries as defined by Google Guava	<ul style="list-style-type: none"> maximum Size = 10000, expireAfterAccess = 10m, expireAfterWrite = 20m, initialCapacity = 100 	Only relevant if sap.dataaccess-common.cachenamequeries = true With PPS release 4.0 the new parameter initialCapacity defines the initial capacity value for the caffeine base price cache.
sap.dataaccess-common.currencycodeslocation	Location of the mapping file to translate between SAP currency codes in the database and ISO codes used within JPA entities. Spring resource syntax is used.	<ul style="list-style-type: none"> classpath:META-INF/currencycodes.properties 	

sap. dataaccess -common. unitcodeslo cation	Location of the mapping file to translate between SAP unit codes in the database and ISO codes used within JPA entities. Spring resource syntax is used.	<ul style="list-style-type: none"> classpath:META-INF/unitcodes.properties 	
sap. dataaccess -common. languageco deslocation	Location of the mapping file to translate between SAP language codes in the database and ISO codes used within JPA entities. Spring resource syntax is used.	<ul style="list-style-type: none"> classpath:META-INF/languagecodes.properties 	
sap. dataaccess -common. businessun itlocationty pelocation	Location of the mapping file to translate between SAP encoding of business unit types (same values as location types of SAP CAR) codes in the database and ARTS codes used within JPA entities. Spring resource syntax is used.	<ul style="list-style-type: none"> classpath:METAINF/businessunitlocationtype.properties 	
sap. dataaccess -common. currencyde cimalslocati on	Location of the properties file containing the number of decimals for SAP currency codes. Only codes for currencies that do not have two decimals are expected in this file.	Not set	Set in dataaccess-ddf or dataaccess-localdb
sap. dataaccess -common. fixednumbe rofplacesin productid	Number of digits of a numerical product ID including leading zeros as provided and expected by the consumer of the PPS	18	Length of the SAP ERP material number is 18
sap. dataaccess -common. db.client	SAP client to use when accessing the database	./.	To be set for each installation
sap. dataaccess -common. logSys	Logical system to use when accessing information having compound key (external ID + logical system), such as the SAP CAR ProductID and LocationID	./.	<p>If the PPS clients provide the master source system ID in the price calculation request, this parameter does not need to be set statically. It is only used as a default if the information is missing in the price calculation request.</p> <div style="border: 1px solid #ffc107; border-radius: 10px; padding: 5px; margin-top: 10px;">  Providing the master data source system ID is possible as of PPS 3.0. </div>
sap. dataaccess -common. defaultBuT ype	Default business unit type to use when reading regular prices and promotional eligibilities	RetailStore	
sap. dataaccess -common. partitionSiz eSqlInState ment	Maximum number of list entries when using IN operator in SQL statements	100	Used, for example, during inbound processing of regular prices
sap. dataaccess -common. bulkitemelit hreshold	Threshold for numbers of line items deciding if an item eligibility prefetch (bulk access) is executed. Note: The overall number of line items (without coupons, and so on) is compared against this threshold - not only the number of items for which the ItemEligibility reference is still missing in the cache.	10	<p>Only relevant if</p> <p>sap.dataaccess-common. cachenedqueries = true</p> <p>Bulk access for Item Eligibilities available with PPS 3.0 or later</p>
sap. dataaccess -common.	Threshold for numbers of line items deciding if a Merchandise Set Eligibility prefetch (bulk access) is executed. Note: The overall number of line items (without coupons, and so on) is compared against this threshold - not only the number of items for which the MerchandiseSet Eligibility reference is still missing in the cache.	10	<p>Only relevant if</p> <p>sap.dataaccess-common. cachenedqueries = true</p>

bulkmerchs etelithresho ld			Bulk access for MerchandiseSet Eligibilities available with PPS 3.0 or later
sap. dataaccess -common. bulkmerchg roupeliithre shold	Threshold for numbers of merchandise hierarchy nodes deciding if a Merchandise Group Eligibility prefetch (bulk access) is executed. Note: The overall number of merchandise hierarchy nodes is compared against this threshold - not only the number of nodes for which the Merchandise Group Eligibility reference is still missing in the cache.	5	Only relevant if sap.dataaccess-common. cachenamedqueries = true Bulk access for Merchandise Group Eligibilities available with PPS 3.0.14 or later
sap. dataaccess -common. overwritewi tholderdata	Switch controlling the behavior when importing OPP promotions. If set to true, the imported promotion replaces the existing promotion on the database, regardless of the value of the changedOn attribute (indicating when this version of the promotion was created). If set to false, the received promotion is only written to the database if it has been changed more recently than the promotion on the database.	true	If the promotions are sent or received in the wrong order, the most recent data are used. It is not possible to resend an old IDoc to revert an unwanted change of a promotion, since the contained promotion will be ignored due to its changedOn value. Available as of PPS 3.0.17.

Dependencies

This module depends on the following modules:

- core
- dataaccess-interface (transitive dependency of core since PPS 3.0)

PPS Module dataaccess-ddf

This module provides the specifics for the data access against the SAP HANA database of the SAP Customer Activity Repository system.

Overview

As mentioned in the documentation for the module **dataaccess-common**, the JPA entities should not depend on the specifics of the underlying database or system that provides the database table. In the case of the central PPS, the database tables are defined using the data dictionary of the SAP Customer Activity Repository system running on an SAP HANA database. This has the following consequences:

- No changes are made to the database schema using JPA - SAP Customer Activity Repository is the leading system. This is done by simply not setting the JPA property **eclipseLink.ddl-generation**.
- The format for time stamps in the database is different from the usual format in a Java environment.
- The format for Boolean values in the database is different from the usual format in a Java environment.
- Running against an SAP HANA database, the JDBC database driver is determined.

Furthermore, the **javax.sql.DataSource** is provided over JNDI when running on SAP HANA XS Advanced.

This module configures the data access accordingly.

Attribute Converters

The conversion between database values and the attributes of JPA entities is realized using an implementation of **javax.persistence.AttributeConverter**. They are declared in the file **META-INF/orm.xml** on the Java classpath.

Boolean Values

In ABAP, there is no dedicated basic type for Boolean values. Instead, this information is usually stored in a character array of length 1 with the following values:

ABAP	Boolean
'X'	TRUE
"	FALSE

This mapping is implemented by the class **com.sap.ppengine.dataaccess.converter.common.AbapBooleanConverter** located in the module **dataaccess-common**.

Time Stamps

In ABAP, time stamps are stored in a packed decimal number. The following time stamps are known:

- A time stamp down to second level (see the domain TZNTSTMP)
- A time stamp down to sub-microsecond level (see the domain TZNTSTMPL)

Only the time stamp with a precision on seconds-level is supported. On the Java side, `java.sql.Timestamp` is usually taken to store time stamps. The mapping between ABAP and Java is done as follows:

ABAP value	Year	Month	Day	Hour	Minute	Second	Nanoseconds
YYYYMMDDHHMMSS	YYYY	MM	DD	HH	MM	SS	0

This mapping is implemented by the class `com.sap.ppengine.dataaccess.converter.common.AbapTimestampConverter` located in the module `dataaccess-common`.

Beans

ID	Alias	Description
sapDefaultDataSource	sapDataSource	Factory bean for the data source looking up JNDI for property <code>java:comp/env/jdbc/DefaultDB</code>
sapDefaultDataSource	sapDataSource	Implementation of the data source reading Spring configuration properties (see below). Only used if the Spring profile <code>development</code> is active, replacing the JNDI variant. This option is not meant for productive use.

Configuration Properties

Name	Description	Default Value	Comment
sap.dataaccess-common.currencydecimalslocation	Location of the properties file containing the number of decimals for SAP currency codes. Only codes for currencies that do not have two decimals are expected in this file.	classpath:/META-INF/currencydecimals.properties	Contains a copy of TCURX
sap.dataaccess-common.db.driverClassName	Name of the JDBC database driver	com.sap.db.jdbc.Driver	Might be changed in a test environment
sap.dataaccess-common.db.url	URL of the database connection	./	To be set in a test environment
sap.dataaccess-common.db.userName	Database user	./	To be set in a test environment
sap.dataaccess-common.db.passWord	Password of the database user	./	To be set in a test environment

Dependencies

This module depends on the following modules:

- dataaccess-common

PPS Module dataaccess-localdb

This module provides the specifics for the access to a local database that is only accessed via JPA.

Overview

As mentioned in the documentation for the module `dataaccess-common`, the JPA entities should not depend on the specifics of the underlying database or system that provides the database table. If the PPS is deployed locally in another hosting application, such as in SAP Hybris Commerce, the following specialties of this module have to be considered:

- The default name of the database tables with the `/ROP/` prefix can lead to issues. Therefore, the prefix `/ROP/` is replaced with the prefix `SAPPS` for all PPS database tables.

- Additional (named) queries and database fields are needed for the inbound of promotional information. In particular, this is a version field for JPA optimistic locking on promotion header level.
- Additional indexes typically needed in (row store) relational databases are needed in order to speed up the process of reading data.
- The database tables are created via JPA.
- If you use an Oracle or a Microsoft SQL Server database, the standard logic to set the lengths of character-like columns is not sufficient because it specifies the length in bytes instead of characters. To overcome this, SAP provides an adjusted implementation of `org.eclipse.persistence.platform.database.DatabasePlatform`. This is automatically checked during startup of the application context.

The first three adjustments are done in the file **orm.xml**.



This module does not provide a `javax.sql.DataSource` as required by the `dataaccess-common` module. It is expected that this is provided by the hosting application.

Indexes

The following table shows the database indexes that are added to the indexes automatically created due to the foreign key relationships that are defined in the JPA entities:

Table	Index	Fields	Unique?
SAPSPROMOTION	SAP_BYFROMDATE	EFFECTIVE_DATE	No
SAPSELIGIBILITY	SAP_BYITEMID	ITEM_ID EFFECTIVE_DATE	No
SAPSELIGIBILITY	SAP_BYNODEID	NODE_ID EFFECTIVE_DATE	No
SAPSELIGIBILITY	SAP_BYTYPECODE	TYPE_CODE EFFECTIVE_DATE	No



The list of database indexes is most likely incomplete for your specific needs. The index that is used if an SQL query is executed depends on the database used and on the amount of data in the corresponding tables. We strongly recommend that you review the database indexes for your specific needs.



When the concept of enhanced product groups (available as of PPS 3.0) is used with a huge amount of product group entries, it could be helpful to create indexes in table `SAPPSMERCH_SET` for the item or the product hierarchy node identifiers.

Configuring the Data Access

Depending on the DB platform that is used by the PPS installation, you have to configure the data access of the PPS. For this, you must proceed as follows:

1. Define the SQL query to remove unwanted foreign key constraints between eligibilities and their parent eligibilities via the configuration parameter `sap.dataaccess-localdb.fkremovalquery`. For more information, see section *Configuration Properties* below.



If you do not set this query, you may encounter unjustified errors during the upload of IDocs.

2. Set the DB platform used by Eclipselink to ensure, for example, that database data types and Java data types are mapped correctly. The DB platform is part of the JPA properties evaluated by Eclipselink. These properties are not directly set as PPS configuration properties. Instead, you need to proceed as follows:

- a. Create a properties file containing these JPA properties and put it on the Java classpath as shown in the following example:

myJpaprops.properties in classpath folder META-INF

```
# Set DB platform to MySQL
# Supported DB platforms are subclasses of org.eclipse.persistence.platform.database.
DatabasePlatform
eclipselink.target-database=org.eclipse.persistence.platform.database.MySQLPlatform
# further examples:
# SQL Server
# eclipselink.target-database=com.sap.ppengine.dataaccess.localdb.SQLServerPlatformOPP
# Oracle
# eclipselink.target-database=com.sap.ppengine.dataaccess.localdb.OraclePlatformOPP
```

```
# SAP HANA
# eclipselink.target-database=org.eclipse.persistence.platform.database.HANAPlatform
# ... further settings
```

b. Add the location of the JPA properties to **ppe-local.properties** to make it known to the PPS as shown in the following example:

JPA properties location in ppe-local.properties

```
# Make location of my JPA properties known to PPS
sap.dataaccess-common.custjpapropertieslocation=classpath:META-INF/myJpaprops.properties
```



SAP does not guarantee that further platforms than SAP HANA and the DB platforms supported with SAP Commerce can be used by the PPS.



DB platform-specific adjustments depend on the database that you use, for example, platform-specific URL parameters to improve performance. For these configuration parameter, see the corresponding database documentation.

Beans

ID	Alias	Description
sapDefaultForeignKeyRemover	sapForeignKeyRemover	Contains functionality to execute the native SQL query specified via configuration property sap.dataaccess-localdb.fkremovalquery . Used to remove a problematic foreign key constraint from the database. Note that this bean itself does not actively execute the query by itself.
sapDefaultValidationQueries	sapValidationQueries	Map of known JDBC drivers and appropriate validation queries for the corresponding DBMS
sapAbstractDataSourceFactory		Abstract base class and bean for factories of DataSources supporting connection pooling with automatic determination of the correct validation query. Can be used by the hosting application where the type of (pooling) DataSource is known. As an example, in newer releases, the local PPS within the sappspricing extension uses a DataSource created by a child of this class.
sapAbstractDbConsistencyChecks		Abstract base class for platform-specific consistency checks. Available as of PPS 3.0.22 / 4.0.10.
sapOracleConsistencyChecks		Performs automatic checks during startup of the application context if you use an Oracle database. Checks if the SAP provided database platform class is configured in the JPA properties of the entity manager factory.
sapSqlServerConsistencyChecks		Performs automatic checks during startup of the application context if you use an MS SQL Server database. Checks if the SAP provided database platform class is configured in the JPA properties of the entity manager factory. Available as of PPS 3.0.22 / 4.0.10.

Required Beans


The following table contains the additional beans that are to be provided if all the dependencies of this module are resolved:


ID/Alias	Comment
sapDataSource	Provides the database access

Configuration Properties

Name	Description	Default Value	Comment
sap.dataaccess-common.	Location of the JPA properties for a local deployment	classpath: /META-INF/	Should usually not be changed - see property sap.dataaccesscommon.custjpapropertieslocation

jpapropertieslocation		dataaccess-localdb-jpaprops.properties	
sap.dataaccess-localdb.connectionpool.initialsize	Initial size of a connection pool if used to access the database via an own connection pool	10	Not used in module
sap.dataaccess-localdb.connectionpool.maxsize	Maximum size of a connection pool if used to access the database via an own connection pool	50	Not used in module
sap.dataaccess-localdb.connectionpool.validationQuery	Validation query used by the connection pool to check if the corresponding connection is still usable	select 1 from INFORMATION_SCHEMA.SYSTEM_USERS	Works for HSQLDB, which is the Hybris default. Other queries are: <ul style="list-style-type: none"> • Oracle - select 1 from dual • DB2 - select 1 from sysibm.sysdummy1 • mysql - select 1 • MS SQL server - select 1 • Postgresql - select 1 • Derby - select 1 • H2 - select 1
sap.dataaccess-common.currencydecimalslocation	Location of the properties file containing the number of decimals for SAP currency codes. Only codes for currencies that do not have two decimals are expected in this file.	classpath:/META-INF/empty.properties	Java-owned database tables store amounts in their natural format
sap.dataaccess-localdb.fkremovalquery	Native SQL query which deletes the foreign key constraint for the parent eligibilities of a given eligibility record. This constraint may cause issues during IDoc inbound processing if the used DBMS does not use deferred foreign key checks. The syntax of this query and the name of the foreign key constraint is DBMS-specific.		To reduce the risk of data corruption caused by a wrong SQL query, it must have a certain format. See class RemoveForeignKeyImpl in case of an issue.

 If you do not set the validation query correctly, the application may not start.

 You only have to set the validation query if one of the following preconditions is met:

- You are using a PPS with patch levels lower than:
 - PPS 3.0.3
 - PPS 2.0.5
 - PPS 1.2.7
 - PPS 1.1.8
 - PPS 1.0.14
- You are using a PPS with a higher patch level than in the list above, but the JDBC driver class is not in the list of known drivers. If the driver class is known, the validation query can be determined automatically using a factory bean having bean **sapAbstractDataSourceFactory** as parent bean.

The same patch levels as mentioned above are also required for the foreign key removal query to take effect.

Dependencies

This module depends on the following modules:

- dataaccess-common

PPS Module idocinbound

This module provides the implementation of IDoc inbound processing for OPP promotions and regular prices.

Overview

If the PPS is deployed locally, for example the PPS is embedded in SAP Hybris Commerce, it accesses its own locally stored data. The module **idocinbound** provides the possibility to receive IDocs holding regular prices and OPP promotions and to update this information on the local database. These IDocs are usually created by the SAP Customer Activity Repository system that contains the central price and promotion repository (PPR). The IDoc inbound supports the following IDoc types and the corresponding message types:

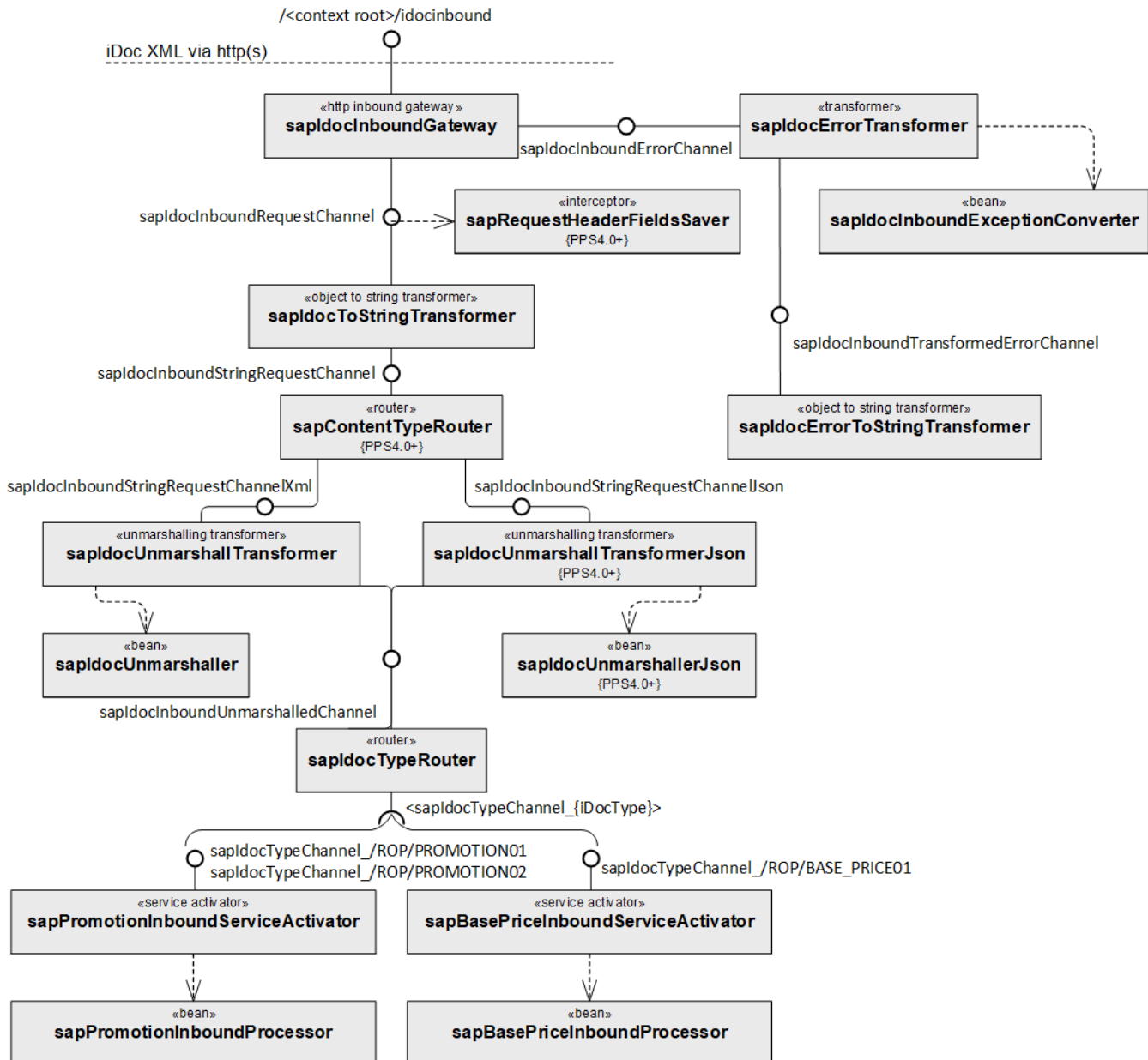
- Regular prices can be processed using IDoc type **/ROP/BASE_PRICE01** (message type **/ROP/BASE_PRICE**)
- OPP promotions can be processed using IDoc type **/ROP/PROMOTION01** or **/ROP/PROMOTION02** (message type **/ROP/PROMOTION**)

These IDocs can be processed with an XML payload and with a JSON payload (available as of PPS 4.0).

The IDoc inbound processing processes the incoming requests synchronously. No staging of requests is executed. Furthermore, only very basic consistency checks of the IDoc content are performed.

Spring Integration Process Definition

The inbound processing is realized based on [Spring Integration](#). The following figure shows the process flow:



In detail, the following is done:

1. The IDoc inbound processing is triggered via an HTTP POST request to the context path `/idocinbound`.
2. The incoming request is handled by a Spring Integration **http-inbound-channel-adapter** (`sapIdocInboundGateway`). This integration is connected to the following Spring Integration channels:
 - a. The `sapIdocInboundRequestChannel` propagates the request to the next processing stage. As of PPS 4.0 the interceptor `sapRequestHeaderFieldsSaver` is attached to this channel to store the HTTP header attributes of the incoming request in the PPS context. This is done to allow proper handling of the Accept header when creating the response payload.
 - b. The `sapIdocInboundErrorChannel` propagates error messages to the corresponding error handler.
3. The request forwarded by `sapIdocInboundAdapter` is received by `sapIdocToStringTransformer`, which is a Spring Integration **object-to-string-transformer**. Its output, a plain string, is propagated via the channel `sapIdocInboundStringRequestChannel`.
4. The subsequent processing depends on the used PPS version:
 - Until PPS 3.0:
 - This channel is directly connected to a Spring Integration **unmarshalling-transformer** (`sapIdocUnmarshalTransformer`) receiving this string. This is a wrapper delegating the actual unmarshalling of the string to a more structured Java class to the `sapIdocUnmarshaller` that is an ordinary Spring bean.
 - This Spring bean of type `XmlToMapUnmarshaller` uses Jackson from FasterXML to convert the string into a Java Map. Each map entry represents one element of the XML payload. In addition, it also supports unwrapped lists in the XML document, for example, payloads in which one XML element is contained on the same level several times together with other XML elements. With this approach, it is not necessary to provide Java classes (usually created by XSD via XJC) for each IDoc type to be processed.
 - As of PPS 4.0:
 - This channel is connected to the router `sapContentTypeRouter`. This determines the receiving channel based on the content type of the incoming request.
 - For application/json, the request is forwarded via channel `sapIdocInboundStringRequestChannelJson` to the unmarshalling-transformer `sapIdocUnmarshalTransformerJson` which delegates the actual work to the bean `sapIdocUnmarshallerJson`.
 - **Otherwise, the request is forwarded via channel `sapIdocInboundStringRequestChannelXml` to the existing unmarshalling-transformer `sapIdocUnmarshalTransformer`.**
 - In both cases, the unmarshalled content is a `LinkedHashMap`.
5. The resulting map is propagated via the channel `sapIdocInboundUnmarshalledChannel` to the next stage.
6. This stage is a Spring Integration **router** looking at the IDoc type that is stored in the IDoc control header. Based on the content of this field, the name of the channel that forwards the message to the next stage is determined dynamically. The name of the channel follows the schema is `sapIdocTypeChannel_{idocType}`.
7. The following three channels exist in the standard implementation:
 - a. `sapIdocTypeChannel_/ROP/PROMOTION01` and `sapIdocTypeChannel_/ROP/PROMOTION02` for IDoc type `/ROP/PROMOTION01` and `/ROP/PROMOTION02`. This is connected to the Spring **service-activator** `sapPromotionInboundServiceActivator`, delegating the actual work to an ordinary Spring bean with the name `sapPromotionInboundProcessor`.
 - b. `sapIdocTypeChannel_/ROP/BASE_SALES_PRICE01` for IDoc type `/ROP/BASE_PRICE01`. This is connected to the Spring **service-activator** `sapBasePricelInboundServiceActivator`, delegating the actual work to an ordinary Spring bean with the name `sapBasePricelInboundProcessor`.
8. If an exception is thrown during the inbound processing, this is automatically wrapped into a message forwarded via the channel `sapIdocInboundErrorChannel` to the Spring Integration transformer `sapIdocErrorTransformer`.
9. The spring integration transformer delegates the actual work to `sapIdocInboundExceptionTransformer`, an ordinary Spring bean. This Spring bean implements the following logic:
 - a. If the error message refers to an exception of type `IllegalIdocContentException` issued during the mapping from the IDoc to the database format, the HTTP response must have the error code 400 (*Bad Request*).
 - b. In the case of an `OptimisticLockException` during posting of the received data, the HTTP response must have the error code 409 (*Conflict*).
 - c. In the case of another exception, the HTTP response must have the error code 500 (*Internal Server Error*).
 - d. In any other cases, the HTTP response will have the error code 200 (*OK*), which is the default return code of the HTTP inbound adapter (actually not part of error handling).
10. The error response is sent via the channel `sapIdocInboundTransformedErrorChannel` to the Spring Integration **object-to-string-transformers** `sapIdocErrorToStringTransformer`. This Spring Integration converts it into a string that is returned to the caller.

Processing the IDoc Data

After the xml file has been converted into a Java `Map<String, Object>`, the converted IDoc content can be mapped to the corresponding entities. This happens in an Inbound-Processor that is implemented by the class `BasePricelInboundProcessorImpl` or `PromolInboundProcessorImpl` respectively. The inbound processor does the following:

- It calls the corresponding mapper to the JPA entities.
- It triggers the posting to the database.

Conversion of the IDoc Payload to the Expected Java Types

As the payload in the received IDocs is converted into a Java `Map<String, Object>`, you need to convert each field of the content in the Java types that are expected by the JPA entities. The class `EntityTypeConversionHelper` is provided for this purpose.

The following Java types are supported by default:

- String
- BigDecimal
- long
- Timestamp
- int
- Character
- byte

In addition, the following data is supported and requires special handling:

- Unit Of Measure Code: Expected ISO code is verified to determine whether it can be mapped to database format.
- Currency Code: Expected ISO code is verified to determine whether it can be mapped to database format.
- Language Code: Expected ISO code is verified to determine whether it can be mapped to database format.
- Business Unit Type Code: Expected SAP code (corresponding to the DDF location type code) is converted to the corresponding ARTS format.
- Boolean: Expected ABAP format ('X' or '') is converted into a Java Boolean.
- Product ID: Expected database format is converted into the format of the JPA entity. This may be different for numeric product IDs. For more information, see the documentation for the module **dataaccess-common**.

Regardless of the target JPA entity type, the mapping follows two strategies:

- For known JPA entities the received IDoc segments are only processed for the fields of the target entity. This means that if the received IDoc segment contains unexpected additional fields, they are simply ignored.
- In addition, the content of extension segments (name/value pairs) are mapped to the extension maps of the target JPA entity if the JPA metamodel contains a field with the corresponding name. If the field is not known in the JPA metamodel, it is ignored.

Mapping Regular Prices

Regular prices are mapped by the class **BasePriceMapperImpl**, which implements the interface **BasePriceMapper**. The regular price entity that needs to be mapped to is **BasePriceImpl**, which implements the interface **BasePrice**. For more information about the regular price entity, see the documentation for the modules **ppengine-dataaccess-common** and **ppengine-dataaccess-interface**.

Mapping OPP Promotions



As of PPS version 2.0, the PPS version is checked against the current PPS version (reflected by the promotion IDoc content-attribute **MIN_PPS_RELEASE**).

The current PPS must be able to process the corresponding promotion. This is assumed to be the case if the PPS version of the promotion is lower than or equal to the version of the local PPS. If this precondition is not fulfilled, the IDoc is rejected. The current PPS version is stored as a configuration property (**sap.idocinbound.currentppsrelease**).

OPP promotions are mapped by the class **PromotionImpl** (that implements the interface **Promotion**) and its subclasses that represent all the entities needed to replicate the runtime model in SAP Customer Activity Repository.

For more information about promotion entities, see the documentation for the modules **ppengine-dataaccess-common** and **ppengine-dataaccess-interface**.



OPP promotions marked as *obsolete* that are not relevant anymore for the receiver of the IDoc, for example, are treated as OPP promotions that have been logically deleted on the sender side.

Validating Uploaded Data

Until PPS 3.0, there is a basic validation of the uploaded data which is set by default. As of PPS 4.0, a new plugin interface **com.sap.ppengine.api.plugin.Validation** is called for the validation of uploaded data. Depending on the setting of the configuration property **sap.core.usebeanvalidation**, you can replace the basic default validation with the more extensive Bean Validation based validation. The structure of the validation result is defined by the class **com.sap.ppengine.idocinbound.common.DataUploadResponse**. It can be represented in XML and JSON (as of PPS 4.0) format.



For regular prices the validation is called several times for separate validation groups (covering different aspects), following the hierarchical structure of the IDoc type **/ROP/BASE_PRICE01**.



The validation via **com.sap.ppengine.api.plugin.Validation** consumes considerably more resources. The processing may take longer and needs more memory.

Posting to the Database

After the mapping process, the JPA entities need to be posted to the database.

Regular Prices



The inbound processing of regular prices relies on the constraint that for a given transfer session all prices for a given combination of product ID, unit of measure code, price classification, business unit type, and logical system are in the same IDoc. If not, this can result in inconsistent data.

The posting of regular prices must ensure that no overlapping prices exist. Since information about price deletions is not transferred, it has to be done on the receiver side as follows:

- For the corresponding list of business units within a top-level IDoc price segment, the existing prices are read for each product, uom, price classification, business unit type, and logical system. This is done for all prices with an effective date that is at least the earliest effective date of the transferred prices.
- These prices are compared to the mapping result as follows:
 - Prices not yet existing are inserted in the database
 - Prices that existed before are updated in the database if at least one attribute has been changed
 - Prices that exist in the database that are not part of the received IDoc are removed



A top level IDoc segment that contains a lot of business unit IDs reading the existing prices can lead to oversized SQL statements. The size of these statements can be controlled by the configuration parameter `sap.dataaccess-common.partitionSizeSqlInStatement` that controls the number of business unit IDs that can be part of one SQL statement. If the total number of business units exceeds this limit, the system automatically reads the data in smaller chunks.

For more information about this configuration parameter, see the documentation for the module `dataaccess-commonin`.

The data access for regular sales prices is delegated to the bean `sapBasePriceService`.

OPP Promotions

The data access for OPP promotions is delegated to the bean `sapPromotionService`.

Posting of the OPP promotions is either a *merge* or a *persist* from a JPA perspective point of view. To optimize performance, a merge is executed only if there is already a version for the corresponding promotion ID. Otherwise a persist is done. Physical deletion of an OPP promotion is not done during IDoc inbound processing.

Beans

ID	Alias	Description
sapIdocInboundGateway	./.	Spring Integration HTTP inbound gateway receiving IDocs and sending confirmation responses
sapIdocErrorTransformer		Spring Integration Transformer transforming exceptions created during request processing into HTTP responses. Delegates work to the bean <code>sapIdocInboundExceptionConverter</code> and sends the result to <code>sapIdocErrorToStringTransformer</code> .
sapDefaultIdocInboundExceptionConverter	sapIdocInboundExceptionConverter	Actual implementation of the exception conversion. Hides the stack trace from the response and sets the HTTP response code depending on the exception type
sapIdocErrorToStringTransformer		Spring Integration Transformer creating a string representation of the converted error response
sapIdocInboundRequestChannel		Spring Integration Channel transporting the originally received payload of the HTTP request
sapDefaultRequestHeaderFieldsSaver	sapRequestHeaderFieldsSaver	As of PPS 4.0. Bean used by a Spring Integration Interceptor storing the HTTP request headers in the PPS context.
sapIdocInboundErrorChannel		Spring Integration Channel connecting <code>sapIdocInboundGateway</code> and <code>sapIdocErrorTransformer</code>
sapIdocInboundTransformedErrorChannel		Spring Integration Channel connecting <code>sapIdocErrorTransformer</code> and <code>sapIdocErrorToStringTransformer</code>
sapIdocToStringTransformer		Spring Integration Transformer creating a string representation of the received IDoc body payload
sapIdocInboundStringRequestChannel		Spring Integration Channel connecting <code>sapIdocToStringTransformer</code> and <code>sapIdocUnmarshalTransformer</code> (until PPS 3.0) or <code>sapContentTypeRouter</code> (as of PPS 4.0).
sapContentTypeRouter		As of PPS 4.0. Spring Integration Router forwarding the request depending on the content type either to channel <code>sapIdocInboundStringRequestChannelXml</code> or <code>sapIdocInboundStringRequestChannelJson</code> .
sapIdocInboundStringRequestChannelXml		As of PPS 4.0. Spring Integration Channel connecting <code>sapContentTypeRouter</code> and <code>sapIdocUnmarshalTransformer</code> .
sapIdocInboundStringRequestChannelJson		As of PPS 4.0. Spring Integration Channel connecting <code>sapContentTypeRouter</code> and <code>sapIdocUnmarshalTransformer</code> .

hannelJson		
sapIdocUnmarshallerTransformer		Spring Integration Unmarshalling Transformer transforming the string payload into a format consumable by the application logic. Delegates work to sapIdocUnmarshaller . For XML requests only.
sapIdocUnmarshallerTransformerJson		As of PPS 4.0. Spring Integration Unmarshalling Transformer transforming the string payload into a format consumable by the application logic. Delegates work to sapIdocUnmarshaller . For JSON requests only.
sapDefaultIdocUnmarshaller	sapIdocUnmarshaller	Unmarshaller using Jackson to create a generic representation of the IDoc payload as a Map<String, Object>. For XML request only.
sapDefaultIdocUnmarshallerJson	sapIdocUnmarshallerJson	As of PPS 4.0. Unmarshaller using Jackson to create a generic representation of the IDoc payload as a Map<String, Object>. For JSON request only.
sapIdocInboundUnmarshalledChannel		Spring Integration Channel connecting sapIdocUnmarshallerTransformer and sapIdocTypeRouter
sapIdocTypeRouter		Spring Integration Router looking at the IDoc type as stored in the IDoc control header to decide to which channel the message shall be forwarded. Channel name is defined as sapIdocTypeChannel_<idocType> .
sapIdocTypeChannel_/_/ROP/_/BASE_PRICE01		Spring Integration Channel connecting sapIdocTypeRouter and sapBasePriceInboundServiceActivator . Intended for IDoc type /ROP/BASE_PRICE01 .
sapIdocTypeChannel_/_/ROP/_/PROMOTION01		Spring Integration Channel connecting sapIdocTypeRouter and sapPromotionInboundServiceActivator . Intended for IDoc type /ROP/PROMOTION01 .
sapIdocTypeChannel_/_/ROP/_/PROMOTION02		Spring Integration Channel connecting sapIdocTypeRouter and sapPromotionInboundServiceActivator . Intended for IDoc type /ROP/PROMOTION02 .
sapBasePriceInboundServiceActivator		Spring Integration Service Activator receiving representation of a regular price IDoc, delegating work to sapDefaultBasePriceInboundProcessor
sapPromotionInboundServiceActivator		Spring Integration Service Activator receiving representation of an OPP promotion IDoc with type /ROP/PROMOTION01 , delegating work to sapDefaultPromotionInboundProcessor
sapPromotion02InboundServiceActivator		As of PPS 3.0. Same as sapPromotionInboundServiceActivator , receiving type /ROP/PROMOTION02 .
sapDefaultEntityTypeConversionHelper	sapEntityTypeConversionHelper	Helper to read information from the Map<String, Object> representation of an IDoc and returning it in the expected java type
sapDefaultEntityPromoMapper	sapEntityPromoMapper	Helper to map the complete content of an OPP promotion IDoc representation as Map<String, Object> into the corresponding JPA entities
sapDefaultEntityBasePriceMapper	sapEntityBasePriceMapper	Helper to map the complete content regular price IDoc representation as Map<String, Object> into the corresponding JPA entities
sapDefaultExtensionMapper	sapExtensionMapper	Generic mapper of the extension segments of the OPP promotion IDoc to the corresponding attributes of the target JPA entities
sapDefaultPromotionInboundProcessor	sapPromotionInboundProcessor	Main entry point into the application logic for inbound processing of OPP promotion IDocs. Delegates work to mapping helper and updates the database.
sapDefaultBasePriceInboundProcessor	sapBasePriceInboundProcessor	Main entry point into the application logic for inbound processing of regular price IDocs. Delegates work to mapping helper and updates the database.
sapDefaultIdocInboundCommon	sapIdocInboundCommon	Parent bean for all IDoc inbound related functions, holding commonly used dependencies.
sapInboundPersistenceAnnotationBeanPostProcessor	sapInboundAnnotationBeanPostProcessor	Bean post processor that enables the support of the @Persistence annotation for a threadsafe EntityManager . This is required for the generic mapping of IDoc extension segments. Note that a Spring Bean of the same type also exists in the dataaccess-common module . However, in a deployment with Hybris, the PPS application context is created in 2 steps and the postprocessor in the dataaccess-common module is no longer considered when creating the second level of the application context (containing the idocinbound module).

sapDefaultFKRemovalExecutor		Bean that automatically executes the native query for foreign key removal as offered via bean sapForeignKeyRemover . The query is executed during the initialization of the PPS application context (as part of the PPS application context in which the idocinbound module is located).
sapDefaultIdocInboundResponseHelper	sapIdocInboundResponseHelper	As of PPS 4.0. Helper to fill the response structure com.sap.ppengine.idocinbound.common.DataUploadResponse with the processing result for an uploaded IDoc.

Required Beans

The following table contains the additional beans to be provided if all dependencies of this module are resolved:

ID / Alias	Comment
sapDataSource	Provides the database access

Configuration Properties

The following properties are used by this module:



Configuration properties defined in other modules with dependencies on this module may **not** be used. Because locally deployed in SAP Hybris Commerce, this module is loaded at a later date when other configuration properties are not visible anymore.

Name	Description	Default Value	Comment
sap.idocinbound.currentppsrelease	Reference to the current version of the PPS	Depends on the current PPS version	<p>This property refers to the current PPS version as follows:</p> <ul style="list-style-type: none"> ▪ The first digits of the decimal representation indicate the major version of the PPS version. ▪ The next 3 digits of the property indicate the minor version of the PPS version. ▪ The lowest 3 digits of the property indicate the patch level of the PPS version. <p>In PPS versions earlier than 2.0, this property is set to 0.</p> <p>For example, the property is set to 2000000 in PPS version 2.0 and it is set to 0 in PPS 1.1 and PPS 1.2.</p>

Dependencies

This module depends on the following modules:

- dataaccess-localdb
- jackson

PPS Performance Hints

The following chapter gives hints on how to achieve optimal performance using the promotion pricing service.

Creating of the Offers

- **For PPS version 1.0 and 2.0: Keep the offers small.** During the price calculation, an OPP promotion is validated for consistency. This is needed because of cache eviction. The time needed for this grows with the number of OPP promotions (not considering the assigned business units/locations assigned to the offer version). An offer with thousands of assigned articles may lead to memory and runtime issues. Try to split one large offer into several smaller ones.
- **For PPS version 3.0 and higher: Do not keep the offers too small.** Compared to older PPS versions, the consistency check for the entire OPP promotion is not required for the price calculation. Therefore, OPP promotions with many promotional rules (mapped from offers with many offer terms) do not impact the performance of the price calculation. However, it is still recommended to make a trade-off for the size of the offer:
 - If you maintain many small offers, the size of the offer is dominated by the list of assigned business units. Having the same list of business units redundantly assigned to many promotions blows up the database and increases the resource consumption during the replication of OPP promotions.
 - If you maintain only a few amount of large offers, the probability increases that these offers have to be updated and resent regularly. In this situation, the replication of a small promotion would be better.
- **Consider restricting the set of product dimension types used.** Each of the product dimensions used within a DDF offer correlates to a certain eligibility type within an OPP promotion:
 - "Product" dimension translates to eligibility type "Item"

- "Product Hierarchy Node" dimension translates to eligibility type "Merchandise Category"
- "Product Group" dimension translates to eligibility type "Merchandise Set"

Each of these eligibility types must be processed, leading to database calls and entries in caches, increased response time, and memory consumption. In particular, item and merchandise set eligibilities have a big influence on performance. Since merchandise set eligibilities offer superior flexibility compared to item eligibilities and merchandise category eligibilities, it might be an option to always maintain offers for product groups. How you deactivate the processing of certain eligibilities is described in the SDK for the promotion calculation engine.

Distributing of the Data

- **Restrict the filter criteria for the data as much as possible.** This increases the performance of reading the data during outbound processing and avoids the expensive replication of data not needed on the receiver side.
- **Consider the usage of the filter "Lead time in days".** This delays the transfer of an active OPP promotion so that changes to it before it actually becomes effective do not need to be transferred again. This reduces the amount of transferred data.


Client Side (Price Calculation)

- **Consider keeping regular prices that were calculated before and provide them with subsequent requests.** Note that this has consequences for the overall behavior - a regular price of a product in a basket would not change any more. Whether this is desired or not is a business decision.
- **Provide only product hierarchy nodes on which it is possible to define promotions within your company.** The PPS has to search for eligibilities for each product hierarchy node provided.
- **Accept and send cookies of the central PPS.** These cookies hold authentication-related information. If the received cookies are not sent back to the PPS, each request requires a complete authentication.
- **Consider compression of the request sent to the PPS.** Depending on the network and client CPU speed, this may lead to faster end-to-end times. In the case of small shopping carts, the effect of this is limited.
- **Consider deactivation of the response compression.** The PPS response is compressed by default. In the case of very fast network connections, it may be faster to deactivate the response compression.



If an ABAP system is the PPS client, the last three settings can be configured for the corresponding RFC destination:

RFC Destination [Redacted]

Connection Test 

RFC Destination [Redacted]

Connection Type G HTTP Connection to External Serv Description

Description

Description 1	[Redacted]
Description 2	[Redacted]
Description 3	

Administration | Technical Settings | Logon & Security | **Special Options**

Timeout

ICM Default Timeout
 No Timeout
 Specify Timeout Timeout in Seconds (1 to 9999999)

HTTP Setting

Status of HTTP Version

HTTP Version HTTP 1.0 HTTP 1.1

Compression Status

Compression Inactive
 Active (Depends on MIME Type)
 Active (Whole Document)

Status of Compressed Response

Compressed Response Yes No

HTTP Cookies

Type of Cookies Acceptance

Accept Cookies No
 Yes (All)
 Input Prompt
 Trigger Event

Client Side (Data Replication)

- **Do not flood the PPS.** When you replicate regular prices using parallel processing, the prices may be sent faster than they can be processed on the receiver side. When you send large volumes of data this may lead to congestion of the Web server and connection timeouts. Try to find a balance between the sender and receiver by setting the right number of parallel processes for outbound processing.

Server Side

Common Rules

- **Set the cache for named queries as large as possible.** This applies for reading regular prices as well as for finding eligibilities. The more query results that can be cached, the less load will be put on the database.
- **Keep an eye on the ratio of the cache sizes.** If you have to save memory and cannot set the cache sizes very high, it is important to have a realistic ratio of the sizes for regular price and OPP promotion reference cache. In most real-world scenarios, the ratio between promotion references and regular prices is about 2:1. This should therefore also be the ratio of the cache sizes (cf. `sap.dataaccess-common.promocachespec` and `sap.dataaccess-common.basepricecachespec`).
- **Set the time to live of named query results as long as possible.** The longer a query result may stay in the cache, the less often it has to be read again from the database. On the downside, emergency updates (due to wrong prices, for example) will take longer to become effective since they will be seen only after the time to live within the cache has expired or if the information was evicted from the cache due to memory shortage.
- **When using your own database connection pool, make sure the pool size is large enough.** To be on the safe side, set the pool to the same maximum number of threads that may be used by the Web application.
- **Set the log level accordingly.** The log levels "debug" or "trace" should be used only in exceptional cases if something does not work as expected.
- **Consider using the bulk access to read regular prices or eligibilities.** Using PPS version 2.0 or higher can mean a significant performance improvement because the bulk access also considers the cache. With PPS version 3.0 or higher, the number of searches supporting a bulk access has increased:
 - PPS 2.0 or higher: Regular sales prices. Controlled via configuration parameter `sap.client-impl.basepricebulkaccessitemthreshold`
 - PPS 3.0 or higher: Item eligibilities. Controlled via configuration parameter `sap.dataaccess-common.bulkitemelithreshold`
 - PPS 3.0 or higher: Merchandise set eligibilities. Controlled via configuration parameter `sap.dataaccess-common.bulkmerchsetelithreshold`
 - PPS 3.0.14 or higher: Merchandise hierarchy node eligibilities. Controlled via configuration parameter `sap.dataaccess-common.bulkmerchgroupelithreshold`
- **Remove obsolete promotions and regular prices from the database on a regular basis.** This keeps the database access times low and reduces TCO.
- **Consider to change the default algorithm for best price determination** (as of PPS 4.0). If several promotional rules have the same sequence and resolution, the PPS tries to determine the combination of rules by giving the best reward for the customer. By default the PPS uses a brute-force attempt for this calculation. Alternatively, it can use an algorithm which usually takes less time and delivers better results. This algorithm can be enabled by setting the property `conflictHandlerStrategy` to **Greedy** instead of the default **BruteForce**.
- As of PPS 3.0: For a given combination of business unit and date: **Keep the number of promotional rules low that have a certain coupon, manual eligibility or customer group eligibility.** Usually, these eligibilities are used in conjunction with item-related eligibilities (such as item, merchandise group or merchandise set eligibilities), linked with an "&&" combination. For example, "*Show coupon XYZ and get x% discount on articles*". Instead of creating separate rules for each item or merchandise category, it is better to define one rule referring to a merchandise set containing all relevant items. In this case, eligibilities are searched separately and afterwards linked by the application. The following example shows the behavior depending on the definition of promotional rules. It has no effect if these rules are defined in one promotion or in separate promotions. Assume you have a shopping cart with item X-10 and coupon ABC:
 - 100 rules: "Get 10% on item X-i if you show coupon ABC", $i=1..100$
 - When searching for eligibilities for coupon code ABC, 100 are found.
 - The corresponding promotion master data are read (partially), including the combination eligibilities and so on.
 - For item X-10, the item eligibility is read.
 - The eligibility trees **for all 100 coupon eligibilities are checked.** Only one of these trees is relevant (with eligibility for coupon ABC and item X-10. Only for this combination the loaded promotion is applied.
 - 99 eligibilities and promotional rules were processed but not needed.
 - 1 rule: "Get 10% on an item contained in merchandise group X-1 to X-100 if you show coupon ABC"
 - When searching for eligibilities for coupon code ABC, 1 is found. It is not required to load the definition of the merchandise set.
 - The corresponding promotion master data is read (partially), including the combination eligibilities and so on.
 - For item X-10, the item eligibility is read.
 - The eligibility tree **for one coupon eligibility is checked.** The system finds out that the corresponding rule may be applied.
 - No promotion was read without being needed.
- As of PPS 4.0: **Consider setting the initial capacity of the caches for regular prices and eligibility search results.** By default, the initial capacity is very small and the cache sizes are enlarged depending on the incoming requests. However, if the PPS receives requests requiring a lot of cache updates within each request, it may run into write contentions if the cache size is not large enough. This can be avoided by setting the initial capacity "large enough" (for instance, larger than the number of threads processing the requests). For more information, see the GitHub documentation under <https://github.com/ben-manes/caffeine/wiki/Faq>. Setting the initial capacity is done as follows:
 - For regular prices (setting capacity to 100): `sap.dataaccess-common.basepricecachespec=initialCapacity=100,maximumSize=10000,expireAfterAccess=10m,expireAfterWrite=20m`
 - For eligibility search results (setting capacity to 100): `sap.dataaccess-common.promocachespec=initialCapacity=100,maximumSize=10000,expireAfterAccess=10m,expireAfterWrite=20m`

Local-PPS-Specific

- **Set the target database platform in the JPA parameters.** This enables the use of optimized SQL statements.

XSA-Based-PPS-Specific

- **Scale the application router.** Unlike the Web application of the PPS, which maintains a thread pool internally, the application router always runs in a single thread. This may become a bottleneck if the load is increased. Therefore, use the `xs scale` command to provide enough instances of the application. As a starting point, choose 1 application router instance per 10 tomcat threads.

- **Log failed login attempts only.** As of with XSA version 1.0.88, it is possible to configure the audit log to create log entries for failed login attempts only. This considerably reduces the amount of entries and improves performance under high load. Enabling the audit log to consider only failed attempts is done in the MTAEXT file as follows:

```

- name: ppservice-webapp-central
  parameters:
    memory: 4096M
  properties:
    DISABLE_SUCCESSFUL_LOGIN_AUDIT_LOG: "true"

```

Database Side


- **Check the database indexes regularly.** Proper indexes are crucial for fast access times. Since the choice of indexes heavily depends on the database platform and the content of the database tables, it is not possible to give precise recommendation here.
 - Example: When you use product groups (available with PPS 3.0) with a huge amount of product group entries, it is helpful to create indexes in table **SAPPSMERCH_SET** for the item or the product hierarchy node identifiers.
- **Set the right configuration parameters.** This may sound obvious but is often overlooked. Again, this depends greatly on the database platform used. Just as an example: if you are using MySQL, setting the option **rewriteBatchedStatements=true** will have a large impact on IDoc inbound processing.


PPS Logging and Tracing

PPS uses SLF4J for logging and tracing. SLF4J provides a facade for writing log messages making the application independent from the actual logging framework. The logging implementation behind SLF4J should be a common choice of the runtime environment, for example, within SAP Hybris Commerce Log4J2 is used. In addition to the SAP created artifacts, PPS makes use of a variety of open source components relying on different logging frameworks.

To enable consistent logging, you have to do the following:

- Spring and the promotion calculation engine (cf. module calcengine-gk) rely on Jakarta Commons Logging (<https://commons.apache.org/proper/commons-logging/>). To enable logging via the implementation of SLF4J, `jcl-over-slf4j` is used.
- Google Guava relies on the logger provided via the Java JDK. To enable logging via the implementation of SLF4J, `jul-over-slf4j` is used.
- EclipseLink comes with its own logging. To enable logging via the implementation of SLF4J, **class `com.sap.ppengine.dataaccess.common.util.impl.Slf4jSessionLogger`** is used to redirect the output to SLF4J.

 When writing own log messages, use SLF4J as well. This ensures a fast logging and consistent configuration, also for future deployment options where a different logging implementation might be used.

 Due to the delegation of commons-logging and the native Java logging API to SLF4J, these frameworks cannot be used as logging implementation, since this would result in an infinite loop.

Further information about logging bridges can be found here: <http://www.slf4j.org/legacy.html>

PPS Authentication

Our application is authenticated in SAP HANA XS Advanced Model (XSA) using the application router. There are two methods of authentication using an application router:

- **OAUTH2** authentication
- **BASIC** authentication

The setup of both authentication methods is the same.

OAUTH2 authentication is always active. If you want to use **BASIC** authentication, you can activate this method additionally.

For more information about the application router, see the SAP HANA Developer Guide for SAP HANA XS Advanced Model > Chapter 9: Maintaining XS Advanced Application Routes and Destinations.

For more information about the XSA security concept, see SAP HANA Developer Guide for SAP HANA XS Advanced Model > Chapter 10: Setting Up Security Artifacts.

Enabling XSA Authentication

This section describes how to enable XSA authentication in a Web application. It is assumed that you are using Maven as your build tool. If you use another build tool, you have to adjust the corresponding steps accordingly.

1. The **web.xml** file of your Web application must define a `<login-config>` and a `<security-constraint>` that contains the scope `Calculate`. This scope is needed to use the service and the URL patterns that are to be protected.

```
web.xml


<login-config>
  <auth-method>XSUAA</auth-method>
</login-config>
<security-constraint>
  <display-name>SecurityConstraint</display-name>
  <web-resource-collection>
    <web-resource-name>WRCollection</web-resource-name>
    <url-pattern>/restapi/*</url-pattern>
    <url-pattern>/restapi</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>Calculate</role-name>
  </auth-constraint>
</security-constraint>
```

2. XSA applications are expected to have a Web part and a back-end part. The Web part contains all the UI stuff (static content) and also the authentication and redirection task. The back-end part contains the business logic. Authentication and redirection is executed with the application router, which is an XSA feature. As the PPS does not need a UI, our Web part consists only of the application router part.

- The application router function is configured in a file called `package.json`. This file defines the start script and the version of the application router. It is located in Maven module `ppservice-approuter` (Web folder).

```
package.json

{
  "name": "ppengine-approuter",
  "dependencies": {
    "approuter": "2.3.0"
  },
  "scripts": {
    "start": "node node_modules/approuter/approuter.js"
  }
}
```

 You need at least version 1.6.3 to configure the application router function.

- To redirect incoming requests correctly, our application router needs routes to be defined in the file `xs-app.json`. This file is also located in Maven module `ppservice-approuter` (Web folder).

```
xs-app.json

{
  "routes": [
    {
      "source": "/restapi",
      "destination": "java",
      "authenticationType": "basic",
      "csrfProtection": false,
      "scope": "$XSAPPNAME.Calculate"
    },
    {
      "source": "^/(.*)",
      "localDir": "resources"
    }
  ]
}
```

In this example, the route to our **restapi** is the most important. The name of the *destination* is **java**. It needs to be aligned with the corresponding destination in the manifest file. The **authenticationType** is set to **basic**. With this parameter you can, for example, specify that basic authentication should also be supported). *csrfProtection* is disabled and the *scope* for our webapp is set.

- Since the PPS does not need a UI, the **index.html** file is just an empty HTML page. The **index.html** is called after a successful login (only in the case of **OAUTH** authentication). This file is also located in Maven module **ppservice-approuter** (*web/resources* folder).

index.html

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>ppservice-approuter</title>
  </head>
  <body>
  </body>
</html>
```

3. To deploy your application router and your Web application, you need to create the following files:

assembly.xml

```
<!-- Artifact: assembly @Copyright (c) 2016, SAP SE, Germany, All rights
reserved. -->
<assembly
  xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.3
http://maven.apache.org/xsd/assembly-1.1.3.xsd">
  <id>mta</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <files>
    <file>
      <source>./mtad.yaml</source>
      <outputDirectory>META-INF</outputDirectory>
    </file>
    <file>
      <source>./xs-security.json</source>
      <outputDirectory>.</outputDirectory>
    </file>
  </files>
  <fileSets>
    <fileSet>
      <directory>../ppservice-approuter/web</directory>
      <outputDirectory>web</outputDirectory>
      <excludes>
        <exclude>pom.xml</exclude>
      </excludes>
    </fileSet>
  </fileSets>
  <dependencySets>
    <dependencySet>
      <includes>
        <include>com.sap.retail.ppservice:ppservice-webapp-central</include>
      </includes>
      <outputDirectory>.</outputDirectory>
      <outputFileNameMapping>ppservice-webapp-central.war</outputFileNameMapping>
    </dependencySet>
    <dependencySet>
      <includes>
        <include>*:sources</include>
      </includes>
      <outputDirectory>src</outputDirectory>
```



```

        </dependencySet>
        <dependencySet>
            <includes>
                <include>*:javadoc</include>
            </includes>
            <outputDirectory>javadoc</outputDirectory>
        </dependencySet>
    </dependencySets>
</assembly>

```

The **assembly.xml** file contains the linking of the different files that are needed for the deployment and the dependency to the webapp.

mtad.yaml

```

_schema-version: "2.0.0"
ID: com.sap.retail.ppservice.XSAC_OPP_PPS
version: 1.0.0
modules:
  - name: ppservice-approuter
    type: javascript.nodejs
    path: ./web
    requires:
      - name: ppServiceUaa
      - name: java
        group: destinations
        properties:
          name: java
          url: ~{url}
          forwardAuthToken: true

  - name: ppservice-webapp-central
    type: java.tomcat
    path: ppservice-webapp-central.war
    properties:
      JBP_CONFIG_RESOURCE_CONFIGURATION:
      JBP_CONFIG_JAVA_OPTS:
    provides:
      - name: java
        properties:
          url: "${default-url}"
    requires:
      - name: ppeHana
      - name: ppServiceUaa

resources:
  - name: ppeHana
    type: org.cloudfoundry.user-provided-service

  - name: ppServiceUaa
    type: com.sap.xs.uaa-space
    parameters:
      config_path: xs-security.json

```

The **mtad.yaml** file contains both modules (approuter and webapp) and the resources (only services).

SL_MANIFEST.xml

```

<!--
Artifact: SL_MANIFEST
@Copyright (c) 2016, SAP SE, Germany, All rights reserved.
-->
<software-component-version formatVersion="1.0" schemaVersion="1.0">
  <software-component-version-key>
    <PPMS-ID>73554900100200005395</PPMS-ID>
    <name>XSAC_OPP_PPS</name>          <!--change also in mtad.yaml-->
    <version>1</version>
    <vendor>sap.com</vendor>

```

```

</software-component-version-key>
<caption>XSAC_OPP_PPS 1</caption>
<sp>
  <sp-key>
    <name>SP000</name>
    <sp-level>000</sp-level>
    <vendor>sap.com</vendor>
  </sp-key>
  <patch-level>0</patch-level>
  <sp-caption>SP000 for XSAC_OPP_PPS 1</sp-caption>
</sp>
<runtime-type>XSART</runtime-type>
</software-component-version>

```

The **SL_MANIFEST.xml** file contains only some naming and version information.

sap-xsac-opp-pps pom.xml

```

<?xml version="1.0"?>
<!-- Artifact: pom @Copyright (c) 2016, SAP SE, Germany, All rights reserved. -->
<project
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>xsac-pps-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>sap-xsac-opp-pps</artifactId>
  <name>sap-xsac-opp-pps</name>
  <packaging>pom</packaging>
  <url>http://sap.com</url>
  <dependencies>
    <dependency>
      <groupId>com.sap.retail.ppservice</groupId>
      <artifactId>ppservice-webapp-central</artifactId>
      <version>${project.version}</version>
      <type>war</type>
    </dependency>
    <dependency>
      <groupId>com.sap.retail.ppservice</groupId>
      <artifactId>ppservice-approuter</artifactId>
      <version>${project.version}</version>
      <type>pom</type>
    </dependency>
    <!-- Set dependency to Source JARs. Unfortunately they seem to be not transitive -->
    <dependency>
      <groupId>com.sap.retail.ppservice</groupId>
      <artifactId>ppengine-core</artifactId>
      <version>${version.pps}</version>
      <classifier>sources</classifier>
    </dependency>
    <dependency>
      <groupId>com.sap.retail.ppservice</groupId>
      <artifactId>ppengine-client-impl</artifactId>
      <version>${version.pps}</version>
      <classifier>sources</classifier>
    </dependency>
    <dependency>
      <groupId>com.sap.retail.ppservice</groupId>
      <artifactId>ppengine-dataaccess-interface</artifactId>
      <version>${version.pps}</version>
      <classifier>sources</classifier>
    </dependency>
    <dependency>
      <groupId>com.sap.retail.ppservice</groupId>
      <artifactId>ppengine-dataaccess-common</artifactId>
      <version>${version.pps}</version>
    </dependency>
  </dependencies>

```

```

        <classifier>sources</classifier>
</dependency>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>ppengine-jackson</artifactId>
    <version>${version.pps}</version>
    <classifier>sources</classifier>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>ppengine-restapi</artifactId>
    <version>${version.pps}</version>
    <classifier>sources</classifier>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>ppengine-client-interface</artifactId>
    <version>${version.pps}</version>
    <classifier>sources</classifier>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>pricing-engine-psi-sap</artifactId>
    <version>${version.pps}</version>
    <classifier>sources</classifier>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>pricing-engine-core</artifactId>
    <version>${version.pps}</version>
    <classifier>sources</classifier>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>pricing-engine-dataaccess-sap</artifactId>
    <version>${version.pps}</version>
    <classifier>sources</classifier>
</dependency>

<!-- Set dependency to JavaDoc JARs. Unfortunately they seem to be not transitive -->
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>ppengine-core</artifactId>
    <version>${version.pps}</version>
    <classifier>javadoc</classifier>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>ppengine-client-impl</artifactId>
    <version>${version.pps}</version>
    <classifier>javadoc</classifier>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>ppengine-dataaccess-interface</artifactId>
    <version>${version.pps}</version>
    <classifier>javadoc</classifier>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>ppengine-dataaccess-common</artifactId>
    <version>${version.pps}</version>
    <classifier>javadoc</classifier>
</dependency>
<dependency>
    <groupId>com.sap.retail.ppservice</groupId>
    <artifactId>ppengine-jackson</artifactId>
    <version>${version.pps}</version>
    <classifier>javadoc</classifier>
</dependency>
</dependency>

```

```

        <groupId>com.sap.retail.ppservice</groupId>
        <artifactId>ppengine-restapi</artifactId>
        <version>${version.pps}</version>
        <classifier>javadoc</classifier>
    </dependency>
    <dependency>
        <groupId>com.sap.retail.ppservice</groupId>
        <artifactId>ppengine-client-interface</artifactId>
        <version>${version.pps}</version>
        <classifier>javadoc</classifier>
    </dependency>
    <dependency>
        <groupId>com.sap.retail.ppservice</groupId>
        <artifactId>pricing-engine-psi-sap</artifactId>
        <version>${version.pps}</version>
        <classifier>javadoc</classifier>
    </dependency>
    <dependency>
        <groupId>com.sap.retail.ppservice</groupId>
        <artifactId>pricing-engine-core</artifactId>
        <version>${version.pps}</version>
        <classifier>javadoc</classifier>
    </dependency>
    <dependency>
        <groupId>com.sap.retail.ppservice</groupId>
        <artifactId>pricing-engine-dataaccess-sap</artifactId>
        <version>${version.pps}</version>
        <classifier>javadoc</classifier>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <configuration>
                <descriptors>
                    <descriptor>assembly.xml</descriptor>
                </descriptors>
            </configuration>
            <executions>
                <execution>
                    <id>assemble-mta-archive</id>
                    <phase>prepare-package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                    <configuration>
                        <archive>
                            <addMavenDescriptor>>false</addMavenDescriptor>
                            <manifest>
                                <addDefaultImplementationEntries>>false<
/ addDefaultImplementationEntries>
                                <addDefaultSpecificationEntries>>false<
/ addDefaultSpecificationEntries>
                            </manifest>
                            <manifestSections>
                                <manifestSection>
                                    <name>web</name>
                                    <manifestEntries>
                                        <Content-Type>text/directory</Content-Type>
                                        <MTA-Module>ppservice-approuter</MTA-Module>
                                    </manifestEntries>
                                </manifestSection>
                                <manifestSection>
                                    <name>ppservice-webapp-central.war</name>
                                    <manifestEntries>
                                        <Content-Type>application/zip</Content-Type>
                                        <MTA-Module>ppservice-webapp-central</MTA-Module>
                                    </manifestEntries>
                                </manifestSection>
                            </manifestSections>
                        </archive>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>

```

```

        <manifestSection>
            <name>xs-security.json</name>
            <manifestEntries>
                <Content-Type>application/json</Content-Type>
                <MTA-Resource>ppServiceUaa</MTA-Resource>
            </manifestEntries>
        </manifestSection>
        <manifestSection>
            <name>META-INF/mtad.yaml</name>
            <manifestEntries>
                <Content-Type>text/plain</Content-Type>
            </manifestEntries>
        </manifestSection>
    </manifestSections>
</archive>
</configuration>
</execution>
</executions>
</plugin>
<plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <version>${version.maven.antrun}</version>
    <executions>
        <execution>
            <id>filter-metadata</id>
            <phase>none</phase>
        </execution>
        <execution>
            <id>copy-jar-to-mtar</id>
            <phase>package</phase>
            <goals>
                <goal>run</goal>
            </goals>
            <configuration>
                <target>
                    <copy
                        file="${project.build.directory}/sap-xsac-opp-pps-${project.version}-
mta.jar"
                        tofile="${project.build.directory}/sap-xsac-opp-pps-${project.
version}.mtar" />
                </target>
            </configuration>
        </execution>
        <execution>
            <id>copy-SL_MANIFEST.XML</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>run</goal>
            </goals>
            <configuration>
                <target>
                    <copy file="${basedir}/SL_MANIFEST.XML" tofile="${project.build.
directory}/SL_MANIFEST.XML" />
                </target>
            </configuration>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>com.sap.lm.sl.alm.prod.assembler</groupId>
    <artifactId>alm-prod-assembler-maven-plugin</artifactId>
    <version>${version.maven.alm.assembler}</version>
    <configuration>
        <mtaSourceDirs>
            <param>${project.build.directory}</param>
        </mtaSourceDirs>
        <targetDir>${project.build.directory}</targetDir>
        <resultZip>${project.build.directory}/sap-xsac-opp-pps-${project.version}.zip<
/resultZip>
        <overwrite>true</overwrite>
    </configuration>

```

```

        <executions>
            <execution>
                <id>create-SCA</id>
                <phase>package</phase>
                <goals>
                    <goal>assemble</goal>
                </goals>
            </execution>
        </executions>
    </plugin>

    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>build-helper-maven-plugin</artifactId>
        <executions>
            <execution>
                <id>attach-distributions</id>
                <phase>verify</phase>
                <goals>
                    <goal>attach-artifact</goal>
                </goals>
                <configuration>
                    <artifacts>
                        <artifact>
                            <file>${project.build.directory}/sap-xsac-opp-pps-${project.version}.
mtar</file>
                            <type>mtar</type>
                        </artifact>
                        <artifact>
                            <file>${project.build.directory}/XSACOPPPPS${version.software.
component}.ZIP</file>
                            <type>zip</type>
                        </artifact>
                    </artifacts>
                </configuration>
            </execution>
        </executions>
    </plugin>
</plugins>
</build>
</project>

```

The `<build>` block in this pom file ensures the creation of the MTA and SCV files. All these files are located in Maven module **sap-xsac-opp-pps** (directly under the root folder).

4. A file is needed to define the *scopes*, *attributes* and *role-templates* of our application. This can be done with a file called **xs-security.json**. This file is also located directly under Maven module **sap-xsac-opp-pps**, and could look as follows:

xs-security.json

```

{
  "xsappname" : "ppservice-webapp-central",
  "scopes"    : [ {
    "name" : "${XSAPPNAME}.Calculate",
    "description" : "calculate" }
  ],
  "role-templates": [ {
    "name" : "PPE_ROLE_TEMPLATE",
    "description" : "PPE Role Template",
    "scope-references" : [
      "${XSAPPNAME}.Calculate" ]
    }
  ]
}

```

In this example, one *role-template* and one *scope* are defined. The scope is checked by the application router and in the **web.xml** file.

Price and Promotion Repository

This chapter describes how the price and promotion repository (*PPR*) is realized.

Overview

The effective sales price is calculated by the promotion pricing service. This service uses a promotion calculation engine and provides an interface (client API) to request a price calculation and an interface to read the data from the database (data access API). The data access API reads price-relevant data in an ARTS-like format. Therefore, we can speak of price rules that calculate the effective sales price. The price rules are maintained based on the DDF offer model that is currently included only in SAP Customer Activity Repository.

As the promotion calculation engine needs the data delivered in an ARTS-like format, the DDF offer has to be translated into this ARTS-like format and the translated price rule has to be stored in the OPP promotion.

The following sections give an overview of the modeling of an OPP promotion and the transformation of a DDF offer into an OPP promotion.

Modeling of OPP Promotions

In the OPP promotion, the entity **Promotion** represents the root entity. This entity consists of status information, validity, DDF offer ID, and others. A language-dependent promotion description is assigned to each OPP promotion in the **PromotionText** entity. At least one business unit is assigned in the **BusinessUnitAssignment** entity. Contains the DDF offer terms for product groups, the several items or product hierarchy nodes assigned to the product groups are stored as subentity to the **Promotion** in the **MerchandiseSet** entity.

An OPP promotion can have one or more promotion derivation rules that are independent of each other. For the customer who triggers the promotion, these promotion derivation rules represent the individual reward. Therefore, each **PromotionPriceDerivationRule** has one or more triggers (**PriceDerivationRuleEligibilities**) and one **PriceDerivationRule**. The **PromotionPriceDerivationRule** is effective if all assigned triggers are fulfilled. The following triggers are supported:

Trigger	Description
ItemPriceDerivationRuleEligibility	Is triggered if the specified item (can also include the specified quantity or unit of measurement) is in the shopping cart.
MerchandiseHierarchyPriceDerivationRuleEligibility	Is triggered if items from the specified merchandise group or article hierarchy node are in the shopping cart. The standard delivery supports two types of merchandise structures: <ul style="list-style-type: none">• Retailer's Merchandise Category Hierarchy• Retailer's Article Hierarchy
TotalPurchaseMarketBasketPriceDerivationRuleEligibility	Is triggered if the value of the shopping cart exceeds the specified threshold.
CouponPriceDerivationRuleEligibility	Is triggered if the specified coupon number is recorded in a sale.
CustomerPriceDerivationRuleEligibility	This entity associates a price derivation rule with a customer group. The customer card is the only condition in the DDF offer that is supported for the identification of a customer group. Therefore, the customer card type from the DDF, such as "Gold Card", is used as the customer group ID with OPP in the standard delivery. Individual card numbers are not supported.
ManualPriceDerivationRuleEligibility	Is triggered if a manual promotion is coming from the client, for example, by pressing a special key at the cash register. The DDF incentive concept is used to specify the manual promotion in the DDF offer.

	<p>For the incentive type Manual Promotions, you can use FreeText, Yes or No for Product is Required.</p> <p>If a product identifier is specified in the offer for the manual promotion, this product identifier and the incentive class identifier represent the manual promotion. If there is no product identifier specified in the offer, the incentive type code of the incentive and the incentive class identifier represent the manual promotion.</p>
CombinationPriceDerivationRuleEligibility	<p>Is triggered if the logical combination of its child triggers (Logic AND, Logic OR) is fulfilled.</p> <p>All eligibilities described above can be child eligibilities of this combination eligibility.</p> <p>This trigger can be used to create eligibility trees.</p>
MerchandiseSetPriceDerivationRuleEligibility	<p>Is triggered if the specified item is in the product group that is modeled as the merchandise set in the PPR.</p> <p>The specified item is in the merchandise set and valid as a trigger for the associated price derivation rule when:</p> <ul style="list-style-type: none"> ▪ The item itself or one of the product hierarchy node where the item is assigned is included in the product group and ▪ The item or one of the product hierarchy node where the item is assigned is not excluded in the product group <p>The standard delivery supports two types of merchandise structures:</p> <ul style="list-style-type: none"> • Retailer's Merchandise Category Hierarchy • Retailer's Article Hierarchy

Exactly one **PriceDerivationRule**, representing the reward, is assigned to a **PromotionPriceDerivationRule**. The following specific **PriceDerivationRules** are supported:

Reward	Description
ItemPriceDerivationRule	Denotes discounts for the items on the trigger side.
MixAndMatchPriceDerivationRule	<p>Allows more complex discounting.</p> <p>A MixAndMatchPriceDerivationRule refers to a set of MixAndMatchPriceDerivationItems that can be logically linked (AND/OR/SET).</p> <p>A MixAndMatchPriceDerivationItem specifies the PromotionalProduct (either a single product or a merchandise hierarchy group) for which the discount is to be applied, and the discount as such.</p>
ExternalActionPriceDerivationRule	<p>This kind of PriceDerivationRule does not define a specific reward or discount, but it contains information that is to be processed by the client of the promotion pricing service (PPS).</p> <p>The DDF incentives are used to provide information to the caller in a generic way.</p> <p>The promotion pricing service returns the information about the external action to the client.</p> <p>An ExternalActionPriceDerivationRule refers to a set of ExternalActionRuleParameters containing simple Key/Value pairs that can be interpreted by the caller.</p> <p>In the standard shipment, the following language-independent attributes of an incentive are provided (if filled) as ExternalActionRuleParameters:</p> <ul style="list-style-type: none"> • Product ID/free style ID • Incentive quantity • Incentive value • Incentive value adjustment <p>Additionally, the ExternalActionPriceDerivationRule refers to a set of ExternalActionRuleTexts containing the language-dependent texts for the external action.</p> <p>In the standard shipment, the attribute <i>Incentive Type description</i> is provided as ExternalActionRuleTexts.</p>

ManualPriceDerivationRule

This type of **PriceDerivationRule** specifies the item discount on trigger side, or determines that the item discount comes from the client.

Keys and Foreign Keys

Unique identifiers (IDs) are generated for the promotion-related entities during the mapping. A new number range object **/ROP/PROID** is used for this. Additionally, the identifier for the DDF offer is also in the **Promotion** entity.

Eligibilities can be modeled as condition trees. Therefore, all eligibility entities have also a **ParentPriceDerivationRuleEligibilityID** and a **RootPriceDerivationRuleEligibilityID** as a foreign key. In an eligibility tree, the **ParentPriceDerivationRuleEligibilityID** refers to the key of the parent node and the **RootPriceDerivationRuleEligibilityID** to the key of the root node. If the condition for the **PromotionPriceDerivationRule** is not a tree, the **ParentPriceDerivationRuleEligibilityID** and the **RootPriceDerivationRuleEligibilityID** are identical to the **PriceDerivationRuleEligibilityID**.

As the **PromotionPriceDerivationRule** provides the association between the eligibilities and price derivation rule, the **PriceDerivationRuleEligibilityID** and the **PriceDerivationRuleID** are foreign keys in this entity. For eligibility trees, the **PriceDerivationRuleEligibilityID** refers to the key of the root node.



All ABAP data elements referring to ABAP domain **/ROP/LONG** will be mapped to Java Long values in the promotion pricing service. In addition, the database type **BIGINT** will be used by default if the service is deployed locally. Therefore, values exceeding the range of Java Long must be avoided.

This is particularly important when defining the number range intervals for IDs of the promotion and other entities. Furthermore, this is important for the control parameters *sequence* and *resolution* of a promotion price derivation rule as these parameters refer to this domain. This means they cannot be provided with values outside of the Java Long range. The following tables show the difference in the value ranges:

Type	From	To
java.lang.Long	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
/ROP/LONG	-9,999,999,999,999,999,999	9,999,999,999,999,999,999

Validity Period for the OPP Promotion

The validity period for an OPP promotion (**EffectiveDate** and **ExpiryDate**) is mapped from the DDF offer. This date is interpreted as the local time of the client that is using the OPP promotion.

Database Tables

The OPP promotions are stored in the following database tables in SAP Customer Activity Repository:

- **/ROP/PROMOTION**
A table for promotion-relevant header data. A promotion can have one or more promotion price derivation rules.
- **/ROP/PROMO_RULE**
A table for promotion price derivation rules that provides the association between eligibility and price derivation rule to determine the price modification.
- **/ROP/ELIGIBILITY**
A table for all data that is relevant for the eligibilities of the promotion.
- **/ROP/PRICE_RULE**
A table for price derivation rules that represent the reward for the customer at the point of sale.
- **/ROP/MAM_ITEM**
A table for mix-and match price derivation items that specifies matching items that may be used to trigger the price derivation rule.
- **/ROP/PROMO_BU**
A table for the business units for which the promotion is relevant.
- **/ROP/PROMO_TEXT**
A table for the language-dependent texts of a promotion.
- **/ROP/EX_ACT_PARM**
A table for the language-independent attributes of an external action.
- **/ROP/EX_ACT_TEXT**
A table for the language-dependent texts of an external action.
- **/ROP/MERCH_SET**
A table to store the entries of the merchandise sets (product groups) within the promotion.

Handling of Amounts

In the database tables of an ABAP system, amounts are stored in a special format. In this format, amounts always have 2 decimals, regardless of whether this number of decimals is allowed for the corresponding currency of the stored amount. Consider the following examples (comma ',' used as thousands separator, dot '.' used as decimal mark):

Currency	Decimals	Amount	Value stored on DB (using a CURR 19,5 field)
EUR (Euro)	2	1234.56€	1,234.56000

JPY (Japanese Yen)	0	¥1234	12.34000
BHD (Bahrain Dinar)	3	1234.567 BD	12,345.67000


The correct display of the amounts within the using ABAP application is usually achieved via conversion exits on the UI level - within the program logic of ABAP application the database format is used. However, in the context of OPP, this storage of amounts has the following consequences if currencies with other than 2 decimals are used:

- Amounts sent via IDocs must be converted into an external format having the decimal mark at the correct position (for regular prices as well as promotional entities).
- Java applications directly accessing the database of the central Price and Promotion Repository must be aware of this format and must perform a scaling of values prior to the calculation. How this is done is explained in the documentation of the PPS module dataaccess-common.
- ABAP applications receiving amounts in external format having the decimal place at the right position (either within IDocs or when requesting the price calculation from a PPS) must convert between the ABAP internal representation of amount and the external format. In particular, this is the case for the integration of the PPS based price calculation into the ERP sales order processing.

Note that local copies of the Price and Promotion Repository exclusively used by the Java based PPS store amounts in the "natural" format, having the decimal place at the correct position. This is e.g. the case for the local PPS integrated into SAP Hybris Commerce.

The decimals of a currency are stored in an ABAP system in database table **TCURX**, containing only those currencies having not 2 decimals. The number of decimals also influences the rounding control data of an OPP promotion. By default discounts are to be rounded to the smallest amount which can be expressed in the corresponding currency.

Transformation from DDF offers into OPP Promotion

 In this chapter, the term *offer classes* reflects the result of the offer classification as, for example, *simple discount offer*.

Technical Information

The offer transformation transforms a DDF offer into an OPP promotion. This promotion is then saved in the price and promotion repository. This transformation is performed automatically during the creation and update as well as during the preceding validation of a DDF offer.

During the validation of the offer it is checked if the offer can be transformed into an OPP promotion. This depends on the offer types and the combination of offer features that are supported with OPP. The validation of the transformation of a DDF offer into an OPP promotion is triggered if the status of the offer is switched to a status that is relevant for transformation. The following table shows which offer status translate to which status of an OPP promotion.

DDF Offer Status	OPP Promotion Status	Comment
In Process	Inactive	OPP promotion will be written with this status only if it was previously in status "Active"
Recommended	Inactive	OPP promotion will be written with this status only if it was previously in status "Active"
Approved	Active	
Released	Active	
Cancelled	Inactive	OPP promotion will be written with this status only if it was previously in status "Active"
<Logically Deleted>	Cancelled	Actually not an offer status

The use of the offer statuses "Released" and "Cancelled" is controlled via a Customizing switch located in Customizing under *Cross-Application Components > Demand Data Foundation > Data Maintenance > Offer > Maintain Indicators for Offer Calculations*. The name of the switch is "Offer Status Management". For more information about offer status management, see the application help of SAP Promotion Management on SAP Help Portal at <https://help.sap.com/viewer/p/CARAB> > <Version> > *Application Help > SAP Promotion Management > Promotion Planning > Maintain Offers*. The mapping of status values is independent of this Customizing switch. The class **/ROP/CL_CONFIG** controls which values of the offer status are translated into status "Active" for an OPP promotion.

You can also manually transform DDF offers using program **/ROP/R_OFFER_TRANSFORM** in SAP Customer Activity Repository. This program reads all DDF offers with the relevant status according to the selection criteria and validates and converts the DDF offers into OPP promotions. Afterwards, it saves the OPP promotions in the SAP Customer Activity database for reuse. If an offer cannot be transformed, the other offers will still be processed using resumable exceptions. The following classes and BAdIs are relevant for the transformation of DDF offers into OPP promotions:

- **/ROP/CL_OFFER_MAPPER** is the entry point for the offer transformation. It expects a list of DDF offers and returns a list of OPP promotions. This class implements both the interface for the mapping and the validation of an offer.
- The mapping logic is realized by calling a number of BAdIs that are contained in enhancement spot **/ROP/OFFER_MAPPING**. These BAdIs offer (but do not enforce) a three-step process to :
 1. **Offer classification** (mandatory)
 - The offer is analyzed and classified in this step. For example, *Only BUY terms linked with OR*.
 - This step results in an offer classification, an offer classification group, and information about whether a promotion recipe has to be

created. The corresponding BAdI is **/ROP/OFFER_CLASSIFIER**. OPP offers an implementation using the class **/ROP/CL_OFFER_CLASSIFIER**.

2. Creation of a promotion recipe (optional)

A recipe can be created for a given classified offer in this step. A recipe is a structure (**/ROP/BL_PROMO_RECIPE_STY**) with detailed information about how to create the OPP promotions in step 3. The recipe determines the transformation from a high-level perspective. The following main instructions for the mapping are offered in the promotion recipe:

- How many promotion rules are to be created
- Which types of price rules are to be created
- Which offer terms are to be used to create eligibility trees and how these trees are to be combined
- Which offer terms are to be used to create mix-and-match items and how these items are to be combined

The corresponding BAdI is **/ROP/PROMO_RECIPE_BUILDER**. This BAdI has the classification group determined in step 1 as a filter. We offer one implementation using the class **/ROP/CL_PROMO_RECIPE_BUILDER**.

3. Building the promotion (mandatory)

In this step, the offer (and optionally the recipe determined in step 2) are used to create the promotion. If you are using the recipe, the implementation can be done in a generic and straightforward way. The corresponding BAdI **/ROP/PROMO_BUILDER** has the classification group determined in step 1 as a filter. We offer one implementation using the class **/ROP/CL_PROMO_BUILDER**.

- All three BAdIs have multiple implementations. In addition, the sequence in which the implementations are executed can be determined. This is done by implementing the BAdI **BADI_SORTER** for all BAdIs of the three-step process and by offering an execution sequence number that is specified for each BAdI implementation. SAP implementations have the sequence number 0. This means that you are free to add preprocessing (sequence number < 0) and postprocessing (sequence number > 0) steps for the SAP implementations. These SAP implementations can be deactivated.

For more information about the offer transformation, see the corresponding BAdI documentation for enhancement spot **/ROP/OFFER_MAPPING**.

Change pointers can be created when DDF offers are transformed into OPP promotions. These change pointers are used during the delta replication of the data replication framework (DRF). The change pointers are created using an implementation of the BAdI **/ROP/PROMO_CHANGE_POINTER** in enhancement spot **/ROP/PROMOTION_DB**. The standard SAP system offers an implementation of this BAdI using the class **/ROP/CL_PROMO_OUT_MDG_CP**. This class creates master data governance (MDG) change pointers based on the business object **ROP_PROMO**. You can use this BAdI to modify the pointer creation or implement your own pointer creation. If you do not want to use the DRF change message, or if no MDG change pointers are to be created, you can deactivate the BAdI implementation.

For more information about change pointers for the OPP promotion outbound, see the corresponding BAdI documentation for enhancement spot **/ROP/PROMOTION_DB**.

How We Transform DDF Offers into OPP Promotions

A DDF offer can have one or more BUY terms and no GET term. However, a BUY term in an offer is mandatory. A combination of BUY terms and one or more GET terms is possible. GET or BUY terms are logically linked with AND or OR. If terms linked with OR means that at least one term must match the basket and AND means that all terms must match the shopping cart (means all terms of the offer must be in the cart to get the reward). A discount can be defined on the BUY side and on the GET side of an offer. The GET lines are relevant only if the BUY lines are filled. However, the discounts of the BUY side become effective even if the GET side has no entries. The prerequisites for getting a reward can also be defined on the BUY side and on the GET side. Prerequisites are defined on the GET side if there are terms linked with AND on the GET side. To get the discount, all the products defined in the GET terms have to be in the shopping cart. A DDF offer can also have one or more incentives. These incentives can have the type *Condition* and the type *Reward*. The offers for all incentives supported by the OPP need to be linked with AND.

The ARTS-like OPP promotion makes a distinction between triggers and rewards. The transformation of offers into OPP promotions means that discounts granted on the offer BUY side are pulled to the ARTS reward side (**PriceDerivationRule**). Prerequisites defined on the GET side have to be pulled from the GET (reward) side to the trigger side. Rewards that are independent of each other (such as a reward defined in BUY terms linked with OR) lead to several **PromotionPriceDerivationRules**.

To simplify these complex transformation rules, several offer types are classified in offer classes. This classification is done in the BAdI **/ROP/OFFER_CLASSIFIER**. Based on an offer class, a recipe can be built that contains the construction information to create a promotion. This recipe is built in the BAdI **/ROP/PROMO_RECIPE_BUILDER**. Based on the offer class and the construction recipe, the mapping can be done in a generic straightforward way with the BAdI **/ROP/PROMO_BUILDER** that builds the promotion. These BAdIs are called during the validation and transformation of a DDF offer into an OPP promotion.

Sequence & Resolution

The OPP promotion data model offers the fields sequence and resolution that control the behavior in the following cases:

- Several OPP promotions related to the shopping cart are eligible for the same shopping cart.
- Several OPP promotions related to the line items are eligible for the same line item.

In this case, the sequence number determines the order in which the promotion price derivation rules are applied. If the sequence numbers are the same, only the promotion price derivation rule with the highest resolution number is applied. If the sequence number and the resolution number are the same, a best price calculation is performed.

Note that there is a strict separation of line item-related price rules and transaction-related price rules. All line item-related price rules are executed before all transaction-related, in other words the scope of the price rule can be seen as an additional sort criterion to the sequence numbers.

The sequence and resolution are set in the standard shipment as follows:

- The sequence of a promotion price derivation rule is the same as its ID.
Exception: If an offer consists of a BUY and a GET side, two promotion price derivation rules are created, both with the same sequence.
- The resolution of a promotion price derivation rule is set to 0.
Exception: If an offer consists of a BUY and a GET side, two promotion price derivation rules are created. The rule containing only the terms of the BUY side has the resolution 0, the rule containing the BUY and GET side has the resolution 1.

The sequence and resolution can be set for each promotion price derivation rule easily during creation of the promotion recipe.

Transformation of Simple Discount Offers

A simple discount is an offer without get terms that can have one or multiple buy terms with a defined discount. If this offer type has multiple buy terms, they are linked with Or and do not depend on each other. This offer type can be combined with incentives of class types Condition and Reward.

This offer type is a separate offer class. The SAP recipe for this offer class defines that one **PromotionPriceDerivationRule** is to be created for each BUY term, and the **PriceDerivationRules** assigned to the **PromotionPriceDerivationRules** are of type **ItemPriceDerivationRule**.


Handling of regular price and EDLP

By default, offers with discount type **Everyday Low Price (EDLP)** are transformed like offers with discount type **Regular Price**. In this case, no discount is defined, and the discount type **Everyday Low Price** can only be used to define a condition to get a reward. As of PPS 4.0, you can enable a zero discount for the offer in Customizing. In this case, offers are transformed into OPP promotions with a zero discount. For these promotions, a monetary discount of zero is applied to the previous price. Hence, a retail price modifier is returned in the calculation response, but the previous price does not change.

Examples

The following examples for simple discount offers and tables show how these offers are transformed into OPP promotions. The examples are restricted to the most relevant database fields. Fields that are always filled with a default value are listed separately in the section *Default Values*.

Example 1: Buy one item of product A for a discount of 10%, or buy three items of product B for a discount of 20%

 The *Enforce Multiple* indicator is set to *Yes* and no *Limit* is to be set.

This offer is translated to the following independent **PromotionPriceDerivationRules** for one promotion per product:

- A rule that sets a discount for each product A in the shopping cart.
 - A rule that sets a discount for each three items of product B.
- In this case, the customer has to purchase at least three items of product B (or multiples of three) to receive the discount. After the multiple of three is reached the remaining items will be sold at the regular price.

The following table shows how this example is translated to the price and promotion repository:

ENTITY	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1 StatusCode = AC
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = TP PriceModificationPercent = 10.000 DiscountMethodCode = 00
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 2 Sequence = 2 Resolution = 0
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = B ThresholdTypeCode = QUTI ThresholdQuantity = 3

		LimitQuantity = 9,999,999,999 IntervalQuantity = 3
	ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = TP PriceModificationPercent = 20.000 DiscountMethodCode = 00

Example 2: Buy for at least USD 50 and get a discount value of USD 10 for your shopping cart total



The *Enforce Multiple* indicator is set to *Yes* and *Limit* is to be set to 1.

The prerequisite *Minimum spend amount* for the transaction is modeled as **TotalPurchaseMarketBasketPriceDerivationRuleEligibility** on the eligibility side. The discount for the shopping cart is stored in the **ItemPriceDerivationRule** with a **PriceRuleControlCode** *SU* (*Transaction Discount Calculated After Subtotal*).

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
TotalPurchaseMarketBasketPriceDerivationRuleEligibility	TypeCode = TOTL TransactionItemTotalRetailTriggerAmount = 50.000 CurrencyCode = USD
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = SU PriceModificationMethodCode = RT PriceModificationAmount = 10.000 DiscountMethodCode = 00

Example 3: Buy for at least USD 50 from merchandise category MC1 and get a discount of 10%



The *Enforce Multiple* indicator is set to *Yes* and the *Limit* is to be set to 3.

The prerequisite *Minimum spend amount* for the merchandise category *MC1* is modeled as **MerchandiseHierarchyPriceDerivationRuleEligibility** on the eligibility side. The discount is stored in the **ItemPriceDerivationRule** with a **PriceRuleControlCode** *PO* (*Item Discount Calculated After Each Item*).

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
MerchandiseHierarchyPriceDerivationRuleEligibility	TypeCode = MSTR MerchandiseHierarchyGroupIDQualifier = 1 MerchandiseHierarchyGroupID = MC1 ThresholdTypeCode = AMTI ThresholdAmount = 50.000 LimitAmount = 150.000 IntervalAmount = 50.000 CurrencyCode = USD
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = TP PriceModificationPercent = 10.000 DiscountMethodCode = 00

i Merchandise Hierarchy

A merchandise hierarchy (DDF: product hierarchy) can be an article hierarchy or a merchandise category hierarchy. When replicated from SAP ERP, an article hierarchy has an alphanumeric indicator that uniquely identifies the article hierarchy. The merchandise category hierarchy has no such an indicator in SAP ERP. Therefore, the DDF default indicator for the merchandise category hierarchy is *1*. This value is also mapped to the price and promotion repository in the field **MerchandiseHierarchyGroupIDQualifier** and can be used to identify the merchandise category hierarchy. The identifier from SAP ERP is mapped to this field for article hierarchies.

Example 4: Buy three products of product group 'Yoghurt' for a discount price of USD 1.33

The product group 'Yoghurt' has the following components:

- Merchandise groups MC1 and MC2
- Item A and Item B are also included

! In this example the Customizing switch for using the enhanced product groups is inactive and so the inclusion of items and merchandise hierarchy nodes is supported.

The product group 'Yoghurt' consists of a subset of assignments of two merchandise categories and two single products. The *Enforce Multiple* indicator is set to *Yes*, the *Regular Price Only* indicator is also be set and no *Limit* is to be set.

This offer is translated to one **PromotionPriceDerivationRule**. As the threshold quantity is greater than one, the **PromotionPriceDerivationRule** is considered as only "Shopping Cart" relevant and the corresponding indicator (attribute **notConsideredInLineItemModeFlag**) is set. The product group is modeled as an eligibility tree with the two merchandise categories and the two products as child eligibilities below the **CombinationPriceDerivationRuleEligibility** that uses the combination code **OR** (OR with total quantity). The threshold quantity and the limit information is also stored in the **CombinationPriceDerivationRuleEligibility**. The discount is stored in the **ItemPriceDerivationRule** with a **PriceRuleControlCode** *PO* (*Item Discount Calculated After Each Item*). Information about the *Regular Price Only* indicator is stored in the **ItemPriceDerivationRule** (attribute **noPreviousMonetaryDiscountAllowedFlag**).

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0 NotConsideredInLineItemModeFlag= X
CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = OR ThresholdTypeCode = QUTI ThresholdQuantity = 3 LimitQuantity = 9,999,999,999 IntervalQuantity = 3 UomCode = PC
MerchandiseHierarchyPriceDerivationRuleEligibility	TypeCode = MSTR MerchandiseHierarchyGroupIDQualifier = 1 MerchandiseHierarchyGroupID = MC1 ThresholdTypeCode = COMB
MerchandiseHierarchyPriceDerivationRuleEligibility	TypeCode = MSTR MerchandiseHierarchyGroupIDQualifier = 1 MerchandiseHierarchyGroupID = MC2 ThresholdTypeCode = COMB
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = B
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = PT PriceModificationPercent = 1.33 DiscountMethodCode = 00 NoPreviousMonetaryDiscountAllowedFlag = X

Example 5: Buy three products of product group 'Yoghurt' for a discount price of USD 1.33

The product group 'Yoghurt' has the following components:

- Merchandise group MC1 is included
- Merchandise group MC2 is excluded
- Item A and Item B are also included



In this example the Customizing switch for using the enhanced product groups is active. The *Enforce Multiple* indicator is set to *Yes*, the *Regular Price Only* indicator is also be set and no *Limit* is to be set.

This offer is translated to one **PromotionPriceDerivationRule**. As the threshold quantity is greater than one, the **PromotionPriceDerivationRule** is considered as only "Shopping Cart" relevant and the corresponding indicator (attribute **notConsideredInLineItemModeFlag**) is set.

The product group is modeled as **MerchandiseSetPriceDerivationRuleEligibility**, which contains the threshold values and a reference to the **MerchandiseSet**, which is a subentity to the **Promotion**. The components of the product group (items and merchandise hierarchy nodes) are stored in the **MerchandiseSet**. The **MerchandiseSet** consists of a root node with type code **OPR**. Below this root node, there are the items and merchandise groups modeled as child nodes. The *Combination '1'* for child nodes marks the node as included, the *Combination '2'* means the child node is excluded in the product group. The discount is stored in the **ItemPriceDerivationRule** with a **PriceRuleControlCode** *PO (Item Discount Calculated After Each Item)* The information about the *Regular Price Only* indicator is stored in the **ItemPriceDerivationRule** (attribute **noPreviousMonetaryDiscountAllowedFlag**).

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	PromotionID = 1
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0 NotConsideredInLineItemModeFlag = X
MerchandiseSetPriceDerivationRuleEligibility	TypeCode = MSET ThresholdTypeCode = QUT1 ThresholdQuantity = 3 LimitQuantity = 9,999,999,999 IntervalQuantity = 3 MerchandiseSetID = 123
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = PT PriceModificationPercent = 1.33 DiscountMethodCode = 00 NoPreviousMonetaryDiscountAllowedFlag = X
MerchandiseSet	MerchandiseSetNodeID = 100 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100 Combination = 0 TypeCode = OPR Operation = DF PromotionID = 1
MerchandiseSet	MerchandiseSetNodeID = 101 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100 Combination = 1 TypeCode = ITEM ItemID = A PromotionID = 1
MerchandiseSet	MerchandiseSetNodeID = 102 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100 Combination = 1 TypeCode = ITEM ItemID = B PromotionID = 1
MerchandiseSet	MerchandiseSetNodeID = 103 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100 Combination = 1 TypeCode = MSTR

	MerchandiseHierarchyGroupID = MC1 MerchandiseHierarchyGroupIDQualifier = 1 PromotionID = 1
MerchandiseSet	MerchandiseSetNodeID = 104 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100 Combination = 2 TypeCode = MSTR MerchandiseHierarchyGroupID = MC2 MerchandiseHierarchyGroupIDQualifier = 1 PromotionID = 1

Example 6: Simple Discount Offer with Target Groups

Target groups can be used to provide a customized offer to a specific customer base to maximize margins or sales.

In an SAP CARAB context a target group consists of a set of customers, suspects, or contact persons, categorized according to criteria, such as geographical location, or common interest that can be used in marketing activities to promote products or services. Based on a target group, the marketing expert can create a campaign to prepare for follow-on actions, such as marketing campaigns. A target group in an SAP CARAB environment contains the several information, such as key information (targeted group ID), status, member.

As of CARAB 4.0 FP2, offers with target groups are processed and transformed into OPP promotions.

During the transformation of an offer with target group into the OPP promotion, the target group ID is written in the customer group ID.

One or more target groups can be assigned to an offer. However, already one target group can trigger an assigned promotion.

The transformation of offers with target groups is similar to transformation of offers with incentives, e.g. the eligibility tree is also built up with an And-linking.

Example: Buy one item of product A and get a 10% discount if you belong to at least one of 2 offer target groups TG1 and TG2.

If both possible target groups are assigned to the offer, the offer is transformed into the following OPP promotion:

ENTITY		Field Mapping
Promotion		
	PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
	CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUT1 ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1 StatusCode = AC
	CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode =
	CustomerPriceDerivationRuleEligibility	TypeCode = CGRP CustomerGroupID = TG1
	CustomerPriceDerivationRuleEligibility	TypeCode = CGRP CustomerGroupID = TG2
	ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = TP PriceModificationPercent = 10.000 DiscountMethodCode = 00

If only one of the possible target groups is assigned to the offer, the offer is transformed as follows:

--	--

ENTITY		Field Mapping
Promotion		
	PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
	CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1 StatusCode = AC
	CustomerPriceDerivationRuleEligibility	TypeCode = CGRP CustomerGroupID = TG1
	ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = TP PriceModificationPercent = 10.000 DiscountMethodCode = 00

Transformation of Offers with Transaction Discount

You use offers with transaction discount to define a discount for an entire shopping cart.

Offers with a transaction discount can have more than one buy term. All other buy terms (except the one term defining the transaction discount) must have discount type **Regular Price**, or **EDLP** if **Zero Discount** is not enabled in the Customizing.

Additionally, offers with transaction discount can have a get term. In this case, the get term must define the transaction discount and no further discounts are supported.



The Enforce Multiple Indicator is set to Yes and no Limit is set

As of CARAB 4.0 FP2, offers with transaction discount are transformed into OPP promotions. This offer type is transformed as a simple discount offer. The type code of the `ItemPriceDerivationRule` is **RB** (Simple Discount) and the price rule control code is **SU** (Transaction Discount). The offer term which contains the condition to get the reward is only used on the eligibility side (see Example below: *Buy for at least USD 50 of merchandise category MC1*).

Example: Buy for at least USD 50 of merchandise category MC1 and get a discount of 10% on your entire shopping cart

Entity		Field Mapping
Promotion		
	PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
	MerchandiseHierarchyPriceDerivationRuleEligibility	TypeCode = MSTR MerchandiseHierarchyGroupIDQualifier = 1 MerchandiseHierarchyGroupID = MC1 ThresholdTypeCode = AMTI ThresholdAmount = 50.000 LimitAmount = 99,999,999,999,999.000 IntervalAmount = 50.000 CurrencyCode = USD
	ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = SU PriceModificationMethodCode = TP PriceModificationPercent = 10.000 DiscountMethodCode = 00

Transformation of Mix-and-Match Offers

A mix-and-match offer is an offer with buy terms that are linked with And or with get terms. It defines a combination of products and product counts that results in a customer reward when purchased together.

This reward can affect prices of products that do not trigger this offer type, for example, buy item A and get items B, C or D at 50% off. Items B, C and D get the reward and item A would be the trigger item. Items B, C and D are linked to the sale of item A. It is also possible to give a discount on the products that are the trigger, for example, buy product A and B and get product A for a discount of 50%. In both cases, a mix-and-match offer depends on the content of an entire transaction.

This offer type can be divided into three offer classes with different recipes for the promotion building:

- An offer class for offers with buy terms that are linked with AND and without GET terms.
The recipe defines that only one **PromotionPriceDerivationRule** is to be created for all BUY terms. Furthermore, it defines that the **PriceDerivationRule** assigned to the **PromotionPriceDerivationRule** is of type **MixAndMatchPriceDerivationRule** and that all BUY terms are linked with AND on the eligibility side.
- An offer class for offers with BUY and GET terms that are linked with AND.
The recipe defines that two **PromotionPriceDerivationRules** are to be created. The first **PromotionPriceDerivationRule** is defined only for the reward on the buy side. Therefore, the **PriceDerivationRule** assigned to the **PromotionPriceDerivationRule** is of type **MixAndMatchPriceDerivationRule** and all BUY terms with a discount are **MixAndMatchItems** that are linked with AND. On the eligibility side all BUY terms are also linked with AND. The second **PromotionPriceDerivationRule** is defined for the reward on the GET side. The **PriceDerivationRule** assigned to the **PromotionPriceDerivationRule** is of type **MixAndMatchPriceDerivationRule** and all GET terms with a discount are **MixAndMatchItems** that are linked with AND. On the eligibility side all BUY and all GET terms are linked with AND.
- An offer class for offers with BUY and GET terms in which the BUY terms are linked with OR and the GET terms are linked with AND. This means that each BUY term defines a reward, independent of the content of the entire transaction. The reward on the GET side is given only if all products from the GET side are in the shopping cart and if at least one of the conditions from the buy side is fulfilled.
The recipe defines that one **PromotionPriceDerivationRule** for each BUY term with a discount is to be created and that the **PriceDerivationRule** assigned to the **PromotionPriceDerivationRule** is of type **ItemPriceDerivationRule**. Furthermore, a **PromotionPriceDerivationRule** for the GET terms is to be created that refers to an eligibility tree in which all BUY terms are linked with OR and all GET terms are linked with AND. The linkage between BUY and GET terms is also AND. The **PriceDerivationRule** assigned to the **PromotionPriceDerivationRule** is of type **MixAndMatchPriceDerivationRule** and all GET terms with a discount are **MixAndMatchItems** linked with AND.



Regular Price and Everday Low Price

The discount types *Everyday Low Price* and *Regular Price* do not define a discount. They are used only to define a condition (eligibility) to get a reward. They will be included in the eligibilities but no **MixAndMatchItems** will be created for them.

Examples

The following section contains some examples for mix-and-match offers and tables that show how these offers are transformed into OPP promotions.

The mapping examples are restricted to the most relevant database fields. Fields that are always filled with a default value are listed separately.

Example 1: Buy one item of product A for a discount price of USD 2.99 and buy one item of product B for a discount of USD 2

The discount for the products is given in this offer only if the two products are purchased together.

This offer is translated to one **PromotionPriceDerivationRule**. The two products are combined with AND as eligibilities and **MixAndMatchPriceDerivationItems**.

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = B ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
MixAndMatchPriceDerivationRule	TypeCode = MM CombinationCode = && PriceRuleControlCode = PO DiscountMethodCode = 00
MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = A PriceModificationMethodCode = PS NewPriceAmount= 2.99 RequiredQuantity = 1
MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = B

					PriceModificationMethodCode = RS PriceModificationAmount = 2.00 RequiredQuantity = 1
--	--	--	--	--	---

Example 2: Buy one item of product A for its regular price and buy one item of product B for a discount of 10% and you will get one item of product A for free

The discount for product B is given in this offer only if at least one item of product A is purchased. The reward for product A (one item for free) is given only if the customer buys at least two items of product A and one additional item of product B.

This offer is translated to the following two independent **PromotionPriceDerivationRules** with the same sequence number but different resolution numbers:

- The first rule is for the reward on the buy side. The two products are eligibilities and are combined with AND. The **PriceDerivationRule** is of type **MixAndMatchPriceDerivationRule** and the discount for product B is provided as **MixAndMatchPriceDerivationItem**.
- The second rule is for the reward on the get side. The two products from the BUY terms plus the product from the get side are eligibilities and are combined with AND. The **PriceDerivationRule** is of type **MixAndMatchPriceDerivationRule** and product B that has a discount on the BUY side and the GET reward for product A are provided as **MixAndMatchPriceDerivationItem**.

The following table shows how this example is translated to the price and promotion repository:

Entity		Field Mapping
Promotion		
	PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
	CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = B ThresholdTypeCode = QUT ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
	MixAndMatchPriceDerivationRule	TypeCode = MM CombinationCode = && PriceRuleControlCode = PO DiscountMethodCode = 00
	MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = B PriceModificationMethodCode = RP PriceModificationPercent = 10.000 RequiredQuantity = 1
	PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 2 Sequence = 1 Resolution = 1
	CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = B ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI

			ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
		MixAndMatchPriceDerivationRule	TypeCode = MM CombinationCode = && PriceRuleControlCode = PO DiscountMethodCode = 00
		MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = A PriceModificationMethodCode = PS NewPriceAmount= 0.000 RequiredQuantity = 1
		MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = B PriceModificationMethodCode = RP PriceModificationPercent = 10.000 RequiredQuantity = 1

Example 3: Buy for at least USD 50 and you will get product A for free

The discount for product A is given in this offer only if the customer buys for at least USD 50. This offer translates to one **PromotionPriceDerivationRule**. The transaction condition and the item condition are linked with AND as eligibilities. The discount for product A is modeled as **MixAndMatchPriceDerivationItem**.



The Enforce Multiple flag is to be set to YES and the limit must be 1.

The following table shows how this example is translated to the price and promotion repository:

Entity		Field Mapping
Promotion		
	PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
	CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
	TotalPurchaseMarketBasketPriceDerivationRuleEligibility	TypeCode = TOTL TransactionItemTotalRetailTriggerAmount = 50.000 CurrencyCode = USD
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 1 IntervalQuantity = 1
	MixAndMatchPriceDerivationRule	TypeCode = MM CombinationCode = && PriceRuleControlCode = PO DiscountMethodCode = 00
	MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = A PriceModificationMethodCode = PS NewPriceAmount= 0,000 RequiredQuantity = 1

Example 4: Buy three products of product group 'Yoghurt' and get one product of product group 'Yoghurt' for free. The product group 'Yoghurt' has the following components:

- Merchandise group MC1 is included

- *Merchandise group MC2 is excluded*

- *Item A and Item B are also included*



In this example the Customizing switch for using the enhanced product groups is active. The *Enforce Multiple* indicator is set to *Yes* and no *Limit* is to be set.

This offer is translated to one **PromotionPriceDerivationRule**, the Buy and the Get condition are linked with AND as eligibilities **MerchandiseSetPriceDerivationRuleEligibility**, which contains the threshold values and a reference to the **MerchandiseSet**, which is a subentity to the **Promotion**.

As the threshold quantity is greater than one, the **PromotionPriceDerivationRule** is considered as only "Shopping Cart" relevant and the corresponding indicator (attribute **notConsideredInLineItemModeFlag**) is set.

The components of the product group (items and merchandise hierarchy nodes) are stored in the **MerchandiseSet**. The **MerchandiseSet** consists of a root node with type code **OPR**. Below this root node are the items and merchandise groups modeled as child nodes. The *Combination '1'* for child nodes marks the node as included, the *Combination '2'* means the child node is excluded in the product group. The discount is modeled as **MixAndMatchPriceDerivationItem**. This **MixAndMatchPriceDerivationItem** contains again a reference to the **MerchandiseSet**, the type code for the item is **PG** (product group).

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	PromotionID = 1
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0 NotConsideredInLineItemModeFlag= X
CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
MerchandiseSetPriceDerivationRuleEligibility	TypeCode = MSET ThresholdTypeCode = QUTI ThresholdQuantity = 3 LimitQuantity = 9,999,999,999 IntervalQuantity = 3 MerchandiseSetID = 123
MerchandiseSetPriceDerivationRuleEligibility	TypeCode = MSET ThresholdTypeCode = QUTI ThresholdQuantity = 3 LimitQuantity = 9,999,999,999 IntervalQuantity = 3 MerchandiseSetID = 123
MixAndMatchPriceDerivationRule	TypeCode = MM CombinationCode = && PriceRuleControlCode = PO DiscountMethodCode = 00
MixAndMatchPriceDerivationItem	TypeCode = PG PriceModificationMethodCode = PS NewPriceAmount= 0,000 RequiredQuantity = 1 MerchandiseSetID= 123
MerchandiseSet	MerchandiseSetNodeID = 100 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100 Combination = 0 TypeCode = OPR Operation = DF PromotionID = 1
MerchandiseSet	MerchandiseSetNodeID = 101 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100 Combination = 1 TypeCode = ITEM ItemID = A PromotionID = 1
MerchandiseSet	MerchandiseSetNodeID = 102 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100

	Combination = 1 TypeCode = ITEM ItemID = B PromotionID = 1
MerchandiseSet	MerchandiseSetNodeID = 103 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100 Combination = 1 TypeCode = MSTR MerchandiseHierarchyGroupID = MC1 MerchandiseHierarchyGroupIDQualifier = 1 PromotionID = 1
MerchandiseSet	MerchandiseSetNodeID = 104 MerchandiseSetID = 123 ParentMerchSetNodeID = 100 RootMerchSetNodeID = 100 Combination = 2 TypeCode = MSTR MerchandiseHierarchyGroupID = MC2 MerchandiseHierarchyGroupIDQualifier = 1 PromotionID = 1

Transformation of Packaged Offers

A packaged offer is a bundling of different items with individual sales prices. When brought together this bundle is sold at a fixed price. The different items are specified as buy terms and linked with And. This offer type is a separate offer class.

Example: *Buy two products of merchandise category MC1 and one item of product A for a fixed total price of USD 24.99*



The *Enforce Multiple* indicator must be set to *Yes* and no *Limit* is to be set.

The fixed total price for the products is given in this offer only if the specified products are purchased together.

This offer is translated to one **PromotionPriceDerivationRule**. The buy terms are combined with AND as eligibilities, the set price is stored in the **ItemPriceDerivationRule** with a **PriceModificationMethodCode** *ST* (*Total Set Price*). The package apportioned discount percentages that can be maintained during the offer maintenance are not considered.

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
MerchandiseHierarchyPriceDerivationRuleEligibility	TypeCode = MSTR MerchandiseHierarchyGroupIDQualifier = 1 MerchandiseHierarchyGroupID = MC1 ThresholdTypeCode = QUTI ThresholdQuantity = 2 LimitQuantity = 9,999,999,999 IntervalQuantity = 2
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO

PriceModificationMethodCode = ST NewPriceAmount = 24,99 DiscountMethodCode = 00
--

Transformation of Offers with Incentives

Incentives can be combined with both simple discounts and mix-and-match offers. The following examples show how to transform offers with incentives into OPP promotions:

Example 1: Mix-and-match offer with incentive category *customer card*

Pay with your gold card and buy one item of product A for a discount price of USD 3.33, or buy product B for a discount of 10% and get 50% off for two items of product C.



The *Enforce Multiple* indicator is to be set to *Yes* and no *Limit* is to be set.

The discounts for the products A and B are given in this offer only if the customers pay with their gold card. The reward for the two items of product C is given only if the customers buy at least two items of product C, one item of product A, one item of product B and additionally pay with their gold card.

This offer is translated to four independent **PromotionPriceDerivationRules**:

- The first and the second rule are for the reward on the buy side, one for each BUY term. These two **PromotionPriceDerivationRules** have different sequence numbers and the resolution number is 0. On the eligibility side, each BUY term results in an **ItemPriceDerivationRuleEligibility** and is linked with AND with a **CustomerPriceDerivationRuleEligibility**. The **PriceDerivationRules** assigned to the **PromotionPriceDerivationRules** have the type **ItemPriceDerivationRule**.
- The third and the fourth **PromotionPriceDerivationRule** are for the reward on the get side. The third **PromotionPriceDerivationRule** combined the first BUY term (product A) with the GET term and the customer card, this **PromotionPriceDerivationRule** has the same sequence number as the first **PromotionPriceDerivationRule** that contains only the discount from product A and the resolution number is 1. The fourth **PromotionPriceDerivationRule** combined the second BUY term (product B) with the GET term and the customer card, this **PromotionPriceDerivationRule** has the same sequence number as the second **PromotionPriceDerivationRule** that contains only the discount from product B and resolution number is also 1.
- Eligibility trees are built on the eligibility side. The **CustomerPriceDerivationRuleEligibility** is combined with the GET term and each with one of the BUY terms via AND linkage. The **PriceDerivationRule** assigned to the **PromotionPriceDerivationRules** has the type **MixAndMatchPriceDerivationRule**. The discount for the GET term is defined in the **MixAndMatchPriceDerivationItem** and via AND linkage combined with one of the BUY terms (the same as defined in the eligibility tree).

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
CustomerPriceDerivationRuleEligibility	TypeCode = CGRP CustomerGroupID = GOLD
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUT1 ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = PT NewPriceAmount = 3.33 DiscountMethodCode = 00
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 2 Sequence = 2 Resolution = 0
CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
CustomerPriceDerivationRuleEligibility	TypeCode = CGRP CustomerGroupID = GOLD
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = B ThresholdTypeCode = QUT1

			ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
		ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = TP PriceModificationPercent = 10.000 DiscountMethodCode = 00
		PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 3 Sequence = 1 Resolution = 1
		CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
		CustomerPriceDerivationRuleEligibility	TypeCode = CGRP CustomerGroupID = GOLD
		ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = C ThresholdTypeCode = QUT1 ThresholdQuantity = 2 LimitQuantity = 9,999,999,999 IntervalQuantity = 2
		ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUT1 ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
		MixAndMatchPriceDerivationRule	TypeCode = MM CombinationCode = && PriceRuleControlCode = PO DiscountMethodCode = 00
		MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = C PriceModificationMethodCode = RP PriceModificationPercent = 50.000 RequiredQuantity = 2
		MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = A PriceModificationMethodCode = PS NewPriceAmount = 3.33 RequiredQuantity = 1
		PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 4 Sequence = 2 Resolution = 1
		CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
		CustomerPriceDerivationRuleEligibility	TypeCode = CGRP CustomerGroupID = GOLD
		ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = C ThresholdTypeCode = QUT1 ThresholdQuantity = 2 LimitQuantity = 9,999,999,999 IntervalQuantity = 2
		ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = B ThresholdTypeCode = QUT1 ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
		MixAndMatchPriceDerivationRule	TypeCode = MM CombinationCode = && PriceRuleControlCode = PO DiscountMethodCode = 00
		MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = C PriceModificationMethodCode = RP PriceModificationPercent = 50.000 RequiredQuantity = 2
		MixAndMatchPriceDerivationItem	TypeCode = IT ItemID = B

PriceModificationMethodCode = RP PriceModificationPercent = 10.000 RequiredQuantity = 1

Example 2: Simple discount offer with incentive category *Show Coupon*

Show coupon 0815 and buy one item of product A for a discount of 10% or buy three items of product B for a discount price of USD 10.



The *Enforce Multiple* indicator is to be set to *No* and no *Limit* is to be set.

In this example, the coupon is a condition. This means that the customer has to show the corresponding coupon in order to be eligible for the offer. Incentives are always linked with AND to its offer. So the offer in this example is translated to the following independent **PromotionPriceDerivationRules**, one for each product:


- A rule that defines a discount for each product A that is in the shopping cart. On the eligibility side the prerequisite *Coupon* is modeled as **CouponPriceDerivationRuleEligibility** and linked with AND to the **ItemPriceDerivationRuleEligibility** for product A. The linkage with AND is done via a **CombinationPriceDerivationRuleEligibility**.
- In this rule, the prerequisite *Coupon* is linked with AND to the **ItemPriceDerivationRuleEligibility** for product B. Even though the discount price in the offer is defined for three items of product B, the threshold quantity is always 1 in the **ItemPriceDerivationRule** because the *Enforce Multiple* indicator in the offer is set to *No*. So this rule will not require the quantity criteria to be met. For any quantity in this example, the discount unit price will be USD 3.33 (results from USD 10 divided by three items) in the **ItemPriceDerivationRule**.

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
CouponPriceDerivationRuleEligibility	TypeCode = COUP CouponNumber = 0815 ConsumptionTypeCode = 00
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = RP PriceModificationPercent = 10.000 DiscountMethodCode = 00
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 2 Sequence = 2 Resolution = 0
CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = B ThresholdTypeCode = QUT ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
CouponPriceDerivationRuleEligibility	TypeCode = COUP CouponNumber = 0815 ConsumptionTypeCode = 00
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = PS NewPriceAmount= 3.33000 DiscountMethodCode = 00

Example 3: Simple discount offer with incentive category *Get Coupon as Reward*

Buy one item of product A for *Everyday Low Price* and *Get coupon 0815* as reward.

 The *Enforce Multiple* indicator is to be set to *Yes* and no *Limit* is to be set.

 The discount type **Zero Discount** is not enabled in Customizing.


The offer in this example is translated to one **PromotionPriceDerivationRule**. Product A with *Everyday Low Price* is the condition to get a coupon as a reward. As *Everyday Low Price* does not define a discount, product A is used only on the eligibility side. The reward is modeled as **ItemPriceDerivationRule** with a **PriceRuleControlCode** *PO* (*Item Discount Calculated After Each Item*) and **DiscountMethodCode** *04* (*A coupon is given to the customer instead of a discount*).

The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO PriceModificationMethodCode = RT PriceModificationAmount = 0.01 DiscountMethodCode = 04 CouponPrintOutID = 0815 CouponPrintoutRule = 00 CouponValidityInDays = 0

Example 4: Simple discount offer with incentive category *Get Points as Reward*

When they show coupon 0815, the customer will get an extra 25 bonus points when they buy product A.

 The *Enforce Multiple* indicator is to be set to *Yes* and no *Limit* is to be set.

The offer in this example is translated to one **PromotionPriceDerivationRule**. On the eligibility side, the prerequisite *Coupon* is linked with AND to the **ItemPriceDerivationRuleEligibility** for product A. The reward is modeled as **ItemPriceDerivationRule** with a **PriceRuleControlCode** *PO* (*Item Discount Calculated After Each Item*) and **RewardGrantedAsLoyaltyPoints** is set (*X*).


The following table shows how this example is translated to the price and promotion repository:

Entity	Field Mapping
Promotion	
PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
CouponPriceDerivationRuleEligibility	TypeCode = COUP CouponNumber = 0815 ConsumptionTypeCode = 00
ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = PO

					PriceModificationMethodCode = RT PriceModificationAmount = 25.000 DiscountMethodCode = 00 RewardGrantedAsLoyaltyPoints = X CalculationBase = 00
--	--	--	--	--	--

Example 5: Incentive category *External Action*

Today you can get the shipping for product A for only USD 5.

 The *Enforce Multiple* indicator is to be set to *Yes* and no *Limit* is to be set.

You have created incentive type *DSHP - Discount Shipping* for an *External Action* incentive and you use the incentive value to maintain the special price for shipping.


The offer in this example is translated to one **PromotionPriceDerivationRule** with a **ItemPriceDerivationRuleEligibility** for product A. The reward is modeled as **ExternalActionPriceDerivationRule** with type code *EX (External Action)*. The incentive type *DSHP* is mapped into the **ExternalActionID**. The maintained incentive value and the incentive value adjustment are modeled as **ExternalActionRuleParameter**. The incentive type description is modeled as **ExternalActionRuleText**.

The following table shows how this example is translated to price and promotion repository:

Entity		Field Mapping
Promotion		
	PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
	ExternalActionPriceDerivationRule	TypeCode = EX PriceRuleControlCode = SU ExternalActionID = DSHP
	ExternalActionRuleParameter	ParameterID = SAP_INC_VALUE Value = 5.000
	ExternalActionRuleParameter	ParameterID = SAP_INC_VALUE_ADJUST Value = 5.000
	ExternalActionRuleText	LanguageCode = EN TextCode = SAP_INC_TYPE_DESCR Text = 'Discount Shipping'

Example 6: Incentive category *Manual Promotion as Reward*

When buying product A: manually triggered discount is allowed.

 Enforce Multiple indicator is to be set to *Yes* and no *Limit* is to be set.

You have created incentive class *31 - Manually triggered Discount* with incentive class type *Reward*. For this incentive class, you have also created incentive type *M2* with Incentive category *Manual Promotion*. You use the product identifier as free text to identify the manual promotion.

The offer in this example is translated to one **PromotionPriceDerivationRule** with a **ManualPriceDerivationRuleEligibility** for the manual promotion and a **ItemPriceDerivationRuleEligibility** for product A as child eligibilities below the **CombinationPriceDerivationRuleEligibility**.

On reward side, there is a **ManualPriceDerivationRule** with type code *MA (Manual Promotion)*. This price rule does not specify a discount, but the discount can be specified by the client, for example, the cashier can specify the exact amount and type of the discount.

The following table shows how this example is translated to price and promotion repository:

--	--	--	--	--	--

Entity		Field mapping
Promotion		
	PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
	CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
	ItemPriceDerivationRuleEligibility	TypeCode = ITEM ItemID = A ThresholdTypeCode = QUTI ThresholdQuantity = 1 LimitQuantity = 9,999,999,999 IntervalQuantity = 1
	ManualPriceDerivationRuleEligibility	TypeCode = MANU TriggerTCD = 31 TriggerValue = Defect (Free text coming from the product identifier)
	ManualPriceDerivationRule	TypeCode = MA PriceRuleControlCode = PO

Example 7: Incentive category *Manual Promotion as Condition*

Get 10 % discount for your transaction when manual promotion is triggered and when a specified customer card is shown.



Enforce Multiple indicator is to be set to Yes and the limit must be 1.

You have created incentive class '30 - Manual Trigger for discount' with incentive class type 'Condition'. For this incentive class, you have also created incentive type 'M1' with incentive category 'Manual Promotion'. You do not use the product identifier to identify the manual promotion, but the incentive class and the incentive type.

The offer in this example is translated to one **PromotionPriceDerivationRule**. On eligibility side the prerequisite *Customer Card* is modeled as with a **CustomerPriceDerivationRuleEligibility** and linked with AND to the **ManualPriceDerivationRuleEligibility** and to the **TotalPurchaseMarketBasketPriceDerivationRuleEligibility**. The linkage with AND is done via a **CombinationPriceDerivationRuleEligibility**.

On reward side, there is a **ItemPriceDerivationRule** with type code RB (Simple Discount). This price rule specifies the 10 % discount for the transaction if the prerequisite 'GOLD card' is fulfilled and the manual trigger is coming from the client, for example, the cashier presses a 'Manual Promotion' button on the point of sale).

The following table shows how this example is translated to price and promotion repository:

Entity		Field mapping
Promotion		
	PromotionPriceDerivationRule	PromotionPriceDerivationRuleID = 1 Sequence = 1 Resolution = 0
	CombinationPriceDerivationRuleEligibility	TypeCode = COMB CombinationCode = &&
	TotalPurchaseMarketBasketPriceDerivationRuleEligibility	TypeCode = TOTL TransactionItemTotalRetailTriggerAmount = 0.000 CurrencyCode = USD
	ManualPriceDerivationRuleEligibility	TypeCode = MANU TriggerTCD = 30 TriggerValue = M1
	CustomerPriceDerivationRuleEligibility	TypeCode = CGRP CustomerGroupID = GOLD
	ItemPriceDerivationRule	TypeCode = RB PriceRuleControlCode = SU PriceModificationMethodCode = TP PriceModificationPercent = 10.000 DiscountMethodCode = 00

Default Values

ItemPriceDerivationRule

- **RoundDestinationValue**
This value defines the multiple of the lowest allowed digit according to the currency to which rounding takes place.
Example: If the currency is EUR, the value 5 means that the rounding should not be done down to single cents but to 5-cent multiples.
If the default value is 1, there is no further handling of the rounding result.
- **RoundingMethodCode**
The default value is 00 (*Commercial Rounding*).
- **ConsiderPreviousPriceRules**
This indicator controls whether the current price derivation rule is based on the result of formerly applied rules. If this indicator is false, the rule is to be applied on the regular sales price. This indicator is relevant only if the **priceRuleControlCode** is PO (*Item Discount Calculated After Each Item*).
The default value is true.
- **CalculationBaseSequence**
This value defines the sequence value for **PromotionPriceDerivationRule**. The resulting price is to be used as the calculation base for the current rule.
The default value is -1, which means that none of the previous rules are considered and the regular price is used as the calculation base.
- **ChooseItemMethodCode**
This code defines the sequence in which the items to be discounted are chosen in the case of a **MixAndMatchPriceDerivationRule**.
The default value is 00 (*Determined by the Promotion Calculation Engine*).
- **CalculationBase**
The default value is 00, which means that the total sales is the calculation base for this rule.
- **DiscountMethodCode**
The default value is 00, which means that the discount reduces the transaction total.
Exceptions:
If the **PriceDerivationRule** has the type *External Action*, this field is not mapped.
If a coupon is given to the customer instead of a discount, the value is 04.
- **NoEffectOnSubsequentRules**
The default value is false, except for coupons.

Fields Only Relevant for Coupons

The following fields are only relevant and filled if the customer gets coupons instead of a discount:

- **CouponPrintoutRule**
The coupon printout rule defines the printout type that is to be given to the customer.
The default value is 00, which means that a coupon is to be printed on a separate document.
- **CouponValidityInDays**
The default value for the validity period for printout coupons is 0, which means that the coupon has no validity limit.
- **NoEffectOnSubsequentRules**
The default value is true for coupons.

Fields Only Relevant for Loyalty Points

The following field is only relevant and filled if the customer gets loyalty points instead of a discount:

- **RewardGrantedAsLoyaltyPoints**
The default value is true if the type of reward is loyalty points.

CouponPriceDerivationRule Eligibility

- **ConsumptionTypeCode**
The default value is 00, which means that coupons are also consumed if **PromotionPriceDerivationRules** is applied with a different sequence.

Note

You can have promotions if the same coupon should trigger more than one **PromotionPriceDerivationRule**. For example: If they show a coupon, the customer gets a discount for a certain product and additional loyalty points.

This promotion only works with **ConsumptionTypeCode** 02, which means 'Coupon Is Not Consumed'. In this case, you have to adapt the default value.

PromotionPriceDerivationRule

- **SaleReturnTypeCode;**
This value specifies if the promotion rule can be used only for sales, for returns, or for both.
The default value is 00 (*For Sales and Returns*).
- **Exclusive**
Specifies if this promotion rule is an exclusive promotion rule.
The default value is false.

- **NotPrinted**

If this indicator is true, the result of the promotion rule is to be suppressed.

The default value is false, except of null discount promotions which are introduced with CARAB 4.0 FP02.

- **NotConsideredInLineItemModeFlag**

Specifies if the promotion rule is applied to prices calculated by the promotion pricing service. It can be applied in the following modes:

- Item mode (*LineItem*)

If the promotion pricing service is called in this mode, the discount is calculated independently for each item. Promotion rules that are not relevant on item level are not applied (for example, promotion rules on transaction level).

- Shopping cart mode (*Basket*)

If the promotion pricing service is called in this mode, the discount is calculated for the total of the shopping cart considering all promotion rules.

If this indicator is set, the promotion rule is only applied in shopping cart mode.

Per default, promotion price derivation rules are considered as only "Shopping Cart" relevant and the default value for this field is true, if the following parameters are fulfilled:

- It is not product-related, for example discounts-based on transaction level, product groups or product hierarchies
- The offer type is mix-and-match
- The product quantity is greater than one
- The minimum spend amount is set
- A coupon fulfills the condition

Replication of the Price and Promotion Repository

The OPP promotions and the regular prices can be replicated to an external system via IDocs. Enhancement segments have been designed so that additional information can be added to the IDoc. For more information about the extensibility of the IDocs, see the *OPP Extensibility* section in this guide.

Outbound Processing of IDocs via DRF

The data replication framework (DRF), a reuse component of SAP Business Suite, is used to replicate the OPP promotions and the regular prices to other systems.

The following ways of replicating data are supported with the OPP:

- Initial replication
- Manual request
- Change request

The initial replication is used to send all relevant data for a receiver by one single request. The initial load expects to have no data on the receiver side.

The change request considers only objects that have to be sent compared to the previous (initial or delta) replication. Usually (but not always, see promotion outbound processing) this includes objects that have changed since the last transfer and that match the specified filter criteria. If an object is considered as transfer relevant, it is sent as a whole. There is no support for marking object internal changes. The initial and delta request share a common filter, the static filter maintained in transaction DRFF.



There is no support for considering changes of static filters before a delta request. If you change the filter, relevant changes may not be detected by the system.

The manual request allows the replication of specific data that can be filtered by adhoc specified filter criteria. There is no merge logic for the static and manual filter. The manual request does not modify the list of objects that are marked as changed since the last initial or delta load.



Use the manual request to make urgent fixes only.

DRF Configuration

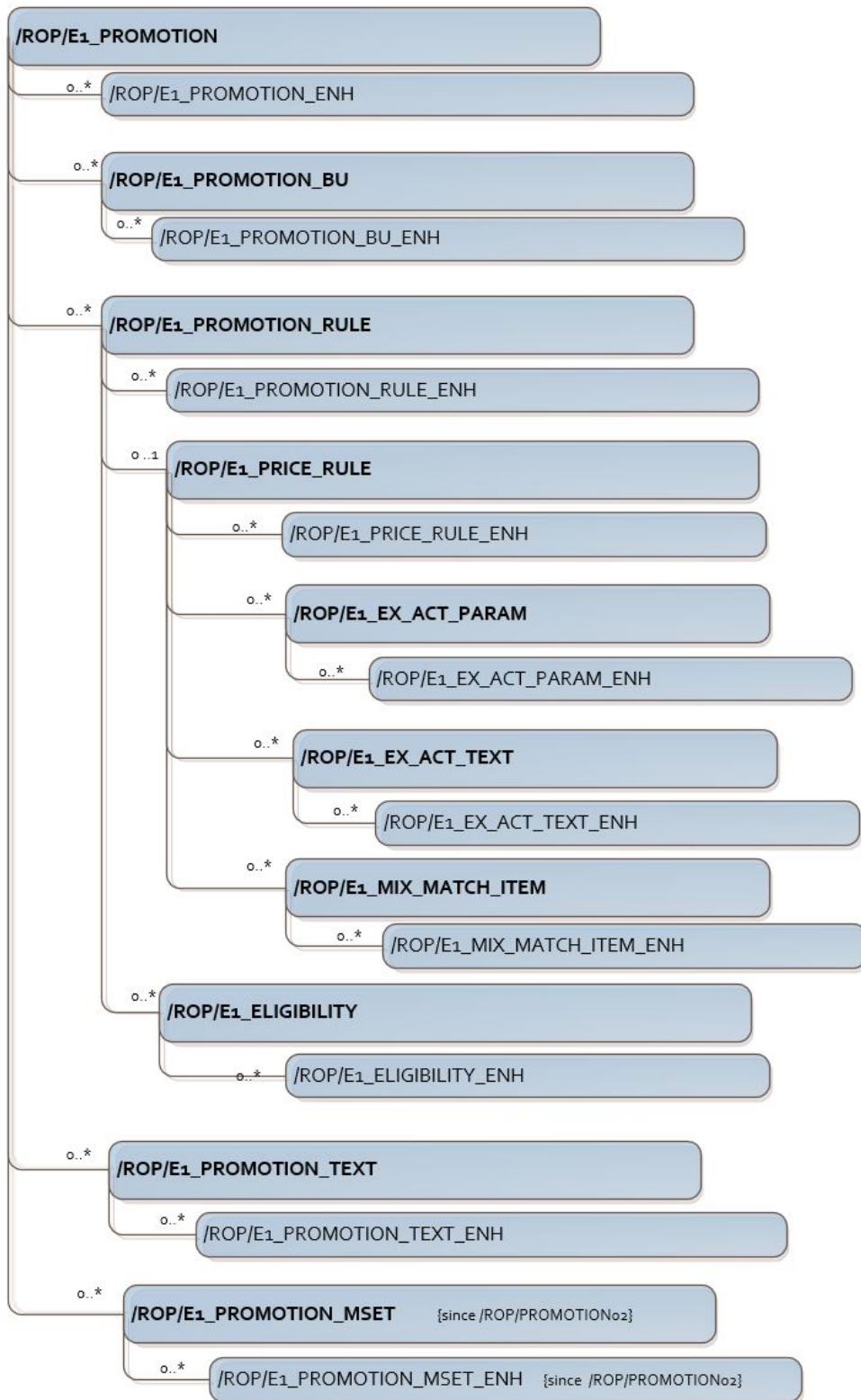
A configuration needs to be done before DRF can be used. SAP delivers outbound implementations and preconfigured settings for the outbound implementations, such as predefined outbound parameters, filter objects, and business objects. Customizing needs to be enhanced for these pre-delivered conditions only if you want to replicate your own business objects and create or enhance the outbound implementation.

The following custom settings are needed for the data replication:

- The landscape definition (determines the technical settings for business systems)
- The replication models (determines the data that is to be sent to a corresponding location)
- The business object-specific settings (Application Link Enabling)


OPP Promotions

The outbound interface that is needed to send OPP promotions to external systems is based on IDocs. IDoc types **/ROP/PROMOTION01** and **/ROP/PROMOTION02** are provided for this. The following picture shows the structure for IDoc type **/ROP/PROMOTION02**:




This structure reflects the database structure of the OPP promotion. The corresponding message type for the IDoc types is **/ROP/PROMOTION**.

As with the regular prices, the outbound is realized using the data replication framework (DRF). Different outbound implementations and filter objects are offered for this.

 The implementation of the Promotion Outbound assumes that the combination of outbound implementation and assigned business system is unique. This is a slightly different assumption than that made by DRF, which expects (and ensures) that the combination of business object type and business system is unique. This does not make a difference for the standard delivery but it must be kept in mind if you create your own outbound implementations based on the logic delivered by SAP.

As of CAR 3.0 FP2, the outbound of OPP promotions is supported in the following ways:

- The business object-centric outbound as offered starting with CAR 2.0 FP3: The underlying principle of this option is to replicate the business object structure of the OPP promotion as it is and to make no changes to the message content. The business system assigned to the corresponding DRF replication model determines the receiver of the created IDocs.
- The location-specific outbound: This is optimized for the supply store like receivers not interested in the whole content of the OPP promotion - in particular with regards to the overall set of location assigned to that promotion. The business units assigned to the OPP promotion determine the receivers of the created IDocs. Each receiver gets only a view to the OPP promotion, with only "his" location assigned. In addition, OPP promotions that do not have the status "active" or are no longer relevant for the receiver of the IDoc are transferred only in a truncated version containing only the header with CHANGE_INDICATOR set to 'D' and the assigned location/business unit.

 Further differences between these two options are listed in the SAP application help.

Outbound Implementation for Promotion-Centric Outbound Processing

The predefined outbound implementation for OPP promotions is **ROP_PROMO**. This implementation uses filter object **ROP_PROMO**. The filter execution time during change analysis is predefined in the data processing and you cannot change it when you configure a replication model. This means that the filter is always applied after the change analysis. The outbound implementation class is **/ROP/CL_PROMOTION_OUTBOUND**. This class implements interface **IF_DRF_OUTBOUND**. This outbound implementation has two predefined outbound parameters:

- **TASK_SIZE_PROCMMSG**
This parameter is relevant only if you execute data replication and have set the indicator to *Parallel Processing*. The parameter sets the maximum number of OPP promotions processed in each parallel package.
- **PACK_SIZE_BULK**
This parameter sets the maximum number of OPP promotions processed for each IDoc. If you want to use the parallel processing, set this parameter to a smaller value than parameter **TASK_SIZE_PROCMMSG**.

In addition to these a specific OPP outbound parameter is given:

- **/ROP/GENERIC_ENH_MAP**
This parameter enables DRF outbound for promotion to execute a generic mapping. In that case it must be set to "X". For more information about this feature, see below under chapter "Generic Mapping of Customer Enhancement Segments".

Outbound Implementation for Location-Specific Outbound Processing

The predefined outbound implementation for OPP promotions is **ROP_PRO_ST**. This implementation uses filter object **ROP_PRO_ST** containing two filters: the same filter as for the business object-centric outbound for determining the OPP promotions, and an additional filter for specifying the target locations of the IDocs to be created. The filter execution time during change analysis is predefined in the data processing and you cannot change it when you configure a replication model. This means that the filter is always applied after the change analysis. The outbound implementation class is **/ROP/CL_PROMO_STORE_OUTBOUND**. This class implements interface **IF_DRF_OUTBOUND**. The supported outbound parameters are the same as for the business object-centric outbound.

Filtering the OPP Promotions

Data filtering allows you to replicate specific OPP promotions. The following criteria can be used for filtering:

Field	In Static Filter /ROP /PROMO_DRF_FILTER_STY	In Manual Request Filter /ROP /PROMO_DRF_MAN_FILTER_STY	Comment
Master data system	✔	✔	
Sales organization	✔	✔	List of single values only, no exclusions
Distribution channel	✔	✔	List of single values only, no exclusions
Location hierarchy type	✔	✔	
Location hierarchy ID	✔	✔	
Location hierarchy node ID	✔	✔	
Location ID	✔	✔	
Location type	✔	✔	
Promotion ID	✔	✔	
External ID of the promotion (the offer ID)		✔	

Promotion type	✓	✓	
Start of the validity period		✓	Daily granularity only
End of the validity period	✓	✓	Daily granularity only
Lead time in days	✓		Single value only, no exclusion
Latest change date		✓	Daily granularity only

The filter class is **/ROP/CL_PROMOTION_FILTER** for the business object-centric outbound and **/ROP/CL_PROMO_STORE_FILTER** for the location-specific outbound. Both classes implement interface **IF_DRF_FILTER**.



There are no filter criteria for the external action attributes as these attributes are only subordinated elements of the price rule. From a business point of view, filtering by these fields is not relevant.

Controlling the Target Locations

This is relevant only for the location-specific outbound. The following criteria can be used to specify the target locations and therefore the set of IDocs to be created:

Field	In Static Filter /ROP /PROMO_STO_FILTER_STY	In Manual Request Filter /ROP /PROMO_STO_MAN_FILTER_STY	Comment
Target location ID	✓	✓	
Target location type	✓	✓	
Flag "Send Also Deletions"		✓	Only single values "Yes" or "No" allowed. If set to "Yes", then both target location ID and target location type must be specified.

The meaning of the "Send Also Deletions" flag is described in the system documentation for data element **/ROP/SEND_DELETIONS**.

Generic Mapping of Customer Enhancement Segments

When doing simple customer enhancements in the OPP data model by adding additional attributes so called customer includes (SAP CI) might be implemented. Each OPP table contains such a possibility to add customer specific attributes. The DRF outbound for sending OPP promotions offers a possibility to map these additional attributes to the corresponding enhancement segment of the IDoc type **/ROP/PROMOTION01** or **/ROP/PROMOTION02** respectively in a generic way. Each IDoc segment of the OPP promotion IDoc types includes a corresponding enhancement segment (see above) which structure is well defined. It contains 3 fields: One for the field group (filled with "SAP_CI" when generic mapping is active), a second one for the attribute name (generically filled with customer's attribute name) and a third one for the attribute value (generically filled with the corresponding attribute value). From customer point of view these enhancement segments can be mapped by implementing a BAdI or by activating a generic mapping that executes a 1:1 mapping from the additional attribute to the enhancement segment. The generic mapping feature can be activated by a specific DRF outbound parameter called **/ROP/GENERIC_ENH_MAP**. (This OPP specific parameter exists beside of the DRF standard parameters already mentioned above). When creating the DRF outbound replication model for OPP Promotions this parameter must be maintained and set to "X". Doing this the generic mapping is activated. Nevertheless, a combination of this 1:1 mapping and a more complex mapping process implemented by a BAdI is possible.

In the following overview all types are listed that can be used for this generic mapping:

- Character Container and Strings
- Numerical Characters (n)
- Long, Integer, Short, Byte
- Packed Number (p)
- Float, Decfloat
- Date
- Time

There are following restrictions:

- Internal tables
- References
- Deep structures
- RAW
- RAWSTRING
- Boxed Components
- Strings longer than 255 characters

Transfer OPP Promotions Using the Global Object List

The following applies for the business object-centric outbound as well as the location-specific outbound.

During the initial and delta load, the filter criteria and the database table **/ROP/DRF_OBJLIST** are evaluated to decide which OPP promotions have been changed and are to be replicated. This list serves the following purposes:

- It detects that a formerly relevant and transferred OPP promotion is obsolete. This may happen if an attribute of an OPP promotion (such as promotion type) is specified in the filter but its new value no longer matches the filter. This must be communicated to the corresponding receiver.
- It supports the filter criterion *Lead Time*. This makes sure that an OPP promotion is not transferred unless it is close to its validation date. To keep track of these OPP promotions, it is necessary to observe OPP promotions that are to be valid soon so that they are sent via the delta load even if there have been no changes. If not, only OPP promotions with unprocessed change pointers are to be considered.

In addition, MDG change pointers are created for the delta load when creating, updating, and deleting an OPP promotion.

The following logic is applied, depending on whether an OPP promotion matches filter criteria and its transfer status in the global object list:

	Promotion Matches Complete Filter Criteria	Promotion Matches Filter Criteria Without Lead Time	Promotion Does Not Match Filter Criteria
OPP promotion in global object list in status TRANSFERRED	1	2	3
OPP promotion in global object list in status PENDING	4	5	6
OPP promotion not in global object list	7	8	9

Cases 1 to 9 are described in detail below including the system reaction:

1. A promotion already transferred has changed --> transfer again. No change to the global object list.
2. A promotion already transferred is classified as not yet transfer relevant. This occurs if the start date of the promotion has been delayed. The receivers must be informed about this change --> transfer again. No change to the global object list.
3. A promotion already transferred is not filter relevant any more, in other words it is now obsolete. This may happen if the filter criteria are defined for an attribute that changed to a value not covered by the filter --> transfer the promotion as "obsolete" (CHANGE_INDICATOR = 'D'). Remove it from the global object list.
4. A promotion with a pending transfer has reached its transfer due date. (Transfer due date = valid_from (of the promotion) MINUS "lead time") --> if not in status "cancelled" send it, set its status in the global object list to TRANSFERRED. Cancelled promotions with a pending transfer are removed from the global object list.
5. A promotion with a pending transfer has been changed but has not yet reached its transfer due date --> if it is not in status "cancelled", do not transfer (yet) but update its transfer due date in the global object list (if valid_from has changed). The promotion will be considered again in the next delta load. Status stays at PENDING. Cancelled promotions are not added to the global object list.
6. A promotion originally set as pending (to be transferred later) is not transfer relevant any more --> since it has not yet been transferred, do not transfer it, and remove it from the global object list.
7. A promotion not examined before is transfer relevant now --> send it and include it in the global object list in status TRANSFERRED.
8. A promotion not examined before is considered as transfer relevant later --> do not transfer it yet but include it in the global object list in status PENDING with the corresponding transfer due date.
9. A promotion not examined before is not considered as transfer relevant --> ignore.

If the corresponding promotion has the status 'CN' (Cancelled), no insert or update to the global object list takes place - instead the promotion is removed from the global object list. This happens in the following cases: 1, 2, 4, 5, 7 and 8. The decision matrix for the initial load differs from that of the delta load in the sense that rows 1 and 2 are not relevant since the global object list is cleared at the beginning of the initial load.

All replication modes (initial, delta, manual) update the global object list.

If no lead time is specified in the static filter, an "infinite" lead time is assumed. This means no promotion is set to pending. In other words, column 2 ("Promotion matches filter criteria without lead time") is not relevant.

The initial load expects that all data is cleared on the receiver side, in other words the receiver must not have any promotions in its database. The initial load automatically clears the global object list for the corresponding outbound implementation and business systems. For the decision matrix, the initial load corresponds to the row "promotion not in global object list". For the initial load, only promotions in status 'AC' (active) are considered.

The delta load and manual request do not filter by the promotion status.

Obsolete or deleted?

The meaning of the field CHANGE_INDICATOR differs between the object-centric and location-specific outbound of OPP promotions:

- Object-centric outbound: If a promotion is logically deleted, it is sent as a regular IDoc record with CHANGE_INDICATOR = 'I'. Its promotion status is 'CN'. If a promotion is considered as obsolete for a certain receiver, it is sent as a "deletion" IDoc record with CHANGE_INDICATOR = 'D'. Its promotion status is not changed.
- Location-specific outbound: CHANGE_INDICATOR is set to 'D' as soon as the corresponding OPP promotion is no longer to be evaluated by the receiver. This can be the case if it is not in status "active", if the corresponding target location is not assigned to the OPP promotion, or if the target location is not contained in the filter for target locations.

Location-Specific Outbound Processing Using the Global Object List

In the case of the location-specific outbound, the tracking of the replication status on business system level is not sufficient, it must take place on the level of the individual target location. This status is stored in database table `/ROP/LOC_REPL_ST`. Each record indicates that the corresponding OPP promotion is expected to be present as active on target location side. The link between the overall replication status and the location-specific replication status is established using the field OBJ_GUID in both tables `/ROP/DRF_OBJLIST` and `/ROP/LOC_REPL_ST`.

The meaning of the overall replication status slightly changes for the location-specific outbound:

- If a record is not present in **/ROP/DRF_OBJLIST** then the promotion does not exist as active in any target location and hence no record exists **/ROP/LOC_REPL_STAT** for that promotion. The reverse conclusion is not possible.
- If a record is in status T(ransferred) in **/ROP/DRF_OBJLIST** then it was sent as active to at least one target locations. This does not necessarily mean that this is still the case.
- The meaning of the status P(ending) does not change.

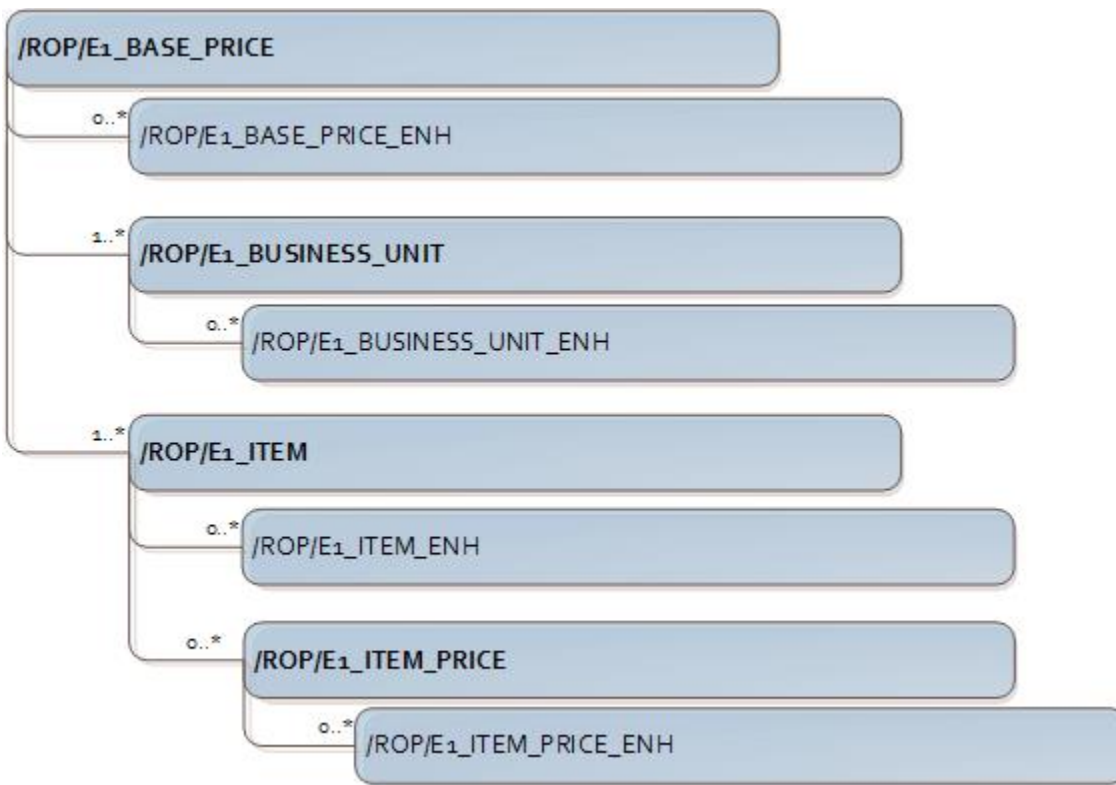
View **/ROP/V_PREPSTAT** provides an overview of the current replication status for each location.

Cleanup of the Global Object List

In the case of an unchanged DRF replication model, the global object list is automatically managed by the promotion outbound processing and kept in a consistent state. However, if a replication model is deleted, a business system for a replication model is removed, its content of the global object list is not removed automatically. For this purpose, transaction **/ROP/DEL_REPLSTAT** can be used. This can be used for the object-centric as well as the location-specific replication status.

Regular Prices

The outbound interface to send regular prices to external systems is also based on IDocs. For this reason, a new IDoc type **/ROP/BASE_PRICE01** has been created with the following structure:



Compared to the promotion IDoc, the regular price IDoc is quite flat. To prevent redundant data being sent, all items with the same regular price are grouped in several business units (locations) and all locations that are assigned to the corresponding items (items with the same regular price in the grouped locations) are assigned to the **/ROP/E1_BASE_PRICE** segment. This explains the following structure of the IDoc:

Segments of segment type **/ROP/E1_BUSINESS_UNIT** under one **/ROP/E1_BASE_PRICE** segment contain all business units (locations) with a unique item price. The advantage of this structure is that these locations have to be added only once to the IDoc. This applies to all items under the same **/ROP/E1_BASE_PRICE** segment. Therefore, the group for these locations is called *reusable location group*. The price is a child segment of the item segment. This item segment only contains the item ID and the change indicator.

In addition to the new price IDoc type, there is a new message type **/ROP/BASE_PRICE**.

Outbound Implementation

The predefined outbound implementation for OPP promotions is **ROP_PRICE**. This implementation uses filter object **ROP_PRICE**. The filter execution time during change analysis is predefined in the data processing. You cannot choose when you configure a replication model. Therefore, the filter time must be set to *Filter Before Change Analysis* when you create the replication model. Furthermore, you cannot activate a replication model with a wrong filter time or execute an outbound implementation in transaction **DRFOUT**. The outbound implementation class is **/ROP/CL_BASE_PRICE_OUTBOUND**. This class implements interface **IF_DRF_OUTBOUND**. The option for parallel processing is supported for this outbound implementation.

The outbound implementation has the following predefined outbound parameters:

- **TASK_SIZE_PROCMSG**

This parameter is relevant only if you execute the data replication using parallel processing. It sets the maximum number of products that are processed for each parallel package. It must be greater than or equal to the PACK_SIZE_BULK parameter. This parameter value does not define the number of regular prices per package.

If this parameter is set to 0, all products are processed in one package. This means that parallel processing is not possible.

PACK_SIZE_BULK

This parameter controls the number of products for which regular prices can be stored in a compressed format at the same time, and sets the maximum number of products that are processed for each IDoc. If this parameter is not set, the default is 1.

If you increase this value, performance at runtime is improved since fewer IDocs need to be processed.

- **/ROP/PACK_SIZE_BULK**

This parameter sets the maximum number of regular prices that are processed for each IDoc. This is an approximate value because regular prices are assigned to different IDocs for each group of business unit with items and prices.

If this parameter is set to 0, it is not possible to restrict regular prices and it is only the number of products that determines the IDoc size.

Hint: Both parameters **PACK_SIZE_BULK** and **/ROP/PACK_SIZE_BULK** restrict the size of an IDoc in a way that the IDocs are as small as possible.

Example 1: Assume **PACK_SIZE_BULK** = 500 and **/ROP/PACK_SIZE_BULK** = 100000. The system reads all prices for 500 products that are, for example, 500000. The system will create 5 IDocs and each IDoc will have 100000 prices.

Example 2: Assume **PACK_SIZE_BULK** = 50 and **/ROP/PACK_SIZE_BULK** = 100000. The system reads all prices for 50 products that are, for example, 50000. The system will create 1 IDoc and this IDoc will have 50000 prices.

- **/ROP/SEQ_READ_SIZE**

This parameter sets the maximum number of products for which the regular prices are read in one select statement. In this way, you can limit memory consumption for products with a large number of regular prices.

If this parameter is set to 0, all products of the corresponding package are read within one call.

- **/ROP/DAY_OFFSET_PAST**

This parameter is only used if the selection of prices lying in the past is restricted with a valid-to date as filter criteria and if the entered valid-to date is not far enough in the past.

During a delta replication, this parameter defines a time range in days that lies before the date of the last replication run. If the the entered valid-to date is after the calculated date, the system subtracts this value from the last replication date and uses the calculated date to construct the select-option for the valid-to date.

During an initial replication, a calculated date (current date minus the time range in days) is defined in this parameter. This date is used automatically if the value entered in field **End of Validity Period** is after the calculated date.

In this way, you ensure that regular prices with a valid-to date in the specified past time range are also transferred.

If this parameter is not set, relevant regular prices might not be transferred. See SAP Note 2338714. In this case, the default is set to 30 days.



All the recommendations for parameter values given above are based on performance measurements. These can be changed depending on the actual customer-specific runtime behavior and situation.

Data Filtering

Data filtering allows you to replicate specific prices. Regular price outbound filtering uses a complex filter. You need to distinguish between manual request, initial, and delta load. The following table gives an overview of the filter attributes:

Field	In Static Filter /ROP /BASE_PRICE_DRF_FILTER_STY	In Manual Request Filter /ROP /BASE_PRICE_MAN_FILTER_STY	Comment
Master data system	✓	✓	
Sales organization	✓	✓	List of single values only, no exclusions
Distribution channel	✓	✓	List of single values only, no exclusions
Location hierarchy type code	✓	✓	Necessary to uniquely identify a location
Location hierarchy ID (external)	✓	✓	
Location hierarchy node ID (external)	✓	✓	
Location ID (external)	✓	✓	
Location type code	✓	✓	
Qualifier of merchandise structure	✓	✓	Only for article hierarchy and merchandise group

Identifier for merchandise hierarchy node	✓	✓	
Product identifier		✓	Only available for manual load
Classification information for regular price	✓	✓	Fixed values are provided
End of validity period	✓		Daily granularity only. Only one filter criteria for inclusion allowed with "is later than".
Date of latest change		✓	Daily granularity only

It is possible to maintain one or more single values for each criteria. For most of them it is also possible to maintain ranges (except of sales organization and distribution channel). A combination is also possible for the filter criteria.

The filter criterion *End of Validity Period* is provided only for delta and initial load. This parameter could be used to improve the runtime behavior by reducing the data load. You can use this parameter to reduce the number of selected prices. You can also prevent the sending of obsolete price records. The attribute *Classification Information for Regular Price* is mandatory due to performance reasons. The price outbound implementation does not process only regular prices, it can also process other price types like *Average Purchase Price* and *Delivery Cost*. Usually only net or gross sales prices are chosen using this application.

The attribute *Date of Latest Change* is available only for the manual load. It has only a daily granularity. Therefore, several select options are possible and will be interpreted as follows:

- *Equal*: internally time interval 00:00:00 to 23:59:59 is applied because externally only a daily granularity is given
- *Greater Than*: internally time is set to 23:59:59
- *Greater Equals Than*: internally time is set to 00:00:00
- *Lower Than*: internally time is set to 00:00:00
- *Lower Equals Than*: internally time is set to 23:59:59
- *Between*: internally for start date time 00:00:00 is used and for end date 23:59:59

The filter class is `/ROP/CL_BASE_PRICE_FILTER`. This class implements interface `IF_DRF_FILTER`.

The defined filter time can be configured when you create the replication model. However, you must set the filter time to *Filter Before Change Analysis*. It is not possible to activate the DRF replication model with a filter criterion other than this one. So there is a preselection of the regular price objects before change analysis is started. This is done due to performance issues.

Handling of the Expected Data Volume

As we expect mass data in the price outbound, SAP implements a special logic for filtering and processing the price data.

To avoid memory issues for mass data, the data filtering does not provide all relevant item price attributes to be processed. Instead, the filter provides the following information for the outbound implementation based on the selection criteria:

- All product IDs (GUIDS)
- All locations (GUIDS) if there is a restriction by the selection screen; if there is no restriction, no locations are passed.
- All selection criteria as provided by the selection screen

This data is passed as "relevant objects" (import parameter `CT_RELEVANT_OBJECTS`) to the outbound implementation. There is one entry for each product ID in this internal table and the selected locations and the selection criteria are given in the first entry. The tables `LOCATIONS` and `SELECT_OPTIONS` are empty in all subsequent entries. The internal table has the structure `/ROP/BASE_PRICE_PACKAGE_STY`.

If you start the outbound processing in manual or initial mode, the internal table `CT_RELEVANT_OBJECTS` contains all the products provided by the filter. This information is passed to the major outbound process (class `/ROP/CL_BASE_PRICE_OUTBOUND`). Due to performance reasons, the option for parallel processing can be used. During parallel processing the table `CT_RELEVANT_OBJECTS` contains the number of products specified with parameter `TASK_SIZE_PROCMMSG` for each call of the outbound implementation. The construction of these parallel packages is carried out in `IF_DRF_OUTBOUND-BUILD_PARALLEL_PACKAGE`.

The processing of `CT_RELEVANT_OBJECTS` takes place in the outbound implementation in the following main steps:

1. DRF method `IF_DRF_OUTBOUND-READ_COMPLETE_DATA`
To avoid memory problems, the prices are **not** read in this method. Instead the data in `CT_RELEVANT_OBJECTS` is stored only in the instance variables `MT_PRODUCTS`, `MT_LOCATIONS` and `MT_SELECT_OPTIONS`.
2. Method `IF_DRF_OUTBOUND-MAP_DATA2MESSAGE`
This method is called from the DRF framework for each entry in `CT_RELEVANT_OBJECTS`.

The entries in `MT_PRODUCTS` (all products that were in `CT_RELEVANT_OBJECTS` before) are divided into logical packages with the size given in parameter `PACK_SIZE_BULK`. When method `IF_DRF_OUTBOUND-MAP_DATA2MESSAGE` is called for the first product of one of these logical packages, all prices are read for all products of this package. The prices for all products of this logical package are not read within one select statement because this could generate memory issues. Instead, only the number of products defined in parameter `/ROP/SEQ_READ_SIZE` are read in one select statement. The result of the select statement is compressed before the next select statement is performed. One part of the compressed result is stored in the table `LT_BASE_PRICE` and the second part, the reusable location groups, are collected by class `/ROP/CL_LOCATION_GROUP_HANDLER`. Using these two data sources, the instance table `MT_BASE_PRICE_IDOC` is created that contains all pricing information.

Even though the DRF framework calls this method sequentially for each product in **CT_RELEVANT_OBJECTS**, prices are read for all products of the package when the first product is processed. If an error occurred during reading of the prices for a product, the exception **CX_DRF_PROCESS_MESSAGES** is only raised when method **IF_DRF_OUTBOUND-MAP_DATA2MESSAGE** is called for the entry in **CT_RELEVANT_OBJECTS** containing that erroneous product.

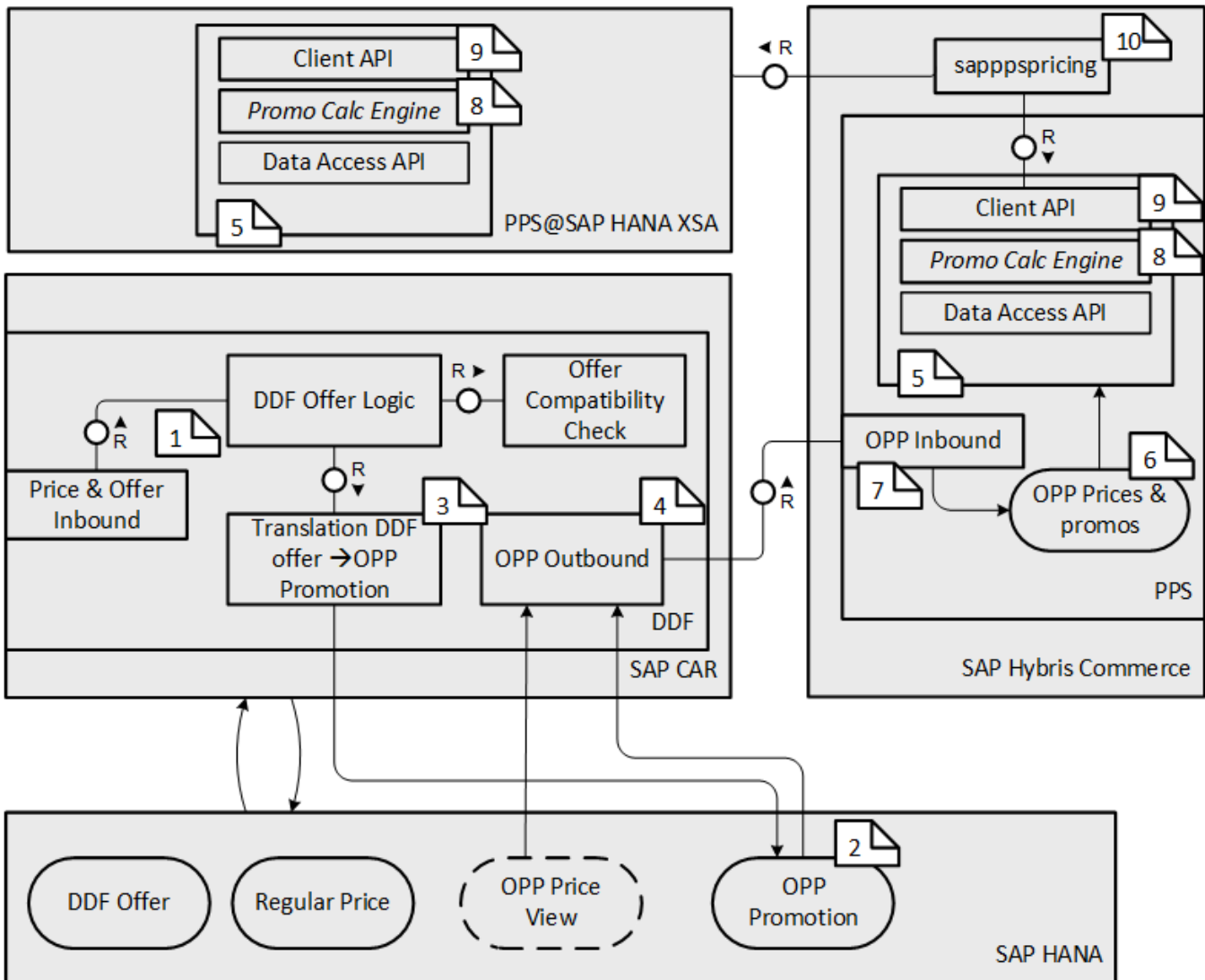
3. Method **IF_DRF_OUTBOUND-SEND_MESSAGE**

The data is mapped in this method from **MT_BASE_PRICE_IDOC** into the IDoc structure, and one or several IDocs are sent. The data is split into several IDocs according to the parameter **/ROP/PACK_SIZE_BULK**.

If performing the delta load, an additional DRF interface method **ANALYZE_CHANGES_BY_OTHERS** is called. The analysis of changes is based on the time stamp **LAST_SALES_PRICE_CHANGE** provided in table **/DMF/PRODLOC**. This is to select all relevant products and locations concerning all changed regular prices. The interface method also passes the relevant data to the methods above to map and build up the IDoc segments. Since the whole delta process is not based on change pointers, it is not possible to apply the manual time selection *Limit Changes Using Interval*.

OPP Extensibility

Modification-free extensibility is a major asset of SAP software. The following figure shows which parts of the overall OPP architecture are relevant for extensibility. Further details on how the parts can be extended will be explained in the following sections.



1. The processing of DDF offers and regular prices including the inbound processing and persistence
2. The data model of the OPP promotion in the price and promotion repository in SAP Customer Activity Repository (ABAP)
3. The logic used to process OPP promotions and regular prices, in particular how DDF offers are transformed into OPP promotions
4. The outbound processing of OPP promotions and regular prices
5. Extending the PPS: general concepts (Java)
6. The data model of the OPP promotion and regular prices in the promotion calculation engine (Java)
7. The inbound processing of OPP promotions and regular prices for a local deployment
8. The data model and processing logic of OPP promotions in the promotion calculation engine of the promotion pricing service
9. The data model of the price calculation requests against the PPS
10. The creation and consumption of price calculation requests for PPS clients, in particular of the **sappspricing** extension within SAP Commerce



This is applicable only for the local deployment of the PPS as part of SAP Industry Package for SAP for Retail.

Extensibility of Demand Data Foundation (DDF)

The existing concepts of SAP Customer Activity Repository are reused. There are no changes here. Mainly the following concepts are offered:

- Data dictionary structures offer Customizing includes that can be used on the customer side. If a table or structure in the standard system is enhanced with customer fields using a Customizing include, there is no need to modify the table and structure definitions of the standard system. The customer fields are automatically inserted in the new delivered table or structure definition during an upgrade. These enhancements cannot be lost during the upgrade. For more information about Customizing includes, see [Customizing Includes](#) in SAP Library on SAP Help Portal.
- Business Add-Ins (BADIs) are offered at various places, allowing the modification-free enhancement of business logic. For more information, see the application help for SAP Customer Activity Repository in SAP Library on SAP Help Portal.

Extensibility of DDF Offer Inbound API

The offer master data is used to plan promotions or to schedule demand modeling and forecasting processes. DDF uses an inbound interface to receive the offer master data through a Remote Function Call (RFC). For more information about this offer inbound API and the ways in which you can enhance this interface, see [Offer Master Data](#) in SAP Library on SAP Help Portal and read the documentation for enhancement spot **/DMF/CUSTOMER_EXT_OFFER** in the SAP Customer Activity Repository system.

Extensibility of DDF Regular Price Inbound API

The product location master data is used to determine the regular price for a product at a specific location in a given time frame. DDF uses an inbound interface to receive the product location master data with a Remote Function Call (RFC). For more information about this regular price inbound API and the ways in which you can enhance this interface, see [Product Location Master Data](#) in SAP Library on SAP Help Portal and read the documentation for enhancement spot **/DMF/CUSTOMER_EXT_PRODLOC** in the SAP Customer Activity Repository system.

Extensibility of the OPP Data Model (ABAP)

The concepts to enhance the OPP data model follow those of the Demand Data Foundation (DDF) offer. For example, in every database table the Data Dictionary offers Customizing includes. Since the structures on which the application logic is based refer to the structure definition of the database tables. Additional fields are immediately available within the business logic, for example, during the transformation of DDF offers into OPP promotions.

The following database tables are relevant for the OPP promotion model:

- **/ROP/PROMOTION**
Table for promotion-relevant header data. The Customizing include for this table is **CI_ROP_PROMOTION**.
- **/ROP/PROMO_RULE**
Table for promotion price derivation rules. The Customizing include for this table is **CI_ROP_PROMO_RULE**.
- **/ROP/ELIGIBILITY**
Table for all data that is relevant for the eligibilities of the OPP promotion. The Customizing include for this table is **CI_ROP_ELIGIBILITY**.
- **/ROP/PRICE_RULE**
Table for price derivation rules. The Customizing include for this table is **CI_ROP_PRICE_RULE**.
- **/ROP/MAM_ITEM**
Table for mix-and-match price derivation items. The Customizing include for this table is **CI_ROP_MAM_ITEM**.
- **/ROP/PROMO_BU**
Table for the business units for which the promotion is relevant. The Customizing include for this table is **CI_ROP_PROMO_BU**.
- **/ROP/PROMO_TEXT**
Table for the language-dependent texts of a promotion. The Customizing include for this table is **CI_ROP_PROMO_TEXT**.
- **/ROP/EX_ACT_PARM**
Table for the parameters of a price derivation rule of type *external action*. The Customizing include for this table is **CI_ROP_PROMO_EXT_ACTION_PARM**.
- **/ROP/EX_ACT_TEXT**
Table for the language-dependent texts of a price derivation rule of type *external action*. The Customizing include for this table is **CI_ROP_PROMO_EXT_ACTION_TEXT**.
- **/ROP/MERCH_SET** (available as of CAR 3.0 FP02)
Table for the promotion relevant merchandise. In this table the product groups used in the DDF offer are stored. The Customizing include for this table is **CI_ROP_MERCH_SET**.

Example:

This example applies only for OPP promotions not for regular prices. You can use the following steps to add a new field to an existing database table via a Customizing include:

1. To add the upselling code to the OPP promotion header table, go to transaction **SE11** and display the database table **/ROP/PROMOTION**.
2. Double-click **CI_ROP_PROMOTION** and create the structure of the data element.
3. Add the field **ZZUP_SELL_TCD** to the structure. In this example, data type **CHAR2** is used since the code should be a string with two characters.
4. Activate the structure. The new field is available in database table **/ROP/PROMOTION** and in the structures used by the business logic.



If you want to add fields to the SAP delivered types via CI includes, use the prefix **ZZ** for field names to avoid name collisions in future versions.

Extending SAP delivered ABAP domains

With the concept of domain appends, it is possible to extend the list of allowed values on customer side without any modifications. To avoid a collision between SAP delivered and your own defined domain values, it is strongly recommended that they are in the reserved customer namespace:

- Custom domain values with alphanumeric type definition (for example, **CHAR**): Z* or Y*
- Custom domain values with numeric or numerical text as type definition (for example **INT4** or **NUMC**): 9*

To find out if new domain values are required, or if an existing domain value is suitable to cover your specific use case, check the the *OPP Functional Guide for the Promotion Calculation Engine* and search for domains referenced by the OPP promotion data model.

If a value is part of the standard shipment of an ABAP domain, it is not necessarily used in the standard mapping of DDF offers into OPP promotions. There is no guaranteed support for this domain value within the PPS/PCE, for example:

- The value **UN** of the domain **/ROP/DB_MERCH_SET_OPERATION** is part of the standard shipment, but currently not used by the PPS (only **DF** is used) and therefore also not used in the standard mapping.

- The value **01** of the domain **/ROP/DB_DISC_METHOD** is part of the standard shipment, but not used in the standard mapping. However, the PCE supports this value as described in the *OPP Functional Guide for the Promotion Calculation Engine*.

Extensibility of the OPP Business Logic (ABAP)

The business logic on the ABAP side is based on the general SAP Enhancement Framework, which uses enhancement spots for customer-specific enhancements. This framework allows a modification-free adjustment of the application logic delivered by SAP. To keep the upgrade effort as low as possible, SAP recommends you apply the following options for objects in the **/ROP/** namespace:

1. Use the predefined enhancement spots via Business Add-Ins (BADIs). They can be used to accommodate most of your specific requirements that are not included in the standard delivery.



If a BAdI is missing, or the existing BAdI does not support your specific use case, you can address your issue [here](#).

SAP guarantees the upward compatibility of all BAdI interfaces. Release upgrades do not affect enhancement calls from within the standard software nor the validity of calling interfaces.

2. If this is not possible, use the applied factory/interface paradigm. Instances of SAP standard classes are centrally created in dedicated factory classes. The classes containing business logic expose it via interfaces. Consumers of the business logic refer only to the interfaces not to the concrete implementations. In addition, most classes are not final and have protected methods, allowing subclassing and overriding specific methods. Compared to direct source code enhancements, this option offers a better defined signature for the extension. Redefine the required factory classes to create subclasses of the SAP standard classes. You should be able to use this approach to cover the vast majority of your requirements.



Use this option only as long as no suitable BAdI is available to support your use case. SAP does not guarantee that classes or interfaces remain stable across releases.

3. Use implicit enhancement implementations, such as direct source code adjustments, only in very exceptional cases. These implementations have the big disadvantage that there is no defined interface on which you can rely. Source code enhancements may even refer to local variables. These enhancements should be needed only in order to adjust the SAP standard factory classes with your customer-specific factories, in order to realize option two.



SAP strongly recommends you do not use this option outside of factory classes since this is the least stable way to extend standard functions.

Example: Extending the OPP Business Logic (Options 2 and Option 3)

To change the transformation logic from offers into OPP promotions so that offers in the status *Recommended* are also to be considered (in the standard shipment only offers in the status *Approved* are to be considered), proceed as follows:

1. Create a new class, **ZZCL_CONFIG** as a subclass to **/ROP/CL_CONFIG**.
2. In the constructor, add the following lines (option 2):

```
METHOD constructor.
  super->constructor( ).
  APPEND /dmf/cl_offer_status=>recommended TO mt_relevant_status. " <- This is the actual enhancement
ENDMETHOD.
```

3. Create a new factory class, such as **ZZCL_COMMON_FAC**, as a subclass to **/ROP/CL_COMMON_FAC**, redefine method **/ROP/IF_COMMON_FAC~GET_CONFIG** as follows (option 2):

```
METHOD /rop/if_common_fac~get_config.
  IF mo_config IS INITIAL.
    mo_config = NEW zzcl_config( ). " <- New class!
  ENDIF.
  ro_config = mo_config.
ENDMETHOD.
```

4. In method **CLASS_CONSTRUCTOR** of class **/ROP/CL_COMMON_FAC**, make the following replacement (option 3):

```
METHOD class_constructor.
  g_factory_name = 'ZZCL_COMMON_FAC'. " <- Your factory class!
ENDMETHOD.
```

Extensibility of the Transformation from DDF Offer into OPP Promotion

As described in the section "Transformation from DDF Offers into OPP Promotions", the transformation logic is realized by calling a number of BAdIs contained in enhancement spot **/ROP/OFFER_MAPPING**. These BAdIs may have multiple implementations and the sequence in which the implementations are executed can be determined.

For more information, see the BAdI documentation for enhancement spot **/ROP/OFFER_MAPPING** in the system.

Example: Extending the Transformation from DDF Offer into OPP Promotion

A DDF offer has the field **ZZUP_SELL_TCD** with an entered value. The value has to be mapped from the offer field to the new field **ZZUP_SELL_TCD** in the promotion header. A new BAdI implementation of **/ROP/PROMO_BUILDER** needs to be created for this mapping. To create this BAdI implementation, proceed as follows:

1. In transaction **SE19**, create a new enhancement implementation for spot **/ROP/OFFER_MAPPING**, such as **Z_ROP_CUSTOMER_MAPPING_IMP**.
2. Create a new BAdI implementation for **/ROP/PROMO_BUILDER**.
3. Choose a *sequence number* greater than 0, for example 100. In this way, you can guarantee a post-execution of the new implementation.
4. Create a new class that implements the interface **/ROP/IF_PROMO_BUILDER**.
5. Implement the method **BUILD_PROMOTION** of the BAdI. This implementation performs the customer-specific mapping. The signature of the method provides all the information you need: **IS_OFFER_BO** (import parameter that includes all information about the offer) and **CS_PROMOTION_BO** (changing parameter that includes all the information about the mapped offer that is to be changed).
6. You can analyze and change the changing parameter accordingly. The following code snippet shows how the requirement mentioned above can be fulfilled:

```
METHOD /rop/if_promo_builder~build_promotion.
```

```
cs_promotion-zzup_sell_tcd = is_ofr_bo-zzup_sell_code.
```



It is not necessary to create your own implementation for the BAdIs **/ROP/OFFER_CLASSIFIER** and **/ROP/PROMO_RECIPES_BUILDER**.

Extensibility of the IDoc Outbound Processing (ABAP)

The promotions as well as the regular prices can be replicated to an external system using IDocs. The promotions are replicated using IDoc **/ROP/PROMOTION01** or **/ROP/PROMOTION02** (as of PPS 3.0), the regular prices are replicated using IDoc **/ROP/BASE_PRICE01**.

Both IDocs have dedicated extension segments to each IDoc subsegment. There are also several BAdIs to extend the logic that is used to transfer the price rules via IDocs.

The BAdIs for the outbound of the OPP promotions are contained in enhancement spot **/ROP/PROMO_OUTBOUND**.

- The BAdI **/ROP/IDOC_CONTROL** can be used to change the IDoc control record of the basis IDoc or an extended type of it.
- The BAdI **/ROP/IDOC_DATA** can be used to change the IDoc content of the basis IDoc or an extended type.
- The BAdI **/ROP/MAP_OUTBOUND_DATA** can be used to change the mapping process within the promotion outbound procedure.

The BAdIs for the outbound of the regular prices are contained in enhancement spot **/ROP/BASE_PRICE_OUTBOUND**.

- The BAdI **/ROP/BASE_PRICE_IDOC_CONTROL** can be used to change the IDoc control record of the basic IDoc or an extended type of it.
- The BAdI **/ROP/BASE_PRICE_IDOC_DATA** can be used to change the IDoc content of the basic IDoc or an extended type.

For more information, see the BAdI documentation for enhancement spots **/ROP/PROMO_OUTBOUND** and **/ROP/BASE_PRICE_OUTBOUND**.

Example: Extending the IDoc Outbound

To enhance the IDoc **/ROP/PROMOTION01** with the new field **ZZUP_SELL_TCD** in database table **/ROP/PROMOTION**, you need to create a new BAdI implementation of **/ROP/MAP_OUTBOUND_DATA**:

1. In transaction **SE19**, create a new enhancement implementation for spot **/ROP/PROMO_OUTBOUND**, such as **Z_ROP_CUSTOMER_OUTBOUND_IMP**.
2. Create a new BAdI implementation for **/ROP/MAP_OUTBOUND_DATA**.
3. Implement the method **MODIFY_MAPPING** of the BAdI. You can use this method to modify the mapping process when a promotion is mapped to the corresponding IDoc structure. The signature of the method provides all the information you need: **IS_PROMOTION** (import parameter that includes all information about the promotion) and **CT_PROMOTION_OUTBOUND_DATA** (changing parameter that represent the IDoc data structure that is to be changed)
4. The following code snippet shows how the extension segment could be mapped:

```
METHOD /rop/if_map_outbound_data~modify_mapping.
```

```

DATA: ls_idoc  TYPE edidd,
      ls_enhanc TYPE /rop/e1_promotion_enhanc.

READ TABLE ct_promotion_outbound_data ASSIGNING FIELD-SYMBOL(<fs_idoc>) WITH KEY segnam = '/ROP
/E1_PROMOTION'.
IF sy-subrc = 0.
  ls_enhanc-flldgrp = 'HEADER'.
  ls_enhanc-flldname = 'ZZUP_SELL_TCD'.
  ls_enhanc-flldval = is_promotion-zzup_sell_tcd.
ENDIF.
ls_idoc-segnam = '/ROP/E1_PROMOTION_ENH'.
ls_idoc-sdata = ls_enhanc.
INSERT ls_idoc INTO ct_promotion_outbound_data INDEX sy-tabix + 1.

```

5. As a result, the IDoc extension segment **/ROP/E1_PROMOTION_ENH** is filled as follows:

Field Name	Field Content
FLDGRP	HEADER
FLDNAME	ZZUP_SELL_TCD
FLDVAL	XY



As of SAP CAR 3.0 FP03, CI include fields of the OPP promotion tables are automatically mapped to the corresponding IDoc extension segments if the DRF outbound parameter **/ROP/GENERIC_ENH_MAP** is set to 'X'.

Extensibility of the OPP Data Model (Java)

The specific use case determines what you need to do when you extend the predelivered entities. The following cases are described below:

- An existing entity is to be enhanced for a field
- A completely separate new entity is to be introduced
- Attribute converters are to be added to existing fields
- A new entity is to be added as a child entity to an existing entity (reachable via a relation, such as the texts for a promotion)
- A new entity is to be added as a specialization of an existing entity (such as the different types of eligibilities)
- The existing logic for equals() and hashCode() of a JPA entity is to be changed

Adding a Field to an Entity

If the existing entity is a subclass of `com.sap.pengine.dataaccess.promotion.common.entities.AbstractEntityImpl`, you can add a field to an entity without any coding on the data access level. This is possible because OPP uses the concept of virtual access methods offered by EclipseLink. For more information, see the documentation on the Eclipse website.

The additional mapping information is stored in a separate object relational mapping file (orm), for example, **ppe-local-orm.xml**. This file must be located on the classpath. To make this file known to the PPS, add the following line to the **ppe-local.properties** file:

```

# Note the leading comma!!
sap.dataaccess-common.custmappingresources=,ppe-local-orm.xml

```



It is possible to add several orm files separated by a comma.

Using the example of the upsell type code of a promotion, the file **ppe-local-orm.xml** needs to contain the following:

Adding a single field to an existing entity

```

<entity-mappings
  xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.eclipse.org/eclipselink/xsds/persistence/orm
http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_4.xsd"
  version="2.4">

  <!-- ... -->

```

```

<entity
  class="com.sap.pengine.dataaccess.promotion.common.entities.PromotionImpl">
  <attributes>
    <basic name="ZZUPSELLING_CODE" attribute-type="String" access="VIRTUAL">
      <column name="ZZUP_SELL_TCD" />
      <access-methods get-method="get" set-method="set" />
    </basic>
  </attributes>
</entity>
</entity-mappings>

```

Assuming that the column **ZZUP_SELL_TCD** has been added to table **/ROP/PROMOTION** on the ABAP side, this is mapped to the new JPA entity attribute **ZZUPSELLING_CODE**. Its content is stored in map **extensions** inherited from **AbstractEntityImpl**. The access takes place via **get()** method and **set()** method. The **get()** and **set()** method are already part of the corresponding entity interface in PPS module **dataaccess-interface**.

Adding a Separate Entry

After you define the new entity, you have to proceed with the standard approach. The new entity is made visible to the entity manager factory (in other words it is added to the list of packages scanned by the entity manager factory for JPA entities or attribute converters) by adding its package name to the Spring property **sap.dataaccess-common.custpackagestoscan**. Assuming that the new entity is in packages **com.mycompany.myentities1** and **com.mycompany.myentities2**, the property must have the following value:

```

# ... you saw the leading comma...?
sap.dataaccess-common.custpackagestoscan=com.mycompany.myentities1,com.mycompany.myentities2

```

Adding an Attribute Converter to an Existing Attribute

If writing the attribute converter, only the converter is made visible to the entity manager factory by adding its package name to the Spring property **sap.dataaccess-common.custpackagestoscan**. The attribute converter is added to the JPA entity attribute using **ppe-local-orm.xml**, as shown in the following example for the OPP promotion:

Adding an attribute converter to an existing field

```

<entity-mappings
  xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.eclipse.org/eclipselink/xsds/persistence/orm
  http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_4.xsd"
  version="2.4">

  <!-- ... -->
  <entity
    class="com.sap.pengine.dataaccess.promotion.common.entities.PromotionImpl">
    <convert
      converter="com.mycompany.converters.MyNewConverter"
      attribute-name="someExistingAttribute"/>
    </entity>
  </entity-mappings>

```

Adding a Subentity to an Existing Entity

You can create the JPA entity in the way you create a separate new entity. You can provide a relation from the new entity to pre-delivered entities via the standard JPA way (**@OneToMany**). If the relation from the existing entity to the new subentity is required, the existing entity must be enhanced by this relation. In **ppe-local-orm.xml**, this is done as shown in the following example in which a new subentity is added to the OPP promotion. We assume that the new entity has SAP client as table column **MANDT** and the promotion ID as table column **PROMOTION_ID** as attributes.

Adding a relation from an existing to a new entity

```

<entity-mappings
  xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.eclipse.org/eclipselink/xsds/persistence/orm
  http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_4.xsd"

```

```

version="2.4">

    <!-- ... -->
<entity
  class="com.sap.ppengine.dataaccess.promotion.common.entities.PromotionImpl">
  <attributes>
    <one-to-many name="myOwnEntities" access="VIRTUAL" attribute-type="java.util.List"
      target-entity="com.mycompany.entities.MyOwnEntity">
      <join-column name="MANDT" referenced-column-name="MANDT" />
      <join-column name="PROMOTION_ID" referenced-column-name="PROMOTION_ID" />
    </one-to-many>
  </attributes>
</entity>

</entity-mappings>

```

Adding a Specialization to an Existing Entity

You can only add a specialization to an existing entry if the existing entity is prepared accordingly, as it is the case for the price derivation rule and the price derivation rule eligibility. Inheritance is represented in the database via a dedicated column holding the discriminator determining the specific type that is stored in the database table record. For the price derivation rule and the price derivation rule eligibility, this is the column **TYPE_CODE**. The new entity is to be defined as shown in the following example of a new eligibility type:

Adding another specializaton to an existing entity

```

package com.mycompany.entities;

@Entity
@DiscriminatorValue(value = "ABCD")
public class AbcdPriceDerivationRuleEligibilityImpl extends
    PriceDerivationRuleEligibilityImpl implements
    AbcdPriceDerivationRuleEligibility {

    // New fields may come here
}

```

In the example above, the new entity **AbcdPriceDerivationRuleEligibilityImpl** also implements a new interface **AbcdPriceDerivationRuleEligibility**. We recommend you extend the existing classes and interfaces to provide a clean interface to the promotion calculation engine:

- **PromotionService** (add new access methods)
- Optionally **NamedQueryService** (add new access methods)
This is needed if new search methods are required that use Spring caches. Then the method performing the to-be-cached access must be external to the calling method within **PromotionServiceImpl**.
- **PromotionServiceImpl** (also redefine the bean **sapPromotionService**)
- Optionally **NamedQueryServiceImpl** (also redefine **sapNamedQueryService**)

The new entity is made visible to the entity manager factory by adding its package name to the Spring property **sap.dataaccess-common.custpackagestoscan**.

Use virtual attributes only?

It is possible to define JPA entities using only virtual attributes. If the entity is a specialization of an existing SAP entity, this approach would make it unnecessary to define a new IDoc type since all fields go to the extension segments of existing segments. However, the use of virtual attributes is more resource intensive than the use of ordinary attributes of the corresponding Java class.

Therefore, we recommend you start with virtual attributes and switch to non-virtual attributes if resource consumption is noticeably higher.

Using Own Logic for Equals() and hashCode() of a JPA Entity

You can use your own logic for equals () and hashCode () by replacing the Spring bean **sapJpaEqualsHashCodeHelper**. For more information, see the description of PPS module **dataaccess-common**.

Extensibility of Client API (Java)

The extensibility of the client API is an easy and effective way to meet customer requirements. The underlying standard of the Association for Retail Technology Standards (ARTS) already offers a lot of functions. However, it does not provide an overall solution for customer-specific requirements. Therefore, customers might have to extend the data model. Some of the extensions will be part of the ARTS standard in later versions, others may be too customer-specific to be part of the ARTS standard.

An extension of the client API is not enough since the underlying promotion calculation engine also has to be extended to be able to process extension data provided by the client API.

There are two types of possible extensions for the client API:

Extensibility of Enumerations

All type code enumerations contain the values needed for the corresponding fields that are determined by ARTS. However, these fields are of type *String* in the Java classes and you can, therefore, add your custom values.

For any of the enumerations listed below, your custom values must match the following pattern (as defined in the XSD provided with the Client API): **[0-9A-Za-z][0-9A-Za-z]*:[0-9A-Za-z]***. If not, the following problems can occur:

- The value that you have added is the same as introduced later on in the standard delivery
- A future XSD validation will reject the request

Entity	Attribute/Element	Possible Value
ARTSCommonHeaderType	ActionCode	Any value from ActionCommonDataActionCodeEnumeration or any other string matching the pattern above
ARTSCommonHeaderType	MessageType	Any value from MessageTypeCodeEnumeration or any other string matching the pattern above
ResponseCommonData	ResponseCode	Any value from ResponseTypeCodeEnumeration or any other string matching the pattern above
BusinessErrorCommonData	Severity	Any value from SeverityCodeEnumeration or any other string matching the pattern above
BusinessUnitCommonData	TypeCode	Any value from BusinessUnitTypeCodeEnumeration or any other string matching the pattern above
PriceCalculateBase	TransactionType	Any value from TransactionTypeEnumeration or any other string matching the pattern above
LoyaltyRewardBase	TypeCode	Any value from LoyaltyRewardTypeCodeEnumeration or any other string matching the pattern above
PointsCommonData	Type	Any value from PointsTypeCodeEnumeration or any other string matching the pattern above
PriceDerivationRuleBase	ApplicationType	Any value from PriceDerivationApplicationTypeCodeEnumeration or any other string matching the pattern above
PriceDerivationRuleEligibility	Type	Any value from DerivationRuleEligibilityTypeEnumeration or any other string matching the pattern above
ItemBase	ItemType	Any value from RetailTransactionItemTypeEnumeration or any other string matching the pattern above
RetailPriceModifierBase	Amount	Any value from RetailPriceModifierAmountActionEnumeration or any other string matching the pattern above
RetailPriceModifierBase	Percent	Any value from RetailPriceModifierPercentActionEnumeration or any other string matching the pattern above
AmountCommonData	Currency	Any value from CurrencyTypeCodeEnumeration or any other string matching the pattern above
RoundingRuleType	RoundingMethod	Any value from RoundingMethodEnumeration or any other string matching the pattern above
CalculationModeTypeCode	CalculationMode	Any value from CalculationModeEnumeration or any other string matching the pattern above

Extensibility of Content with User-Defined Attributes / Elements

Well-defined points in the ARTS data model, so-called *any attributes/elements* are provided. These attributes allow the extension of the client API with anything a customer wants to add.

OPP only supports *any elements* because of problems with the Jackson XML/JSON parser. The following entities contain these extension points:

Entity	Object Type

LineItemChoiceDomainSpecific	Object
SaleBase	List<Object>
SaleForDeliveryBase	List<Object>
SaleForPickupBase	List<Object>
ReturnBase	List<Object>
ReturnForDeliveryBase	List<Object>
ReturnForPickupBase	List<Object>
CustomerOrderForDeliveryBase	List<Object>
CustomerOrderForPickupBase	List<Object>
ItemDomainSpecific	List<Object>
PriceDerivationRuleBase	List<Object>
PriceDerivationRuleEligibility	List<Object>
RetailPriceModifierDomainSpecific	List<Object>
DiscountBase	List<Object>
TenderCouponBase	List<Object>
ARTSCommonHeaderType	List<Object>
ExternalActionType	List<Object>
LoyaltyAccountType	List<Object>
LoyaltyRewardBase	List<Object>
PriceCalculate	List<Object>
PriceCalculateBase	List<Object>
PriceCalculateResponse	List<Object>
PromotionExternalTriggerType	List<Object>
PromotionManualTriggerType	List<Object>
PromotionPriceDerivationRuleReferenceType	List<Object>
RoundingRuleType	List<Object>
ShoppingBasketBase	List<Object>

More information about the general ARTS extension concept of the XML schemas can be found [here](#).

Restrictions

It is not possible to use XML attributes within any elements, for example:

Restrictions

```
<any>
  <SimpleExtension myAttribute='hello'>MyExtension</SimpleExtension>
</any>
```

Instead, you could use the following attributes:

Alternative

```
<any>
  <SimpleExtension>
    <myAttribute>hello</myAttribute>
    <data>MyExtension</data>
  </SimpleExtension>
```

```
</any>
```

Example: Enrich SaleForDelivery Entity with Address Information

A customer wants to enrich the *SaleForDelivery* entity with address information.

Request excerpt

```
...
  <ShoppingBasket>
    <LineItem>
      <SequenceNumber>0</SequenceNumber>
      <MerchandiseHierarchy ID="ID1" >hier1</MerchandiseHierarchy>
      <SaleForDelivery ItemType="Stock" NonDiscountableFlag="false" FixedPriceFlag="false">
        <TaxIncludedInPriceFlag>false</TaxIncludedInPriceFlag>
        <NonPieceGoodFlag>false</NonPieceGoodFlag>
        <FrequentShopperPointsEligibilityFlag>false</FrequentShopperPointsEligibilityFlag>
        <DiscountTypeCode>2</DiscountTypeCode>
        <PriceTypeCode>00</PriceTypeCode>
        <NotConsideredByPriceEngineFlag>false</NotConsideredByPriceEngineFlag>
        <ItemID>CHA2111012</ItemID>
        <Quantity Units="1" UnitOfMeasureCode="PCE">5</Quantity>
        <any>
          <Street>Neue Bahnhofstrasse 21</Street>
          <City>Sankt Ingbert</City>
          <PostalCode>66386</PostalCode>
          <Country>Deutschland</Country>
        </any>
      <any>
        <Street>Dietmar-Hopp-Allee 16</Street>
        <City>Walldorf</City>
        <PostalCode>69160</PostalCode>
        <Country>Deutschland</Country>
      </any>
    </SaleForDelivery>
  </LineItem>
...

```

This example shows that the line item has been enriched with two addresses.

To access this information from Java, you can use the following code snippet as reference:

Access any information in Java

```
final List<Object> anyList = priceCalculate.getPriceCalculateBody().get(0).getShoppingBasket().getLineItem().get(0).getSaleForDelivery().getAny();

for (int i = 0; i < anyList.size(); i++)
{
    //Do whatever you want with the address information
}

```

Extending the PPS Business Logic (Java)

 This chapter is relevant as of PPS version 3.0.

In order to extend business logic on the customer side, it is crucial that the extended application offers an well-defined API that:

- Has a clearly defined facade
- Calls the extension at a defined point during the application logic
- Is well documented
- Is stable across releases
- Does not require modification of the delivered code
- Is easy to consume

In addition, it is necessary to write the customer extension in such a way that it is independent of its later runtime environment. In particular, this is relevant for extensions of the promotion calculation engine, which can be used within the PPS as well as within a GK OmniPOS deployment. This implies the following:

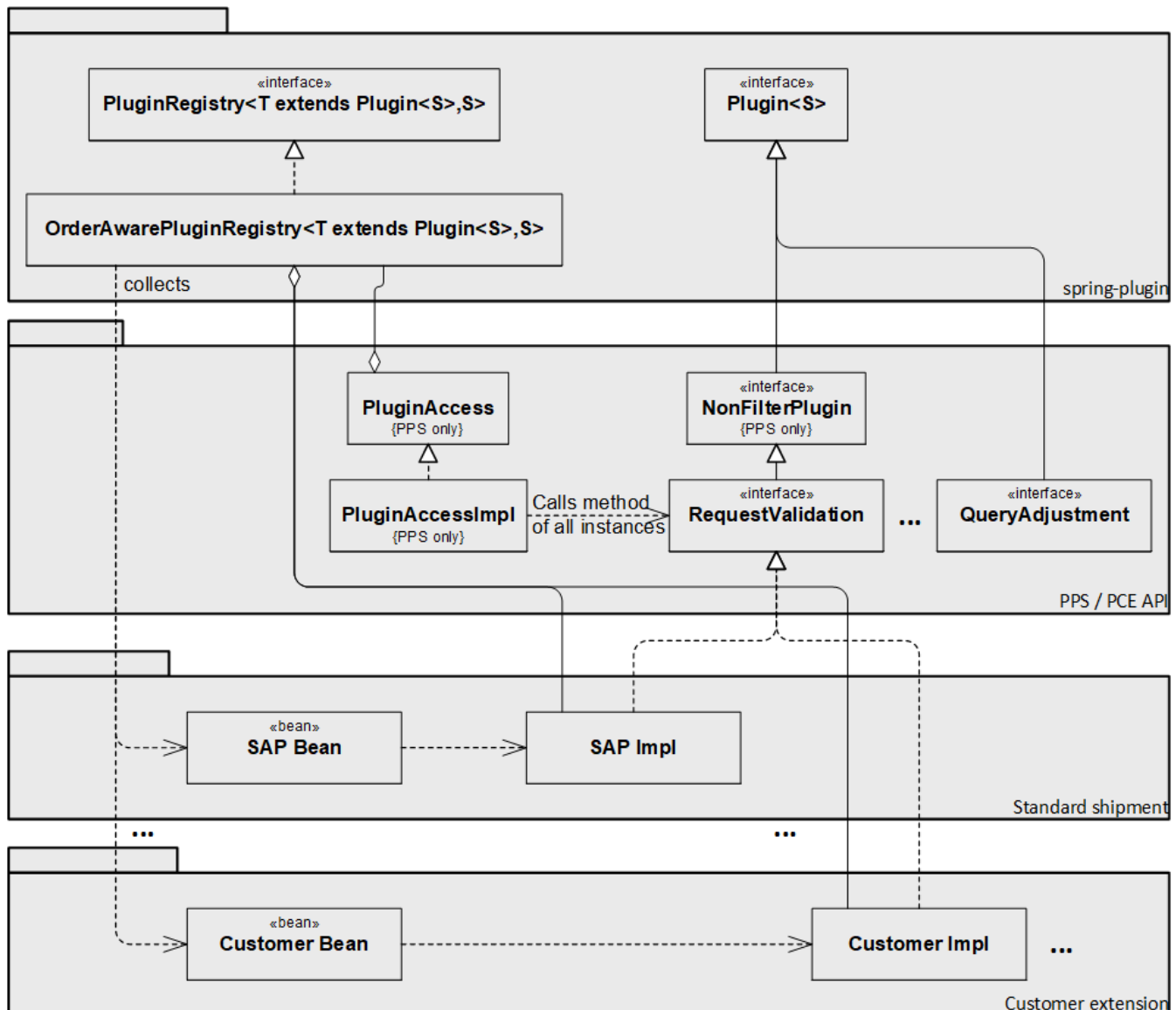
- There must be one file structure of the Java project containing the customer extension
- It must be possible to distribute and install the built customer artifact independently of the standard artifacts

For this purpose, the PPS and the contained promotion calculation engine (PCE) offer the following:

- A plugin concept to allow customer extensions of the standard business logic
- The guaranteed stability of certain artifacts

Plugin Concept

This is similar to the concept of the Business Add-Ins ("BAI") offered by ABAP: The following figure illustrates its components and how it works:



The plugin concept is based on the Spring plugin framework (see <https://github.com/spring-projects/spring-plugin>). The Spring plugin framework offers the interface `org.springframework.plugin.core.Plugin<S>`. This is the parent interface for all application-specific interfaces that provide extension hooks.

The **Plugin** interface offers the type parameter <S>, which allows an implementation of this interface to tell for in which context the corresponding implementation shall be used. This is realized via the **supports()** method, allowing a caller to filter implementations by a specific criterion. How to implement this is described below.

The PPS/PCE now offers interfaces extending **Plugin**, offering specific methods. These interfaces are called **Plugin interfaces**. Plugin interfaces offer the extensibility for a certain aspect of the application logic. The meaning of the type parameter <S>, i.e. the filter criterion, depends on the individual plugin interface. The interface **NonFilterPlugin** is a special case, serving as the parent for all Plugin interfaces where a filter on implementation is not feasible or required. In the diagram above, two example plugin interfaces are shown:

- Interface **RequestValidation** which enables the addition of further validations of the incoming Price Calculation request. This does not offer the selection of individual plugin implementations based on a filter value. Therefore it extends **NonFilterPlugin**.
- Interface **QueryAdjustment** which allows changing query parameters of JPA NamedQueries, setting query hints etc. This works per query to be executed. Hence, the query name (of type **String**) is a filter criterion. In the **supports()** method, an implementation of this interface would compare the provided query name with the query name this implementation is intended for.

The implementation of a Plugin Interface is called **Plugin Implementation**. This consists of two parts:

- The Java class implementing the Plugin Interface
- The Spring Bean adding an instance of the Java class to the Spring Application Context.

Having the Plugin Implementations created, they must be somehow collected so that within the application all implementations of a plugin interface can be called. This is done by the **Plugin Registry**. During startup of the Spring Application Context, for a given Plugin Interface it looks for all Spring Beans implementing this interface. **No static wiring of the implementations to the Plugin Registry is needed.** When calling the Plugin Implementations, the Plugin Registry offers the list of references to the corresponding Spring Beans. The Plugin Registry itself is an ordinary Spring Bean as well. However, the Spring Plugin framework adds another XML namespace to the Spring XML file, making it easier to define the registry. As a second possibility, the plugin implementations can be collected by the registry which itself is not exposed - instead the collected implementations are exposed as a simple list which can be injected into the Spring bean calling the plugin.

The following example shows how a Plugin Registry and a Plugin Implementation is created. The Plugin Registry is added in a Spring XML of the SAP delivered artifacts.

Defining a Plugin Registry in SAP Spring XML

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

    <!-- Option 1: Plugin registry for adjustment of named queries -->
    <alias name="sapDefaultQueryAdjustmentPluginRegistry" alias="sapQueryAdjustmentPluginRegistry" />
    <plugin:registry id="sapDefaultQueryAdjustmentPluginRegistry"
        class="com.sap.ppengine.api.plugin.QueryAdjustment" />

    <!-- Option 2: Plugin collect plugin implementations as list -->
    <alias name="queryAdjustments" alias="queryAdjustmentImplsAsList" />
    <plugin:list id="queryAdjustmentImplsAsList"
        class="com.sap.ppengine.api.plugin.QueryAdjustment" />

</beans>
```

The non-PCE part of the PPS uses the registry approach, the PCE part uses the list approach. From an implementor's view this does not make a difference though.

Calling the Plugins

PPS Only

This section is only valid for the PPS and does not apply for the promotion calculation engine (PCE) that is part of it.

Calling the Plugins - to be more precise: the relevant implementations of the corresponding Plugin Interfaces - is done via the helper class **PluginAccess**. For a given Plugin Interface and Plugin Registry, it allows a simple invocation of the desired interface method for all relevant implementations. Some examples are shown below:

Calling Plugins via CallPlugins class

```
// Injected via Spring
PluginAccess pluginAccess;

// Call the validate() method of all implementations for plugin RequestValidation expecting a checked exception
pluginAccess.callAll(ContextEnrichment.class, p -> p.enrichContext(getContext(), priceCalculate));

// Call the validate() method of all implementations for plugin RequestValidation expecting a checked exception
pluginAccess.callAllChecked(RequestValidation.class, p -> p.validate(priceCalculate));

// Call the single implementation of a method with return parameter
Class<T> clazz = pluginAccess.callFunction(CustomEligibility.class, eliType, p -> p.classForType());
```

Implementing a Plugin

A corresponding implementation on customer side is just a regular Spring Bean to be added to the Spring XML:

Defining a Plugin Implementation in Customer Spring XML

```
<bean id="myQueryAdjustment" class="com.customer.MyQueryAdjustmentImpl"/>
```



Currently, only Spring beans with scope "singleton" (which is the default scope) are supported for plugin implementations.

The corresponding Java class implements the Plugin Interface. In this example we want to adjust the query "findItemEligibilityIDsByItemID" after any potential SAP implementation.

Customer class implementing a Plugin Interface

```
package com.customer;

import org.springframework.core.annotation.Order;
import com.sap.ppengine.client.dto.PriceCalculate;
import com.sap.ppengine.api.plugin.QueryAdjustment;
import com.sap.ppengine.client.impl.RequestValidationException;

// Note that order -10000000 to 10000000 is reserved for SAP
// ... but only multiples of 100
@Order(value = 10000001)
public class MyQueryAdjustmentImpl implements QueryAdjustment {

    @Override
    public boolean supports(final String queryName) {
        return "findItemEligibilityIDsByItemID".equals(queryName);
    }

    @Override
    public void adjustQuery(final Query query, final Context context) {
        // Do something
    }

    void adjustResult(final Query query, final Context context, final Object result) {
        // Do something else
    }
}
```

Note that the implementation of a Plugin Interface is not only possible on customer side but done on SAP side as well. For example, the standard request validation is an implementation of this Plugin Interface:

```
<alias name="sapDefaultCalculateRequestValidation" alias="sapCalculateRequestValidation" />
<bean id="sapDefaultCalculateRequestValidation" class="com.sap.ppengine.client.impl.RequestValidation30Impl">
  <property name="objectFactory" ref="sapClientApiDtoFactory" />
  <property name="maxNumberOfLineItems" value="{sap.client-impl.maxnumberoflineitems}" />
  <property name="requestHelper" ref="sapClientApiHelper" />
</bean>
```

This raises the question in which sequence the implementations are called. The used Plugin Registry supports the sorting of Plugin Implementations either via Java interface **org.springframework.core.Ordered** or via annotation **org.springframework.core.annotation.Order**. The specified integer value determines the sort sequence of the implementations - negative values are allowed. To avoid collisions, it is strongly recommended to use separate order values for each implementation of a certain Plugin Interface.



Reserved order values

Order values being multiples of 100 are reserved for the Plugin Implementations of the standard shipment. If you want to have your Plugin Implementation executed between delivered implementations, use a value which is not a multiple of 100. In case you want to make sure your implementation is executed before or after any current or future implementation, use order values below -10,000,000 or above 10,000,000.

If you want to replace an SAP implementation of a Plugin Interface, this is also possible using the PPS Module concept. In this case, define your Spring bean with the same ID (and not just alias) as the SAP standard bean:

Replacing an SAP standard Plugin Implementation

```
<alias name="sapDefaultCalculateRequestValidation" alias="sapCalculateRequestValidation" />
<bean id="sapDefaultCalculateRequestValidation" class="com.customer.MyReplacingValidationImpl" />
```

To make sure this bean is taken instead of the SAP standard bean, your PPS module must depend on the PPS module where the SAP standard bean was defined. In the example above this would be the **client-impl** module. This is specified in the `(..)-ppe-module-metadata.xml` file of your module:

Defining the dependency to the PPS module of the bean to be replaced

```
<module xmlns="http://www.sap.com/ppengine/core/module"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.sap.com/ppengine/core/module ppengine-module-0.2.xsd">
  <name name="custextension" vendor="customer" />
  <dependencies>
    <module name="client-impl" vendor="sap"/>
  </dependencies>
</module>
```

If you just define an additional Plugin Implementation, declaring such a dependency is not required.



Stability of SAP Spring Bean IDs

Although SAP tries to ensure compatibility, there is no guarantee that for a provided Plugin Interface the number of standard implementations, the contained logic, the execution sequence or the IDs remain stable.

In order to easily find the list of Plugin Interfaces offered for the PPS and the PCE, they are bundled at central places:

- For the PPS this is the PPS module **api**, which exists as of PPS 3.0
- For the PCE, this is module **pricing-engine-api** (which is not a PPS module).

Due to constraints of the usage of common Java classes, plugins are identified differently for the PCE and non-PCE part of the PPS. The following table illustrates differences and similarities of the plugin definitions:

Aspect	Non-PCE	PCE
Inherits from	<code>org.springframework.plugin.core.Plugin<S></code>	
Annotated with	<code>@ExtensionStable</code>	<code>@ExtensionPoint(ExtensionType.Plugin)</code>

Located in package `com.sap.ppengine.api.plugin` | `com.gk_software.pricing_engine.api.*`

Further details of the offered Plugin Interfaces are documented in the corresponding chapter of these modules.

- Often the PPS context is provided as a method parameter. This is done on purpose - you are encouraged to store data you need throughout the application within a parameter of the PPS context. Note that this parameter can have any type - it is neither needed nor good practice to define a new PPS context parameter for every piece of information you want to store.

Guaranteed Stability

As of PPS 3.0, dedicated objects of the standard shipment offer a guaranteed stability for future releases.

What does that mean? When referring to the stability of an artifact two degrees of stability must be distinguished:

- The stability of an object towards the external callers / users of this object, in the following called **Consumer stability**. It is guaranteed that the usage of the artifact does not lead to compile errors in future releases. Example: The artifact is a Java interface, the caller invokes a method of this interface. The addition or removal of a method parameter would violate the caller stability constraint for that interface. On the other hand, replacing the type of a method parameter with a super type would not violate the caller stability.
- The stability of an object towards extenders of this object, in the following called **Extension stability**. This only applies for Java interfaces and classes. It is guaranteed that an implementor or an interface or a subclass of a class will not have compile errors in future releases. Example: The artifact is a Java class which has been extended. The extension uses a protected method. The removal of this method or changing its signature would violate the extension stability constraint for the extended class. On the other hand, adding a method to a Java interface using the Java 8 concept of (empty) default implementations is considered as uncritical, even if this leads to name collisions with customer implementations.

- To be on the safe side and avoid name collisions, it is recommended to prefix customer specific methods and attributes, e.g. "zz" or <customerName>.

- Extension stability includes consumer stability since an extension can always act like an external caller.

Unforeseen requirements to change something may come, and this is also true for the guaranteed stability. It may turn out that an artifact declared as (consumer or extension) stable must be changed in an incompatible way. In this situation, the following happens:

- In release X, it is announced that a certain incompatible change is required. The change itself is not yet done though.
- The change is also not done in releases X+1, X+2, X+3, X+4. However, during that time an alternative to the incompatible change will be offered (e.g. a method to be called instead if the original method will be removed).
- In release X+5 the incompatible change is performed.

- In this context a "release" means a PPS release (i.e. the version number of the shipped JAR files), not the release of the software component used to ship the PPS. Example: PPS 2.0.3 is contained in software component version **XSAC_OPP_PPS** 1.1.2. A new PPS release means that either the major or minor release number changes.

Example:

- Release number changes from 3.0.0 to 3.0.1: patch number changes no change of PPS release
- Release number changes from 3.0.1 to 3.1.0: minor version number changes new PPS release
- Release number changes from 3.1.0 to 4.0.0: major version number changes new PPS release

Documented Stability

- This chapter does not apply for the promotion calculation engine (PCE) of the PPS. For further information about the PCE, please consult the SDK of the Promotion Calculation Engine (chapter "PCE Extensions").

The guaranteed stability applies for the following PPS modules:

- dataaccess-interface**: All contained artifacts are guaranteed to be consumer stable. Incompatible changes will be documented via JavaDoc.
- client-interface**: All contained artifacts are guaranteed to be consumer stable. Incompatible changes will not happen. Changes to the expected way of using the client API will lead to new interface versions.
- api**: The degree of stability (caller / extension) is documented via the Java annotations **com.sap.ppengine.api.ConsumerStable** and **com.sap.ppengine.api.ExtensionStable**. Incompatible changes are documented via JavaDoc and Java annotation **com.sap.ppengine.api.PlannedIncompatibleChange**.

- In addition, PPS-specific DB tables have guaranteed extension stability in the sense that the DB key will not change and no delivered fields will be removed in any future release.

Your Choices for Extending the PPS Java Side

With the introduction of the Plugin Interfaces, you have three options for extending the PPS business logic. SAP recommends to use them in the following order of preference:



1. Implement the offered Plugin Interfaces. These are guaranteed to keep stable.
2. Use the PPS Module concept to replace SAP provided beans using a subclass of the Java class offered by SAP. Try to use as few of the protected methods of the super class as possible in order to reduce the risk of an incompatible change.
3. Use Spring AOP if option 2 is not feasible or requires the redefinition of too many beans. Depending on the kind of change, this approach may be very robust (e.g. if you want to grab the methods of the call of a ConsumerStable interface) or very risky. A general recommendation cannot be given here.

SAP Delivered Plugin Implementations

The following tables contains the Plugin Implementations of PPS Plugin Interfaces (excluding the PCE) which are part of the SAP standard shipment. Implementations of the PCE Plugin Interfaces can be found in the SDK of the PCE.

Plugin Interface	Plugin Implementation Class	Order	Plugin Implementation Bean	Description
ContextEnrichment	ContextFromRequestEnrichmentImpl	0	sapDefaultContextFromRequestEnrichment	Enrich PPS context with BU Type etc
FeatureCheck	FeatureCheckImpl	0	sapDefaultFeatureCheck	Checks PCE features from config stored in PPS context
PromotionServiceInitialization	ItemPriceDerivationRuleEligibilityCacheAwareBulkAccessorImpl	0	sapDefaultItemEligibilityBulkAccessor	Bulk access for Item Eligibilities
	MerchandiseSetEligibilityCacheAwareBulkAccessorImpl	1000	sapDefaultMSetEligibilityBulkAccessor	Bulkd access for MerchandiseSet Eligibilities
RequestAdjustment	AddBasePricesToRequestImpl	0	sapDefaultAddBasePricesToRequest	Read needed regular prices and write them into request forwarded to PCE
RequestValidation	RequestValidationImpl	0	sapDefaultCalculateRequestValidation	SAP standard consistency checks of request
Validation	BeanValidationWrapper	0	sapBeanValidatorWrapper	Available as of PPS 4.0. Wrapper for calling bean validation
	CheckPromotionForCycles	1000	sapPromoCyclesChecker	Checks for cycles in promotional subentities.
NonUniqueBasePriceHandling	NonUniqueBasePriceHandlingDefaultImpl	n/a	sapDefaultNonUniqueBasePriceHandling	Raises an exception if strategy SAP00 is configured in the PPS configuration.

Structure of Your Extension Project

If you create an extension of the PPS, it may be the case that this extension shall also be used in a GK OmniPOS solution (at least the PCE extension part of it). The extension concepts of the PPS and the PCE (in a non PPS context) are slightly different (see below). This needs to be kept in mind when creating the extension. This chapter describes how to set up the file and folder structure of a Java project which compiles to one JAR which can be used by the PPS as well as by the PCE in an OmniPOS context. This documentation assumes that you use Eclipse. In the simplest setup (shown below), the needed dependencies for compiling your extensions could be placed into the lib folder a separate Java project with a build path dependency set. However, it is recommended to use a build management tool such as Apache Maven in order to have a cleaner project setup. How to set up Maven dependencies to the provided JARs is described in the extensibility example "Promotions on Brand Level". Regardless of how the dependencies are resolved, the basic structure of the source (or resource folders) of an extension project remains the same.

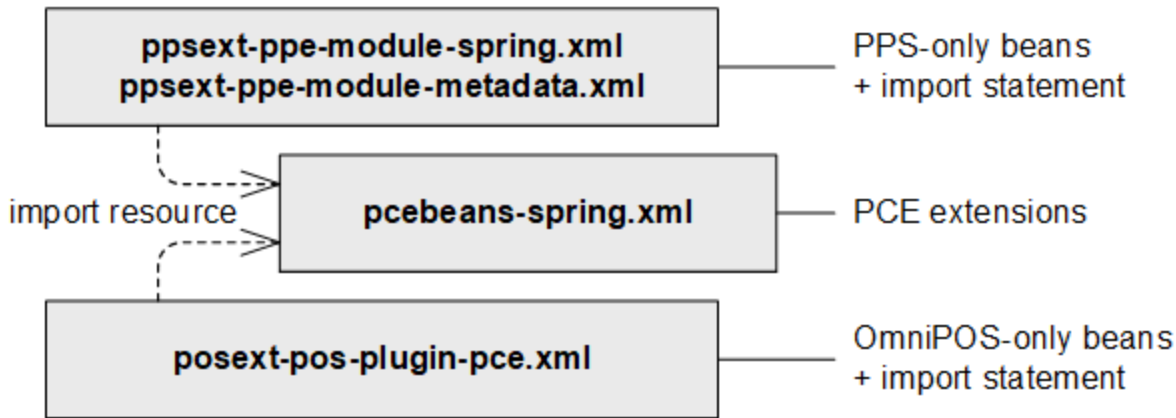
First, you need to categorize your extension objects into the following categories:

- Objects that shall only be used in a PPS context. As an example, extensions of the data access layer (incl. changes to the DB table definition) of the PPS or of the mapping between SAP data access interfaces and PCE objects fall into that category.
- Objects that shall be used in both a PPS and an OmniPOS context. These are extensions of the PCE itself.
- Objects that shall only be used in an OmniPOS context.

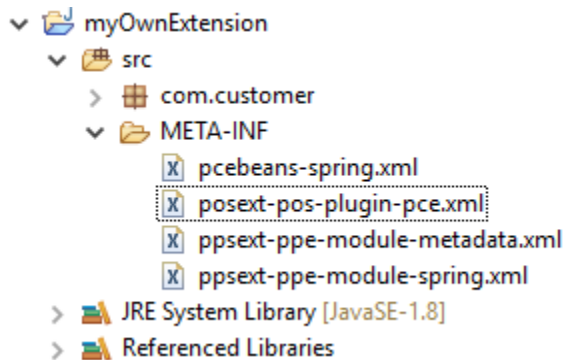
In all cases, the extension of the business logic is done by defining customer specific Spring Beans which are searched for in dedicated XML files when setting up the Spring Application Context.

- For the PPS, the XML files must match the following pattern (as defined in the **core** PPS module) in Spring resource syntax:
 - PPS Module metadata are located in `classpath*:META-INF/**/*-ppe-module-metadata.xml`
 - PPS Spring Beans are located in `classpath*:META-INF/**/*-ppe-module-spring.xml`
 - Metadata and Bean definition files are located in the same folder.
- For the PCE in an OmniPOS environment, further beans are searched for in `classpath*:META-INF/**/*-pos-plugin-pce.xml`

This leads to the following recommended file and folder structure, using some speaking prefixes. Note that this introduces the PPS module **ppsex**.



... which looks as follows in the IDE:



In order to make sure that your extension runs both in the PPS and in the OmniPOS environment, you also have to consider the version of the Java Runtime Environment. The PPS runs on Java 8 or later. However, some versions of GK OmniPOS still run on Java 6. If this is the case for you, you have to compile your extension for a Java 6 target runtime. If you are in doubt about the used Java runtime, please contact your GK Software contact person.



If you do not intend to extend the OmniPOS based PCE you can simply follow the PPS module concept and directly add the PCE extensions to the Spring XML of the PPS module.



In some cases it may be required to split up your PPS extension into several parts, i.e. several PPS Modules. This is the case when the PPS Spring Application Context is a real hierarchy as it is the case for the local PPS within SAP Commerce. Do not introduce dependencies to PPS modules which are not visible in the Spring Application Context, to which your PPS Module is added. In the example of the local PPS within SAP Commerce, an extension to the **idocinbound** module must be loaded into the Web Application Context as well.

Installing your Extensions

How to install your extensions depends on the hosting application and is described in the corresponding documentation. The common idea is the following:

1. Build the extension JAR once.
2. Add the extension JAR to the classpath of the hosting application.

For the central XSA based PPS, this is described in chapter *Integrating Custom Extensions into the XSA Based PPS*. For the local PPS in SAP Commerce, this is described in the Administration Guide of *SAP Commerce, integration package for SAP for Retail* under *Omnichannel Promotion Pricing*.

Extensibility of the Promotion Calculation Engine (Java)

The extensibility of the promotion calculation engine is described in the *SDK of the Promotion Calculation Engine* that can be found on the product page of SAP Customer Activity Repository.

Extensibility of the sappspricing PPS Integration (Java)

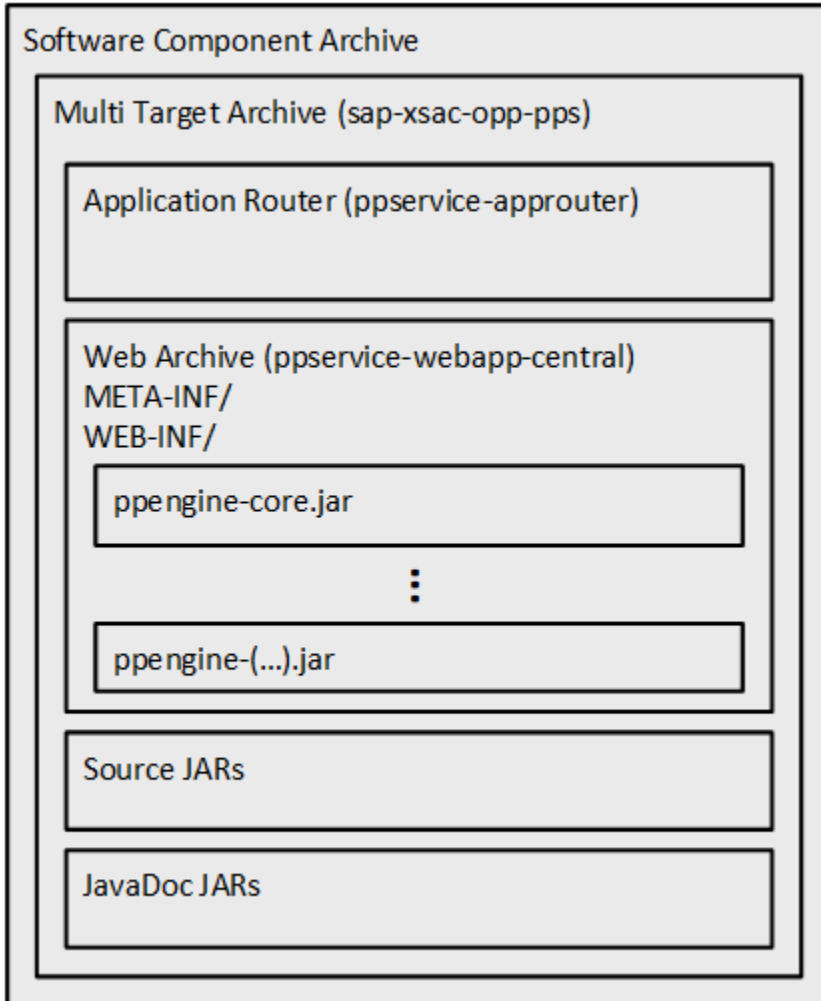
The extensibility of the sappspricing PPS integration is described in the Administrator Guide that can be found on the product page of SAP Hybris Commerce, integration package for SAP for Retail.

Extensibility Examples

The collective SAP note **2542001** contains references to examples showing how the extensibility concept can be used to implement certain requirements. It is planned to add further examples over time.

Integrating Custom Extensions into the XSA-Based-PPS

The only requirement for the use of a PPS module is that it is located on the classpath. In this case, the PPS Spring application context finds the module automatically and loads the contained Spring beans. The XSA-based PPS is shipped as follows:



When you install the Software Component Archive on XSA, the Multi Target Archive is deployed. This archive contains an application router and the Web application itself (provided as a Web archive). The Web application consists mainly of Java archives containing the actual business logic.

A JAR inside the **WEB-INF/lib** folder of the Web application looks like the obvious place for custom logic extending the PPS. However, deploying such an extended PPS comprises the creation of a new Web archive replacing the SAP standard archive and the creation of a new Multi Target Archive replacing the SAP standard shipment, which is not recommended. SAP is working on a clean way to add custom logic to a Web application without breaking its integrity. This chapter describes only the currently recommended way of adding further modules to the PPS shipped by SAP.



SAP does not require the use of a specific development environment (Build Tool, Source Code Management, Editor, and so on). However, in order to be able to provide a concrete example Eclipse is used as IDE in the following description. You can use a more advanced setup, including the use of Maven and GIT, for example.

Setting Up the Development Environment

1. Extract the Multi Target Archive from the Software Component Archive shipped by SAP.
2. Extract the following from the Multi Target Archive:
 - The Web archive (ppservice-webapp-central)
 - The source JARs
 - The Javadoc JARs
3. Extract the content of the **WEB-INF/lib** folder of the Web archive.

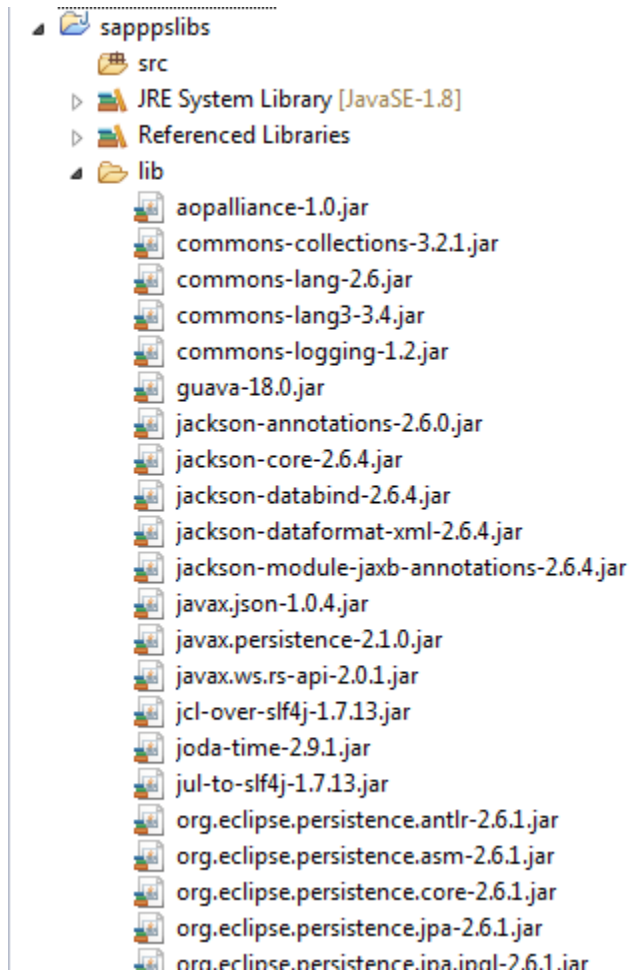
4. In your Eclipse workspace, create a new Java project, such as **sappslibs**, create a **lib/** folder and include all JARs of the **WEB-INF/lib** folder of the Web application. Add the JARs to the build path of the Java project.



This Java project is used only to compile and (unit-)test your extensions.

5. Add the following JARs to the **lib/** folder. These JARs will be provided by the tomcat runtime container:
 - **slf4j-api 1.7.13** or higher (see <http://slf4j.org/download.html>)
6. Ensure that all JARs of the **lib** folder are exported to the build path.
7. Create the folder **sourcejars** within **sappslibs**, in which you move all source JARs of the multi target archive.

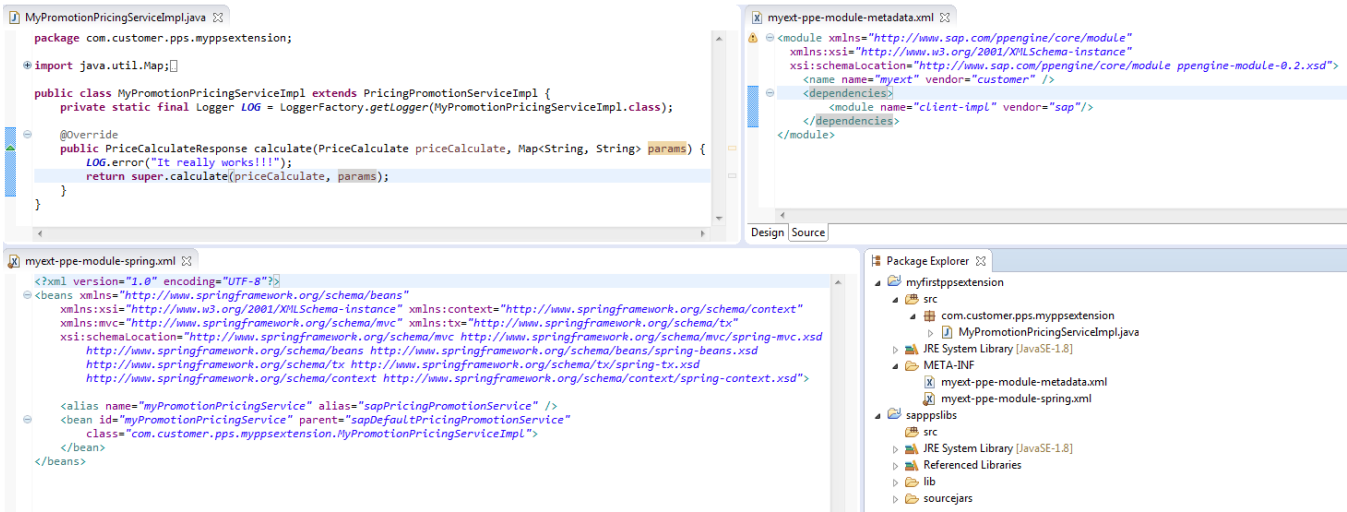
As a result, your project should look as follows. Note that the list of JARs is not complete.



Creating Your Extension Projects

1. In the same Eclipse workspace, create your custom extension as a Java project.
 - In this example, one project contains one PPS module. However, it is also possible to have a 1:n relationship between projects and PPS modules.
2. Add the project **sappslibs** to the build path of your Java project.
3. Define your PPS module metadata and spring beans via the corresponding XML files in the **META-INF** folder.
4. Create your Java classes for extending the standard functions.
5. Build the JAR file.

As a result, your Eclipse project could look as follows:



Adding Your Extension to the PPS

Once the JAR with your custom logic has been created, it needs to be placed on the classpath of the SAP standard PPS as follows:

1. Create a directory that is accessible by the XSA runtime. Restrict the access rights of that directory so that only trusted people are allowed to access it. If you are unsure how to create this directory, contact your system administrator. In this procedure, the path to this directory is `/usr/sap/hana/shared/XSA/customjars`.
2. Copy the JAR file into this directory and set the access rights accordingly.
3. Create an **MTAEXT** (Multi Target Archive Extension) file, for example `myPPS.mtaext`, with the following content:

```
_schema-version: "2.0.0"
ID: com.customer.retail.ppservice.XSAC_OPP_PPS
extends: com.sap.retail.ppservice.XSAC_OPP_PPS
modules:
# some lines omitted
- name: ppservice-webapp-central
  parameters:
    memory: 1024M
  properties:
    JBP_CONFIG_RESOURCE_CONFIGURATION: >
      ['tomcat/webapps/ROOT/WEB-INF/classes/ppe-schema-orm.xml':
        {'sap.dataaccess-common.schema': '<DB_SCHEMA>' },
        'tomcat/webapps/ROOT/META-INF/context.xml':
          {'ppeHana-service-name': 'ppeHana',
           'custJarBasePath': '/usr/sap/hana/shared/XSA/customjars'}]
    JBP_CONFIG_JAVA_OPTS: >
      java_opts: -Dsap.dataaccess-common.db.client=<DB_CLIENT>
                 -Dsap.dataaccess-common.logSys=<LOGSYS>
  provides:
    - name: java
# some lines omitted
```



Only the replacement of parameter `custJarBasePath` is relevant. Choose the other settings according to your specific setup.

4. (Re-)install the PPS as follows:

```
xs install XSACOPPP<version>.ZIP -e myPPS.mtaext -o ALLOW_SC_SAME_VERSION
```

5. If the content of the directory changes, restart the PPS:

```
xs restart ppservice-webapp-central
```

Extending the PPS-Based Price Calculation in SAP ERP and SAP S/4HANA Sales Documents

Depending on the SAP ERP or SAP S/4HANA release, it is also possible to call the PPS from SAP ERP/SAP S/4HANA. In this context, extensions are possible as well. This chapter describes the offered possibilities.

Extending via BAdIs

The enhancement spot **OPP_ENHANCE_SD** offers several BAdIs to extend the PPS-based price calculation in SAP ERP sales documents. For more information about implementing these BAdIs, see the system documentation. In Customizing, you can find these BAdIs under *Logistics - General > Omnichannel Promotion Pricing (OPP) > Business Add-Ins (BAdIs)*.

Enriching with Further Article Hierarchy Nodes

By default, the implementation **OPP_ENHANCE_BY_ARTHIER** of BAdI **OPP_ENHANCE_REQUEST** adds up to three article hierarchy nodes to the corresponding article via the following logic:

- The maximum depth of the article hierarchy is determined from table **WRF_MATGRP_TREE**. The entry with the highest value of **TREE_LEVEL** defines the article hierarchy depth.
- The enrichment is done for the maximum level and the two levels below. For example, the maximum level has value 08, the enrichment searches for nodes with level 06, 07 and 08 having this article as leaf.
- If the article hierarchy is not balanced, it can result in less than 3 article hierarchy nodes. For example, the considered article is assigned to a hierarchy node on level 06, only this node is considered. Nodes on level 04 or 05 are not taken into account.

If this logic is not sufficient and more than the three lowest levels should be considered, do the following:

- Create an append structure to DDIC structure **KOMP**.
- Add the following fields to this append structure:
 - Field name **NODEx** (x=4,5,...)
 - Type **WRF_STRUC_NODE2**

The system automatically considers further article hierarchy nodes according to the logic described above.

Extending the SAP ERP/ SAP S/4HANA PPS Client

The PPS client in SAP ERP is responsible for the conversion between ABAP data objects (structures, internal tables, data elements) representing the elements of the PPS client API and their XML representation as supported by the PPS. Moreover, it takes care for the HTTP-based data exchange. It is independent from the integration into SD processing and implemented by class **CL_OPP_PPS_CLIENT**. Technically, the ABAP types processed by the SAP ERP PPS client are not simple Data Dictionary types, but proxy data types with a binding between the ABAP type and the corresponding XSD type of the client API.

Therefore, it is not possible to simply enhance the ABAP part of the client API in order to add further information to an request or response. To support extensibility, the generated ABAP proxy structures provide predefined extension segments that can be used to transport additional information to the PPS and back to the caller. Each of these extension segments has the field name **ANY** and type **OPP_ANY_TAB** which is a standard table of raw strings. The following picture shows the ABAP proxy editor with the top level elements of the data type corresponding to the PPS request (**OPP_MESSAGE2**):

Display Message Type message2

Message Type: Active

Properties | External View | Internal View | Objects | Warnings

External Name

- message2
 - ARTSHeader
 - PriceCalculateBody
 - TransactionID
 - DateTime
 - Loyalty
 - ShoppingBasket
 - RegularSalesUnitPriceRoundingRule
 - any**
 - Transaction Type
 - NetPriceFlag
 - CalculationMode
 - any
 - InternalMajorVersion
 - InternalMinorVersion

any	
Name	any
ABAP Object	FIEL Feld
ABAP Name	ANY

Global Simple Type	
Name	any
Namespace	http://www.sap.com/IXRetail/namespace/
ABAP Object	TTYP Table Type
ABAP Name	OPP_ANY_TAB
Prefix	OPP_
XSD Type	any
ABAP Type	XSDANY

During runtime, these raw strings may contain XML fragments that are automatically mapped by the PPS into a generic data format so that it can be processed by server side customer extensions. The structure of each XML fragment can be arbitrarily complex, so that also deep ABAP structures or tables can be used. The mapping between the ABAP data structures and the XML fragment that is contained in the raw string is offered by ABAP interface **IF_OPP_PPS_EXTENSION_HELPER** with the following 2 methods:

- **WRAP**: This method transforms the provided ABAP data into the XML fragment
- **UNWRAP**: This method transforms the provided raw string containing an XML fragment into the corresponding ABAP data object. The target type of the ABAP data object must match the structure of the XML fragment.

The following ABAP program shows how to perform the wrapping and unwrapping for the extension segments:

Usage of IF_OPP_SD_EXTENSION_HELPER

```

*&-----*
*& Report ZZ_DEMO_EXTENSION_HELPER
*&-----*
REPORT ZZ_DEMO_EXTENSION_HELPER.
* Get instance of IF_OPP_PPS_EXTENSION_HELPER
DATA(go_helper) = cl_opp_core_factory=>get_factory( )->get_pps_extension_helper( ).
* The PPS request
DATA gs_request TYPE opp_message2.
* Example ABAP data: An integer giving the final answer
DATA g_src_data TYPE i VALUE 42.
DATA g_tgt_data LIKE g_src_data.
* Wrap ABAP data into XML fragment
DATA(g_wrapped) = go_helper->wrap( g_src_data ).
* Append XML fragment to extension segment of ARTS header
APPEND g_wrapped TO gs_request-artsheader-any.
* Do the PPS call etc. For reasons of simplicity we here just extract the request data
* Unwrap XML fragment to ABAP format - note that the data type matches
go_helper->unwrap( EXPORTING i_xml_fragment = gs_request-artsheader-any[ 1 ]
                  IMPORTING ed_data = g_tgt_data ).

IF g_src_data = g_tgt_data.
  WRITE 'It really works!'.
ENDIF.

```

From a technical perspective, the "identity" ABAP Simple Transformation is used to convert between ABAP and XML representation, hence the possibilities and restrictions described in the ABAP keyword documentation for format "asXML" apply (see also http://help.sap.com/abapdocu_740/en/abenabap_xslt_asxml.htm). The PPS is a Java-based application that does not know ABAP-specific concepts. This has some implications:

- Reference types should not be wrapped into XML fragments as the unwrapping may not be possible.
- Hashed or sorted tables including sorted or hashed table Indexes should not be used because the PPS is not aware of the restrictions for the structure of the corresponding XML representation.
- The ABAP-specific handling of currencies with other than 2 decimal places is not supported. We recommend to use a string representation of amounts for Transfer within XML ANY elements.
- By default Java has no direct counterpart to the ABAP built-in types **decfloat16** and **decfloat34**. If **java.math.BigDecimal** is used on Java side, precision loss can occur while unwrapping.



In the initial shipment of the SAP ERP PPS client, Client API version 2.0 is supported. This means, that all elements of version 2.0 have the corresponding ABAP proxies present. However, from an application side, only version 1.0 requests are supported.

Support for Mocking of the SAP ERP/ SAP S/4HANA PPS Client

It is possible to perform integration tests of PPS-based price calculation in the pricing of a sales document without having a running PPS. This is done by replacing the class **CL_OPP_CORE_FACTORY** that is responsible for creating the PPS client.

This can be done as follows:

1. Create a subclass of class **CL_OPP_CORE_FACTORY**, for example, **ZZCL_OPP MOCK_FACTORY**.
2. In this class redefine the method **IF_OPP_CORE_FACTORY~GET_PPS_CLIENT** so that it returns a mocked version of the PPS client. This must be a subclass of class **CL_OPP_PPS_CLIENT**.
3. In the subclass of **CL_OPP_PPS_CLIENT** redefine the method **IF_OPP_PPS_CLIENT~CALL** so that the PPS call is mocked according to your needs.
4. Set the SET / GET parameter **OPP_CORE_FACTORY** to the name of the class replacing **CL_OPP_CORE_FACTORY**, for example, **ZZCL_OPP MOCK_FACTORY**.



If in transaction **SCC4** the client role is set to 'P' (Productive), the mocking of the PPS client is not possible with this approach.

www.sap.com/contactsap

© 2015 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies. Please see <http://www.sap.com/corporate-en/legal/copyright/index.epx#trademark> for additional trademark information and notices.

Material Number:

