User Guide | PUBLIC

2018-12-27

SAP Solution Sales Configuration

THE BEST RUN SAP

Content

1 SAP Solution Sales Configuration for SAP S/4HANA

Product Information

Product	SAP Solution Sales Configuration for SAP S/4HANA
Release	1907 SP04
Based On	S/4HANA 1809 SP01
Documentation Published	2020

Use

SAP Solution Sales Configuration for SAP S/4HANA is a set of software products that help customers to configure and sell solutions made of complex product combinations. In addition to selling their own products, many businesses also sell solutions that include products, services, and parts from other manufacturers and providers. For example, technology companies sell solutions that would typically include multiple combinations of highly complex hardware, software, and services, and each of these can have options or features that must be specified by a customer during the ordering process.

From a modeling perspective, each individual product can be split into multiple components, and there are numerous possible relationships and dependencies between the product instances. SAP Solution Sales Configuration for SAP S/4HANA provides a flexible modeling environment, which in turn simplifies the ordering and configuration process for the customer.

SAP Solution Sales Configuration for SAP S/4HANA also includes a configuration engine that provides the ability to perform bottom-up configuration in addition to the normal top-down approach. The system also offers innovations that make it easier to maintain configuration model data. The system is designed to provide efficient configuration execution performance.

Integration

SAP Solution Sales Configuration for SAP S/4HANA is integrated with the following systems:

- SAP S/4HANA 1809 SP01 (and higher versions)
- Hybris Commerce Suite
  For information about supported versions, see SAP Note 2227752 (Supported Combinations of SAP Commerce Platform and SAP Solution Sales Configuration)

Features

- **Solution Modeling Environment**
  The system provides a modeling environment for creating complex solution models. The solution modeling environment is based on the Eclipse Rich Client Platform (RCP) and can be integrated with SAP S/4HANA for exchanging configuration master data. Within the modeling environment, you define model elements such as classes and characteristics, and then you define dependencies between your model elements.
  The solution modeling environment can also be integrated with a Source Code Management System to support collaborative modeling and allow several modelers to work on the same model simultaneously.
  The solution modeling environment includes a test user interface and a configuration engine, which allow you to test and optimize your model before transporting it to the target sales system.
  For more information, see Solution Modeling Environment [page 7].
- **Solution Configuration Environment**
  The application provides a configuration engine and a configuration user interface that fully integrate with the ordering process in the SAP S/4HANA systems. You can enter an advanced mode product (a solution) as an item in a sales document, and then choose to configure it. The system opens the enhanced configuration user interface, where you can enter the solution components and configure their options. When you save the configuration, you are returned to the sales document.
  For more information, see Solution Configuration Environment [page 140].

2 Solution Modeling Environment

Use

The solution modeling environment is the part of SAP Solution Sales Configuration that is used to create advanced solution models. It is integrated with SAP S/4HANA. You can download and use existing S/4HANA master data like products, classes, and knowledge bases. You define and test your solution model in the solution modeling environment before transporting it to the target S/4HANA system, where it is integrated with the sales ordering process and used during solution configuration.

Integration

The solution modeling environment is integrated with the following systems:

- SAP S/4HANA

> i Note
>
> The Solution Modeling Environment is a reusable component that can be connected with CRM, ECC, and S/4. ECC and S/4 can be considered to be synonymous here.

Features

- **Advanced Mode Solution Modeling**
  Models for products, such as a car, are delimited and have a finite number of component parts. Solutions are not delimited; they can have many optional parts with complex dependencies between these parts. The solution modeling environment allows you to create advanced solution models. For example, you can define non-part components (instances) without using a bill of material (BOM).
  For information about the elements in a solution model, see Solution Model [page 9]. For information about the steps required to create a solution model, see Solution Modeling [page 71].
- **Data Exchange**
  The solution modeling environment is integrated with SAP S/4HANA and can be used to exchange configuration master data. For more information, see Data Exchange [page 107].
- **Collaborative Modeling**
  The system supports collaboration between modelers by integrating with standard version control systems. For more information, see Collaboration Between Modelers [page 109].
- **Test User Interface and Test Configuration Engine**
  The system provides a test user interface and a test configuration engine, where you can test and optimize your model before transporting it to the target system. In the test user interface, you can configure your solution to ensure that your model is performing as expected. In addition, you can save test scripts,

identify the constraints applied during a configuration session, and test the performance of constraints. For more information, see Testing Models Locally [page 90].

- **Knowledge Base Orchestration**
  The system supports knowledge base orchestration in the downstream processes in the configuration engine and the configuration user interface. Knowledge base orchestration is a process that enables multiple knowledge bases to handle large configurations. It encapsulates large knowledge base results into smaller, more manageable components and presents multiple independent configuration objects as a single, coordinated configuration session to the end user.
- **Solution Modeling User Interface**
  The solution modeling user interface is based on the Eclipse Rich Client Platform (RCP). It is designed to support different ways of navigation and to perform quickly when modelers are working with large, complex models.
  The solution modeling user interface provides wizards for creating model elements. It also provides a text editor for writing and maintaining element definitions. The text editor has the following features:
  - **Auto-Completion**
    You define model elements with a precise syntax. The text editor provides an auto-completion feature to support modelers when they manually define elements. To use auto-completion, perform the following steps:
    1. Position the cursor at the relevant location and press `CTRL` + `SPACEBAR`.
       The system displays a dialog box listing all of the syntactically correct elements at the current cursor position.
    2. Double-click an element to insert it into the text editor.
  - **Syntax Error Indicator**
    The syntax error icon is displayed on the left side of any lines that contain a syntax error. Furthermore, if syntax errors exist, a red square is displayed in the top right corner of the text editor. You can move the cursor over the red square to see the total number of syntax errors.
  - **Tabbed Display**
    Elements opened in the text editor are displayed on tab pages. You can open several elements for editing and use the tab pages to switch between them.
  - **Templates**
    The solution modeling environment provides templates for common model elements and also allows you to define your own templates. Templates are indicated by a green dot.
    For information about maintaining templates, see Maintaining Modeling Templates [page 108].
- **Data Loader**
  The Data Loader is a component used to download configuration data for your solutions from a SAP S/4HANA source system to a local database management system such as Microsoft SQL Server. For more information, see Data Loader in the Solution Modeling Environment [page 105].

Constraints

- Certain modeling syntax that works in Variant Configuration (LO-VC) is not supported in the Internet Pricing and Configurator (IPC) configuration engine. Since SAP Solution Sales Configuration is based on the same IPC technology, those restrictions also apply to SAP Solution Sales Configuration. For more information about the restrictions, refer to SAP Note 1819856 (Additions to IPC VC deltalist).
- Solution modeling can be very complex as the requirements and dependencies to be considered when you assemble a solution can be both numerous and complex. While the configuration engine in SAP Solution

Sales Configuration is built specifically to handle these processing intensive requirements, the critical factor in ensuring efficient processing is the definition of the model. Great care must be taken in developing solution models. It is a similar process to any other application development and requires training, abstract thought, sophistication, and elegance of design.

2.1 Introduction to Solution Models

Definition

A solution model is a hierarchical decomposition of a solution. It defines the products (configurable materials and the services) that can be contained within the solution. It also defines the relationships between the various elements of the solution, including any dependencies (constraints and rules).

Structure

A solution model contains the following parts:

- **Bills of Material (BOMs)**
  For more information, see Bill of Material [page 63].
  For information about defining BOMs, see Defining Dynamic BOMs [page 63].
- **Characteristics (cstics)**
  For more information, see Characteristic [page 23].
  For information about defining characteristics, see Defining Characteristics [page 24].
- **Classes**
  For more information, see Class [page 15].
  For information about defining classes, see Defining Classes [page 15].
- **Dependency Nets**
  For more information, see Dependency Net [page 53].
  For information about defining dependency nets, see Defining Solution Dependencies [page 44].
- **User-Defined Functions**
  For more information, see User-Defined Function [page 54].
- **Interface Designs**
  For more information, see Interface Design [page 61].
  For information about creating interface designs, see Example: Defining an Interface Design [page 61].
- **Knowledge Bases (KBs)**
  For more information, see Knowledge Base [page 10].
  For information about defining knowledge bases, see Defining Knowledge Bases [page 10].
- **Materials**
  For more information, see Material [page 20].
  For information about defining materials, see Defining Materials [page 20].
- **Variant Tables**
  For more information, see Variant Table [page 37].
  For information about defining variant tables, see Defining Variant Tables [page 42].

2.1.1 Knowledge Base

Definition

A knowledge base contains the master data for a solution model. The knowledge base is exported from the solution modeling environment, initially to the test user interface and the test configuration engine to be tested, and then to the target sales system where it is used by the configuration engine.

Structure

In the solution modeling environment, a knowledge base is defined with identifier, version, tasks, and profiles.

More Information

For information about defining knowledge bases, see Defining Knowledge Bases [page 10].

2.1.1.1 Defining Knowledge Bases

Use

You use this procedure to create a task, which is then used to define a knowledge base. You can define the task in the same file as the knowledge base, or in a separate file.

Procedure

Creating a Knowledge Base

1. Choose *File* > *New* > *Empty Model File*.
   The system displays a dialog prompting you to choose a folder and enter a file name for the new knowledge base.
2. Enter the required data.
3. Choose *Finish*.
   The system displays an empty file.
4. Press CTRL + SPACEBAR.
5. Double-click *knowledgeBase*.
6. Enter the required data and save the knowledge base.

2.1.1.2 Knowledge Base Runtime Version

A knowledge base runtime version (kbrtv) is created from a knowledge base definition by invoking the *Export Knowledge Base* function in the solution modeling environment.

You can call the *Export Knowledge Base* function:

- From the *File* menu, by choosing ▶ *File* ▶ *Export* ▶ *SAP Solution Configuration* ▶ *Export Knowledge Base* ◀.
- From the Model Explorer, by right-clicking a knowledge base definition and choosing *Export Knowledge Base*.

When you export a knowledge base, you can include variant table content in the kbrtv even though the `externalTable` keyword is specified in the variant table definition. This is useful for testing the knowledge base locally. If external content is available and content is also included in the knowledge base, the **external** content is used at runtime.

To include local content in the kbrtv (from either "rows" values or "file" content), select the *Include Local Variant Table Content During Export* option in the kb export dialog, or *Local VT* in the export multiple kbs dialog.

2.1.1.3 Example: Defining a Knowledge Base Definition

> i Note
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

A knowledge base definition indicates the materials or classes for which the knowledge base will be used. It also lists the dependencies to be included in the knowledge base runtime version. To define a knowledge base definition, use the following syntax:

knowledgeBase **`identifier`** {

version **`kb-version`**

*validFrom***`date`**

logsys **`logical-system`**

tasks **`task-id`**

profiles

name **`profile-name`** *class class-id* || *material material-id*

}

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
knowledgeBase	Denotes the start of a knowledge base definition encompassed within **{...}**	-	Required
identifier	Knowledge base ID used in the model. Max. length 30 bytes It appears in the KBOBJNAME field in the comm_cfgkb table.	-	Required
version	Keyword	-	Required
kb-version	Version is a text field; common format is “0.0.0.0” but any alternative format is acceptable.	-	Required
logsys	Keyword	-	Optional
logical-system	Logical system name where the materials/classes named in profiles (below) originated. The logical system value is derived from the back end. All materials defined for profiles of this knowledge base are checked. If multiple entries exist, a list is presented for selection by the user.	-	Optional
validFrom date	Date in the format “YYYY-MM-DD”from which this knowledge base will be effective from. If validFrom is not specified, the current date is used.	Required with validFrom keyword	
tasks	Keyword	-	Optional
task-id	ID of task containing the dependencies to be included in this knowledge base	Required with tasks keyword	

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`profiles`	Denotes the start of the list of profiles. This list can contain multiple profiles. Each name-class/material pair is a distinct profile definition.	-	Required
`name`	Keyword	-	Required
`'profile-name'`	The profile name is a key value for the knowledge base tables.	-	Required
`class`	Keyword	At least one class or material must be specified	Required
`class-id`	A class-id defined in this model	-	Required with class keyword
`material`	Keyword	At least one class or material must be specified	Required
`material-id`	A material-id defined in this model	-	Required with material keyword

Example

```
knowledgeBase PRODUCT_X_KB {
   version "0.0.0.6"
   validFrom "2020-01-01"    logsys "LOCAL"
   tasks
      PRODUCT_X_TSK
   profiles
   name 'PRODUCT_X_PRF_2' class PRODUCT_X
}
```

2.1.1.4 Definition of External Texts

Use

After you have exported a knowledge base to the SAP backend system, you can create or change the texts that have been defined for each characteristic, characteristic value, class, or material. Editing the texts directly in the backend means that you do not have to export your knowledge base again if you want to make changes.

> i Note
>
> A particularity of maintaining material texts externally is that if a material is also defined in the product master in the back end, the text from the product master is taken automatically. This avoids the need for double maintenance. If the material is not defined in the back end, you can maintain the texts manually in the same way as all of the other objects.

Integration

You can maintain external texts manually in Customizing or you can export them from the Solution Modeling Environment as follows:

1. Choose *File* > *Export* > *Export Knowledge Base Localization*.
2. Choose a connection.
3. Select your knowledge base and, if required, enter a date from which the texts are valid.

Activities

You maintain the texts in Customizing for S/4HANA under *Logistics - General* > *Solution Sales Configuration* > *Maintain External Texts*.

> i Note
>
> If you expect size of the external variant table to be large, consider the recommendations provided in SAP Note 2428939 (Implement query caching to improve Variant Table access performance).

Example

In your knowledge base, you have a characteristic with the names COLOR and FARBE for English and German respectively. After you have exported your knowledge base to the back end, you can localize the characteristic for the United Kingdom by changing COLOR to COLOUR, and you can also add a new translation COULEUR for French.

More Information

For more information, see the Customizing documentation in the system.

2.1.2 Class

Definition

Classes are used to represent the different configurable parts within a solution model. Classes can be defined in a hierarchy, with subclasses inheriting characteristics from their parent class. For example, you can define a class called computer with two subclasses: laptop and desktop.

Structure

A class has an identifier, a name, and one or more characteristics.

More Information

For information about characteristics, see Characteristic [page 23].

For information about defining classes, see Defining Classes [page 15].

2.1.2.1 Defining Classes

Use

You use this procedure to define new classes in the Eclipse-based solution modeling environment.

Procedure

Creating a New Class Using the Main Menu

1. Choose *File* > *New* > *Class* .
2. Enter the required data.
3. Choose *Finish*.
   The new class is opened in the text editor.

Creating a New Class Using the Context Menu

1. In the *Project Explorer*, right-click *classes*.
   The context menu is displayed.
2. Choose *New* > *Other...*
3. Select *Class* and choose *Next*.

> i Note
>
> As you start to type *class*, the contents of the explorer are filtered, such that only the matching entries are displayed.

4. Enter the required data.
5. Choose *Finish*.
   The new class is opened in the text editor.

Adding Characteristics to a Class Using the Text Editor

1. Open the class in the text editor.
2. Place the cursor at the desired location.

> i Note
>
> Characteristics are entered in a comma-separated list under *characteristics*.

3. Press CTRL + SPACEBAR.
   The system displays a list of all the available characteristics.
4. Double-click the characteristic you want to add to the class.

Adding More Information for a Characteristic

1. In the text editor, place the cursor after the characteristic.
2. Enter a space and press CTRL + SPACEBAR.
   The system displays the following options:
   - **Default**
     You use this option to specify a default value for the characteristic.
   - **Invisible**
     You use this option to hide the characteristic in the configuration user interface (UI).
   - **Required**
     You use this option to make the characteristic mandatory in the configuration UI.
   - **Value**
     You use this option to specify the value of the characteristic.
3. Double-click the option you want to add to the characteristic.
4. You must specify a value for the *default* and *value* options.

Example

```
class SME_EXAMPLE_MEMORY {

name "Memory"

characteristics

SME_INSTANCE_NF invisible,

SME_MEMORY_SPEED_S required,

SME_PART_NUMBER_S required

}
```

2.1.2.2 Defining a Class

> i Note
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

To define a class, use the following syntax:

class **`identifier`** **extends** **`super-class-id, super-class-id2,...`** {

name **`"30 bytes descriptive text"`** || names EN **`"descriptive English text"`**

, FR **`"descriptive French text"`**

longName **`"unlimited descriptive text"`** || longNames EN **`"unlimited text in multiple languages"`**

urls { **`"url"`** *label* **`"30 bytes descriptive text", "url"`** *label* **`"30 bytes descriptive text" ...}`**

> i Note
>
> The `urls` keyword can be used at group, cstic, cstic value, class, and material level. As a general rule of thumb, all elements that could be UI relevant have this attribute. It is also possible to define more than one URL by using curly brackets.

characteristics **`characteristic-id`**

--remaining keywords are specified for each characteristic on this class definition--

required

noinput

urls { **`"url"`** *label* **`"30 bytes descriptive text", "url"`** *label* **`"30 bytes descriptive text" ...}`**

values (**`value1, value2,...`** **)**

defaultValues(**`value1,value2,...`** **)**

assignedValues(**`value1,value2,...`** **)**

}

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`class`	Denotes the start of a class definition encompassed within {...}	-	Required

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`identifier`	This is the class ID used in the model to reference this class. It must be uppercase. Max. length 18 bytes	-	Required
`extends`	Keyword indicating inheritance from a super-class	-	Optional
`super-class-id`	Identifier of the class extended by this definition	Required if extends keyword is used	Optional
`super-class-id2`	Identifier(s) of additional classes extended by this definition. Multi-inheritance is supported.	-	Optional
`{`	Starts the set of keywords that define this object	-	Required
`}`	Ends class definition	-	Required
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	-	Optional
`longName` or `longNames`	Unlimited length text available from the “more information” link on the configuration UI. It is possible to embed hyperlinks in this text.	-	Optional
`characteristics`	Keyword indicating the start of the list of characteristics for this class	-	Optional
The following keywords are specified for each characteristic			

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`required`	Indicates that the configuration is considered incomplete unless a value is specified for this characteristic	-	Optional
`noinput`	Blocks the field from user input. Values may still be assigned/changed by means of constraints or rules.	-	Optional
`defaultValues()`	Sets the default value(s) of this characteristic for this class. More than one value may be specified only if the characteristic is defined as multiValue.		Optional
`assignedValues()`	Assigns a value (or set of values) to this characteristic for this class. More than one value may be specified only if the characteristic is defined as multiValue. This is a fixed assignment. The value(s) cannot be changed by the user, constraints, or rules.	-	Optional

Example

```
class PRODUCT_Y extends SSC_SD_HIER {
name "Product Y class"
characteristics
SALES_ITEM_NAME      required noinput,
NUM_SUBSCRIBERS      required defaultValues (5000),
TRAFFIC_PER_SUBSCRIBER  required assignedValues (25),
TOTAL_TRAFFIC           invisible noinput
}
```

2.1.3 Material

Definition

A material is an elementary component of a solution model.

In the Eclipse-based solution modeling environment, modelers define one or more materials, which can then be referenced in constraints and rules.

Along with products and classes, materials are associated with a knowledge base by means of a profile in the knowledge base definition.

Structure

Materials have a class and an identifier.

More Information

For information about defining materials, see Defining Materials [page 20].

2.1.3.1 Defining Materials

Context

You use this procedure to define new materials in the Eclipse-based solution modeling environment.

Procedure

1. Choose *File* > *New* > *Empty Model File*

   The system displays a dialog prompting you to choose a folder and enter a file name for the new material.
2. Enter the required data.
3. Choose *Finish*.

   An empty file is displayed in the text editor.
4. Press CTRL + SPACEBAR and double-click *material*.

> i Note
>
> Alternatively you can type **`material`**.

5. Enter the required data and save the new material.

Example

material SME_LAPTOP {

name 'Laptop'

classes

SME_LAPTOP

boms

SME_LAPTOP

}

2.1.3.2 Example: Defining a Material

> i Note
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

To define a material, use the following syntax:

material **`identifier`** **externalID** **`'external_identifier'`** *{*

name **`"30 bytes descriptive text"`** || **names** EN **`"descriptive English text"`**

, FR **`"descriptive`**

`French text"`

longName **`"unlimited descriptive text"`** || **longNames** EN **`"unlimited text in multiple languages"`**

urls { **`"url"`** *label* **`"30 bytes descriptive text"`** , **`"url"`** *label* **`"30 bytes descriptive text...`** **}**

classes **`class-id`**

boms **`bom-id`**

```
}
```

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`class`	Denotes the start of a class definition	-	Required
`identifier`	This is the material ID used in the model to reference this material. Max. length 18 bytes	-	Required
`external_ID`	Keyword indicating that S/4HANA uses an alternative ID to reference this material	-	Optional
`external_idenitfier`	The ID used by S/4HANA to reference this material	Required if `external_ID` keyword is used	Optional
`{`	Starts the set of keywords that define this object	-	Required
`}`	Ends class definition	-	Required
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	-	Optional
`longName` or `longNames`	Unlimited length text available from the “more information” link on the configuration UI. It is possible to embed hyperlinks in this text.	-	Optional
`classes`	Keyword designating the start of the list of classes assigned to this material	-	Optional
`class-id`	One or more class IDs assigned to this material. Separate multiple classes by a comma.	Required if `classes` keyword is used	Optional

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`boms`	Keyword designating the bill of material associated with this assembly material. (Note: BOMs are not recommended; best practice recommends use of ADTs to represent “part-of” relationships.)	-	Optional
`bom`	A bill of material defined to represent the components of this assembly material	Required if `boms` keyword is used	Optional

Example

```
material CABLE_SET {
 name "Cable set"
 classes
  CABLE_SET
}
```

2.1.4 Characteristic

Definition

Characteristics are the lowest level of detail in a solution model. They are used to define the distinguishing properties or attributes of a class. For example, a computer monitor can have a characteristic called size.

Structure

A characteristic has an identifier, a name, and a type. Depending on the type selected, you may need to specify additional information, as shown in the following table:

Characteristic Types

Characteristic Type	Information to Be Defined
Text	Length

Characteristic Type	Information to Be Defined
Numeric	Length Decimal Places
Date	
ADT	Class

Reference Characteristics

Characteristics of type *Text*, *Numeric*, and *Date* can be made reference characteristics by using the `reference` keyword. Reference characteristics can be used to do the following:

- Pass a value to the configuration as a context
- Pass a value from the configuration to the sales document
  In this case, the user is usually required to enter the value of the reference characteristic during the configuration session (for example, size, weight, or start date).

Abstract Data Types

An abstract data type (ADT) is a special type of characteristic that is used to define the relationship between two classes. For example, a computer monitor can have ADTs called **has part** and **is part of**.

SAP Solution Sales Configuration provides the following predefined ADTs:

- `SALES HARD`
- `SALES SOFT`

These ADTs are used to define parent-child relationships as either a **hard tie** or a **soft tie**. If two classes have a hard tie, they cannot be split into separate sales documents. Only classes with soft ties can be split into different sales documents.

You maintain the names of the hard and soft tie sales ADTs in Customizing for *Customer Relationship Management* under ▶ *Basic Functions* > *Solution Sales Configuration* > *Maintain Item Relationship Types* ◀.

More Information

For information about defining characteristics, see Defining Characteristics [page 24].

2.1.4.1 Defining Characteristics

Use

You use this procedure to define new characteristics in the Eclipse-based solution modeling environment.

Procedure

Defining Characteristics in the Main Menu

1. In the main menu, choose *File* › *New* › *Characteristic*.
2. Enter the required data.

> i Note
>
> An *Identifier* must consist of uppercase alphanumeric characters and underscores; it cannot start with a numeric character. The *Name* defaults to the same as *identifier*; it can be changed, if required.

3. Choose *Finish*.
   The new characteristic is opened in the text editor.

Defining Characteristics in the Context Menu

1. In the *Project Explorer*, right-click *cstics* to display the context menu.
2. Choose *New* › *Other...*
3. Choose *Characteristic.*

> i Note
>
> As you start to type *characteristic*, the contents of the explorer are filtered, such that only the matching entries are displayed.

4. Choose *Next.*
5. Enter the required data.
6. Choose *Finish.*
   The new characteristic is opened in the text editor.

2.1.4.2 Defining a Text Characteristic

> i Note
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

To define a text characteristic, use the following syntax:

characteristic **`identifier`** *{*

name **`"30 bytes descriptive text"`** || **names EN** **`"descriptive English text"`**

, FR **`"descriptive French text"`**

longName `"unlimited descriptive text"` || longNames EN `"unlimited text in multiple languages"`

status released || prepared || blocked

textLength `1-132`

restrictable `(x-additionalValues)`

caseSensitive

multiValue

additionalValues `(x-restrictable)`

`values` "a" name `"Aye"` || names EN `"descriptive English text"`

, FR `"descriptive French text"`

`, "b"` name `"Bee"` || names EN

}

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`characteristic`	Denotes the start of a characteristic definition encompassed within {...}	-	Required
`identifier`	This is the characteristic ID used in the model to reference this characteristic. It must be uppercase. Max. length 40 bytes	-	Required
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	-	Optional
`longName` or `longNames`	Unlimited length text available from the "more information" link on the configuration UI. It is possible to embed hyperlinks in this text.	If the name keyword is not used, the longName is **not** used as the name.	Optional

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`status`	If used, one of three additional keywords must be specified - required, prepared, blocked. This keyword is ignored by the SME. It is available for compatibility with master data definitions imported from S/4HANA.	-	Optional
`textLength`	Indicates that this is a text characteristic and specifies the maximum length of the value. Maximum `textLength` is 132.	-	Required
`restrictable`	Indicates that the domain of values can be restricted. If the `values` clause is used, the domain is restricted to those values, and can be further restricted in constraints. If no `values` clause is used, the initial static domain is the "unrestricted domain". Any value is valid, but can still be restricted by constraints.	This clause and the `additionalValues` clause are mutually exclusive.	Optional
`caseSensitive`	Indicates that values can contain uppercase and lowercase values	-	Optional
`multiValue`	Indicates that more than one value can be assigned to this characteristic	-	Optional
`additionalValues`	Allows user entry of values. Can be used with the values clause. The values specified in the `values` clause are presented for selection, but the user can also enter a different value if the `additionalValues` clause is specified.	This clause and the `restrictable` clause are mutually exclusive.	Optional

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`values`	Lists the permitted values. No values indicates an unrestricted domain. Any type-consistent value is allowed. Values can be specified with or without names and names can be specified in multiple languages.	-	Optional

Example

```
characteristic CABINET_TYPE {
      name "Cabinet Type"
      textLength 1
      caseSensitive
      restrictable
      values
         'a' name 'little aye',
         'A' name 'Aye',
         'B' name 'Bee',
         'C' name 'See',
         'D' name 'Dee',
         'd' name 'little dee'
}
```

2.1.4.3 Defining a Numeric Characteristic

> i Note
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

To define a numeric characteristic, use the following syntax:

characteristic **`identifier`** {

name **`"30 bytes descriptive text"`** || names EN **`"descriptive English text"`**

, FR **`"descriptive French text"`**

longName **`"unlimited descriptive text"`** || longNames EN **`"unlimited text in multiple languages"`** *numericLength* **1-15**

decimalPlaces **0-(numericLength-1)**

restrictable (x-additionalValues)

multiValue

negativeValues

additionalValues (x-restrictable)

specialFunction aggregating

intervals < n1 || **<=** n1 || **>** n2 || **>=** n2 || n1 - n2 || (where n1, n2 are numeric values)

values **n1, n2**

}

Keyword/User Value	Meaning	Required/Optional
`characteristic`	Denotes the start of a characteristic definition encompassed within {...}	Required
`identifier`	This is the characteristic ID used in the model to reference this characteristic. It must be uppercase. Max. length 30 bytes	Required
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	Optional
`longName` or `longNames`	Unlimited length text available from the “more information” link on the configuration UI. It is possible to embed hyperlinks in this text.	Optional
`numericLength`	Indicates that this is a numeric characteristic and specifies the maximum length of the value. Maximum `numericLength` is 15.	Required
`decimalPlaces`	Indicates the number of decimal places of this numeric value. Maximum `decimalPlaces` is one less than the `numericLength`.	Optional

Keyword/User Value	Meaning	Required/Optional
`restrictable`	Indicates that the domain of values can be restricted. If the `values` clause is used, the domain is restricted to those values, and can be further restricted in constraints. If no `values` clause is used, the initial static domain is the “unrestricted domain”. Any value is valid, but can still be restricted by constraints.	Optional
`multiValue`	Indicates that more than one value can be assigned to this characteristic	Optional
`negativeValues`	Allows a numeric value to be a negative value	
`additionalValues`	Allows user entry of values. Can be used with the values clause. The values specified in the `values` clause are presented for selection, but the user can also enter a different value if the `additionalValues` clause is specified.	Optional
`specialFunction aggregating`	Designates this characteristic as usable in a rule with the `then_increment` statement	Optional
`specialFunction balanced`	Do not use - not supported	Prohibited
`values`	Lists the permitted values. No values indicates an unrestricted domain. Any type-consistent value is allowed	Optional
`specialFunction default`	Designates this value as the default value	Optional

Example

```
characteristic BLOCKING {
      name "Blocking"
      numericLength 2 decimalPlaces 1
      restrictable
      values 0.1, 0.5, 1, 2, 5
}
```

2.1.4.4 Defining a Date Characteristic

> i Note
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

To define a date characteristic, use the following syntax:

characteristic **`identifier`** *{*

name **`"30 bytes descriptive text"`** || **names EN** **`"descriptive English text"`**

, **FR** **`"descriptive French text"`**

longName **`"unlimited descriptive text"`** || **longNames EN** **`"unlimited text in multiple languages"`**

date

restrictable `(x-additionalValues)`

multiValue

additionalValues `(x-restrictable)`

values **`yyyy-mm-dd`**

}

Keyword/User Value	Meaning	Required/Optional
`characteristic`	Denotes the start of a characteristic definition encompassed within {...}	Required
`identifier`	This is the characteristic ID used in the model to reference this characteristic. It must be uppercase. Max. length 30 bytes	Required
`{`	Starts the set of keywords that define this object	Required
`}`	Ends the object (characteristic) definition	Required

Keyword/User Value	Meaning	Required/Optional
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	Optional
`longName` or `longNames`	Unlimited length text available from the “more information” link on the configuration UI. It is possible to embed hyperlinks in this text.	Optional
`date`	Indicates that this is a date characteristic	Required
`restrictable`	Indicates that the domain of values can be restricted. If the `values` clause is used, the domain is restricted to those values, and can be further restricted in constraints. If no `values` clause is used, the initial static domain is the “unrestricted domain”. Any value is valid, but can still be restricted by constraints.	Optional
`multiValue`	Indicates that more than one value can be assigned to this characteristic	Optional
`additionalValues`	Allows user entry of values. Can be used with the values clause. The values specified in the `values` clause are presented for selection, but the user can also enter a different value if the `additionalValues` clause is specified.	Optional
`values`	Lists the permitted values. No values indicates an unrestricted domain. i Note The date must be specified in “YYYY-MM-DD”format only. Any type-consistent value is allowed.	Optional

Example

```
characteristic BIRTHDAY {
      name "Date of Birth"
      date
      values
       1992-12-25
}
```

2.1.4.5 Defining an Abstract Data Type Characteristic

> **Note**
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

To define an abstract data type (ADT) characteristic, use the following syntax:

characteristic **`identifier`** *{*

name **`"30 bytes descriptive text"`** || **names EN** **`"descriptive English text"`**

, **FR** **`"descriptive French text"`**

longName **`"unlimited descriptive text"`** || **longNames EN** **`"unlimited text in multiple languages"`**

classType *class-identifier*

multiValue

}

Keyword/User Value	Meaning	Required/Optional
`characteristic`	Denotes the start of a characteristic definition encompassed within {...}	Required
`identifier`	This is the characteristic ID used in the model to reference this characteristic. It must be uppercase. Max. length 30 bytes	Required

Keyword/User Value	Meaning	Required/Optional
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	Optional
`longName` or `longNames`	Unlimited length text available from the “more information” link on the configuration UI. It is possible to embed hyperlinks in this text.	Optional
`classType`	Indicates that this is an abstract data type characteristic	Required
`class-identifier`	Identifier of the class referenced by this characteristic	Required
`multiValue`	Indicates that more than one value can be assigned to this characteristic	Optional

Example

```
characteristic CONTAINS_THESE_ITEMS {
 name "Contains these items"
 classType ITEM
 multiValue
}
```

2.1.4.6 Defining a Reference Characteristic

> i Note
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

A reference characteristic refers to the value of a field in the sales document. To define a reference characteristic, use the following syntax:

characteristic **`identifier`** *{*

name **`"30 bytes descriptive text"`** || names EN **`"descriptive English text"`**

, FR **`"descriptive French text"`**

longName **`"unlimited descriptive text"`** || longNames EN **`"unlimited text in multiple languages"`**

textLength **`1-30`** || *numericLength* **`1-15`** *decimalPlaces* **`0-(numericLength-1)`** || *date*

reference table **`table-name`**

field **`field-name`**

}

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`characteristic`	Denotes the start of a characteristic definition encompassed within **{...}**	-	Required
`identifier`	This is the characteristic ID used in the model to reference this characteristic. It must be uppercase. Max. length 30 bytes	-	Required
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	-	Optional
`longName` or `longNames`	Unlimited length text available from the “more information” link on the configuration UI. It is possible to embed hyperlinks in this text.	-	Optional

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`textLength`	Indicates that this is a text characteristic and specifies the maximum length of the value. Maximum `textLength` is 30.	One of the `numericLength` / `decimalPlaces`, `textLength`, or `date` keywords is required	Required
`numericLength`	Indicates that this is a numeric characteristic and specifies the maximum length of the value. Maximum `numericLength` is 15.	One of the `numericLength` / `decimalPlaces`, `textLength`, or `date` keywords is required	Required
`decimalPlaces`	Indicates the number of decimal places in this numeric value. Maximum `decimalPlaces` is one less than the `numericLength`.	Allowed only with `numericLength` keyword	Optional
`date`	Indicates that this is a date characteristic	One of the `numericLength` / `decimalPlaces`, `textLength`, or `date` keywords is required	Required
`reference table`	Indicates the table containing the field to which this characteristic refers	-	Required
`field`	Indicates the field to which this characteristic refers	-	Required

Example

```
characteristic ITEM_QUANTITY {
names
      EN 'Component quantity',
      DE 'Komponentenmenge'
numericLength 13 decimalPlaces 3
negativeValues
reference table 'STPO' field 'MENGE'
}
```

2.1.5 Variant Table

Definition

Variant tables are used to store combinations of values for characteristics. They can be used to infer values as part of constraints.

Structure

You define a variant table for one or more characteristics, whereby each characteristic has a column in the variant table. You enter a row in the variant table for each possible combination of characteristic values. Entering one of the characteristics then restricts the other characteristics that can be selected.

You define a variant table in the Solution Modeling environment by using the `row`, `file`, or `externalTable` keywords.

> i Note
>
> The `externalTable` keyword references a table from the ABAP back-end system. If this table exists, its content is loaded at runtime instead of the content that is part of the knowledge base runtime version.
>
> If you choose to enter a table from the ABAP back-end system, you must ensure that the ABAP table contains not only columns for the characteristics of your variant table definition, but also valid to and from dates, as well as the client (ABAP tables should be created on a client-specific basis). The valid to and from date columns must be named `SSC_FROM_DATE` and `SSC_TO_DATE` and must be of data type `DATS`.

Example

In the following example, memory description can be used to infer the memory size and memory speed:

Memory Description	Memory Size	Memory Speed
`1_1333`	1	1333
`2_1333`	2	1333
`2_1600`	2	1600
`2_2000`	2	2000
`4_1600`	4	1600
`4_2000`	4	2000

In this case, if a memory size of 2 is specified, the system automatically restricts the possible entries for the memory speed to 1333, 1600, and 2000.

More Information

For information about defining variant tables, see Defining Variant Tables [page 42].

2.1.5.1 Syntax for Defining a Variant Table

> **i Note**
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

To define a variant table, use the following syntax:

variantTable **`identifier`** *{*

name **`"30 bytes descriptive text"`** || **names** EN **`"descriptive English text"`**

, FR **`"descriptive French text"`**

characteristics **`characteristic-id`**

--indented keywords are specified for each characteristic in this class definition--

Primary

externalTable **`'db_tablename'`**

rows 'textvalue', numericvalue;

'textvalue', numericvalue

||

file "filename.csv"

type 'CSV'

options

}

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`variantTable`	Denotes the start of a variant table definition	-	Required

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`identifier`	This is the variant table ID used in the model to reference this variant table.	-	Required
`{`	Starts the set of keywords that define this object	-	Required
`}`	Ends object definition	-	Required
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	-	Optional
`characteristics`	Keyword indicating the start of the list of characteristics for this variant table	-	Optional
The following keywords are specified for each characteristic			
`primary`	Denotes a characteristic used to look up values of other characteristics. This is treated as a key field to look up other values from the variant table.	-	Required for at least one characteristic
End characteristic-specific keywords			
`externalTable`	Used to indicate to the knowledge base build process that `variantTable` content is not to be embedded in the knowledge base runtime version, and instead should be retrieved by the engine at runtime from the named SAP database table.	Can be used with or without the `rows` or `file` keywords. However, data defined in `rows` or `file` will be used if the variant table data is also exported with the model and no data is available in the external table (defined with the `externalTable` keyword).	Optional
`'db_tablename'`	Name of a table in SAP database	Used with `externalTable` only	Required

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`rows` (`x-file`)	Used to provide table content in the table definition. Values are separated by a comma. Rows are delimited by a semicolon. Text values are enclosed within quotes.	One of the keywords `rows`, `file`, or `externalTable` is required. `externalTable` can also be used in conjunction with `rows` or `file`.	Optional
`file` (`x-rows`)	Used to provide table content from a file. The file must be in the same project folder as the file containing this variant table definition. Using the .vtable suffix is recommended.	One of the keywords `rows`, `file`, or `externalTable` is required. `externalTable` can also be used in conjunction with `rows` or `file`.	Optional
`type`	Indicates the type of file being referenced. Possible values are: • `vtable` (default): Supports .vtable files with tabulator-separated fields, without quotes, and without escapes • `cvs`: comma separated values	Used with `file` only	Optional

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`options`	Used to provide options related to data present in the file. Available options can be specified in the format: option1=value1 option2=value2. For example, CSV supports the following options: • `charset=ISO-8859-1` The character set of the file, such as “ISO-8859-1” or “UTF-8”. Default is Eclipse/OS-dependent. • `delimiter=,` The value delimiter. Default is ','. For tabulator or space characters, use TAB or SPACE. • `quoteChar="` The character to quote a value. Default is '"'. • `commentChars=#` The character(s) to recognize a comment line. Default is disabled. • `surroundingSpacesNeedQuotes=false` Flag indicating whether spaces at the beginning or end of a cell are to be ignored if they're not surrounded by quotes. The default is false because spaces are considered part of a field and are not to be ignored according to RFC 4180. Use `true` to enable. Similarly for `Vtable`	Used with `file` only	Optional

Example

```
variantTable VT_ERLANG {
       name "VT_ERLANG"
       characteristics
               ERLANG_LO primary,
               ERLANG_HI primary,
               BLOCKING  primary,
               NUM_LINES
       externalTable 'MY_SAP_ERLANG'
    file "VT_ERLANG.vtable"
}
```

2.1.5.2 Example for Defining a Variant Table

Context

You use this procedure to create variant tables in the Eclipse-based solution modeling environment.

Procedure

1. Choose ▶ *File* › *New* › *Variant Table* ◀.
2. Enter the required data.
3. Choose *Finish*.

   The new variant table is opened in the text editor.

   > i Note
   >
   > You must enter the rows for the new variant table using the text editor. The data can be entered in-line, as shown in the example below, or in a separate table file.

Example

variantTable SME_VIDEO_CARD_EXAMPLE {

name 'SME_VIDEO_CARD_EXAMPLE'

characteristics

SME_VIDEO_CARD_S **primary**,

```
SME_PART_NUMBER_S

rows

"LOW END", "VID1";

"HIGH END", "VID2"

}
```

2.1.5.3 Variant Table Views

Large variant tables, especially if they are designed as external variant tables, can lead to performance degradation of the configuration, due to slower queries. To improve the performance of these large variant tables, it is possible to reduce the scope of the table data by defining views on the table. The view defined on the variant table is applicable within a configuration session.

You can define a view on a variant table using available APIs in the configuration engine. You can invoke these APIs using pFunctions. For more information about this, refer to SAP Note 2509009 (Implement Variant table views).

> **Note**
>
> Defining the variant table views improves performance in both, external and internal, variant tables.

2.1.6 Dependency

Definition

Dependency is the generic term used for constraints and rules. Dependencies are defined within a solution model and specify the relationships between the characteristics and characteristic values of several classes.

Constraints

Constraints are interdependencies between objects and their characteristics. They are a primary form of declarative dependency. All declarative dependencies can be modeled using constraints. You can use them to set characteristic values or to check the consistency of assigned values. Constraints are grouped into constraint nets, which are then assigned to tasks. Constraints are used by the configurator engine to derive or infer any of the following:

- Contradiction
- Value assignment (for characteristics with an ADT, this means linking instances)
- Domain restriction
- Creation and placement of an instance in the PART_OF hierarchy (declarative selection)
- Specialization of an instance
- Relation link between instances for declared relations, when and if this feature becomes available

The configurator engine performs these derivations by applying constraint rules derived from the constraint itself.

Rules

Rules are the primary procedural form of dependency. They act as a point of reference for the classes, characteristics, materials, and variant tables created by the modeler. Rules are the procedural counterpart of constraints used when procedural logic is required, for example, counting or invoking procedural functions (pfunctions).A rule states that some action should be performed when the pattern of the rule is matched in the DDB. It can either be one of the specially provided built-in ones or an existing function (such as built-in APIs or user provided functions).Rules are grouped into rule nets, which are assigned to tasks. The tasks are then assigned to the knowledge bases.In particular, a rule can:

- Perform aggregations
- Access and modify the DDB
- Access the KB
- Trigger front-end functions
- Access and modify external data
- Make logical inferences.

More Information

For information about defining dependencies, see Defining Solution Dependencies [page 44].

2.1.6.1 Defining Solution Dependencies

Use

You use this procedure to create solution dependencies by defining constraints and constraint nets as well as rules and rule nets.

Procedure

Defining Constraints

1. Choose *File* > *New* > *Empty Model File*.
   The system displays a dialog prompting you to choose a folder and enter a file name for the new constraint.
2. Enter the required data.
3. Choose *Finish*.
   The system displays an empty file.
4. Press CTRL + SPACEBAR.
5. Double-click *constraint.*

> **i Note**
>
> You can choose a constraint template if a suitable one is available. Templates are indicated with a green dot in the auto-completion dialog box.

6. Enter the required data and save your constraint.

Defining Constraint Nets

1. Choose *File* › *New* › *Empty Model File*.
   The system displays a dialog prompting you to choose a folder and enter a file name for the new constraint net.
2. Enter the required data.
3. Choose *Finish*.
4. Press CTRL + SPACEBAR.
5. Double-click *constraintNet* or choose a template.
6. Enter the required data and save your constraint net.

Defining Rules

1. Choose *File* › *New* › *Empty Model File*.
   The system displays a dialog prompting you to choose a folder and enter a file name for the new rule.
2. Enter the required data.
3. Choose *Finish*.
4. Press CTRL + SPACEBAR.
5. Double-click *rule* or choose a template.
6. Enter the required data and save your rule.

Defining Rule Nets

1. Choose *File* › *New* › *Empty Model File*.
   The system displays a dialog prompting you to choose a folder and enter a file name for the new rule net.
2. Enter the required data.
3. Choose *Finish*.
4. Press CTRL + SPACEBAR.
5. Double-click *ruleNet* or choose a template.
6. Enter the required data and save your rule net.

2.1.6.2 Defining a Rule

> **i Note**
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

Rules are used when procedural logic is required, for example, counting or invoking procedural functions (pfunctions). To define a rule, use the following syntax:

rule **`identifier`** *{*

name **`"30 bytes descriptive text"`** || **names** EN **`"descriptive English text"`**

, FR **`"descriptive French text"`**

objects: ? **`a`** **is_a (300)** **`class-id`**

where **`?b`** = **`adt-cstic-on-a`**

,? m **is_object (material) (300) (nr=** **`material-id`** **)**

condition:

`?a.cstic-on-a` <|| <=|| =|| >=|| = **`value`** || **specified** || **not specified**

or || **and**

part_of **`(?m,?a)`**

or || **and**

table **`table-name (cstic-id = ?a.domain cstic-on-a)`**

> **i** Note
>
> A table statement in the “condition” section evaluates “True” if at least one row is found and “False” if no row is found.

or || **and**

...see the inline help for a full list of potential functions/content

body: then do: pfunction **`pfunction-id`** **(** *parameters list* **)**

||

then do: **`?a.cstic-on-a ?= value || cstic`**

||

then increment: **`?a.cstic-on-a`** **by** *i*

explanations: **'`any descriptive text`'**

}

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`rule`	Denotes the start of a rule definition encompassed within {...}	-	Required
`identifier`	Rule ID used in the model to reference this rule. Max. length 30 bytes	-	Required

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	-	Optional
`objects`	Keyword indicating the start of the objects section	-	Required
`?a/?m`	A user specified symbol to represent the object. Must start with a question mark (?) but can have any alphanumeric characters after the question mark.	At least one object is required	Required
`is_a (300)`	Keyword indicating the object is a class	-	Required
`class-id`	Class identifier of object	-	Required
`is_object (material) (300) (nr= )`	Keyword indicating the object is a material	-	Required
`material-id`	Material identifier of object	-	Required
`condition`	Keyword indicating the start of the condition. Must be a single condition statement, but can have multiple clauses joined with Boolean operators “and”/ “or”	-	Optional
`condition content`	Expressions in the condition content can include a large range of functions. Use context help (CTRL + SPACE) to view the entire list	-	Optional
`body:`	Keyword indicating the start of the body of the rule	-	Required

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`then do:`	Keyword used if the rule invokes a pfunction or sets a “sticky” default value	Either `then do:` or `then increment:` is required.	Optional
`then increment:`	Keyword used if the rule increments an aggregating characteristic	Either `then do:` or `then increment:` is required.	Optional
`?a.cstic-on-a`	An aggregating characteristic for instance `?a`	Required with `then increment:`	Required with `then increment:`
`by`	Keyword denoting the range by which the increment is made	Required with `then increment:`	Required with `then increment:`
`i`	The number to be added to cstic-on-a Can be a number or the numeric result of a function.	Required with `by`	Required with `by`
`pfunction`	Keyword used to indicate that a pfunction is to be invoked	Allowed only with `then do:`	Optional
`pfunction-name`	Name of pfunction to be invoked	Required with `pfunction`	Required with `pfunction`
`parameters`	List of parameters; any specified parameter must be defined in the pfunction definition. It is not necessary to provide all parameters.	Optional	Optional
`?a.cstic-on-a`	Any characteristic for class a	-	Optional
`?=`	Indicates setting a “sticky default”. Whenever the characteristic has no value assigned, this value will be assigned as its default.	Required with `pfunction`	Required with `pfunction`
`parameters`	List of parameters; must match the parameters in the pfunction definition	Required with `pfunction`	Required with `pfunction`
`explanation`	Keyword indicating the start of the explanation section.	-	Optional

Keyword/User Value	Meaning	Relation to Other Keywords	Required/Optional
`explanation text`	Text entered here is available for debugging conflicts	-	Optional

> **Note**
>
> Non-sticky defaults (assigned in the class definition using the defaultValues keyword) are retained until changed by a user input or constraint. If subsequently reset to no value (“space”), the assigned default is not recovered. The cstic has no value assigned.

Example

```
rule SET_PROD_COLOR {
 objects:
  ?p is_a (300)PRODUCT
 condition:
  ?p.USAGE_TYPE = 'STEALTH'
 body:
  then do:
   pfunction SET_PROD_COLOR_PF (
    PRODUCT = ?p
   )
}
rule COUNT_COMPONENTS {
 objects:
  ?bundle is_a (300)HARDWARE_BUNDLE
      where ?comp = COMPONENTS
 body:
  then increment:
   ?bundle.COMP_COUNT by 1
   )
}
```

2.1.6.3 Defining a Constraint

> **Note**
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*
>
> **Optional keyword**
>
> **`User-specified value`**
>
> `Mutually exclusive keyword`

Constraints assert facts and rely on deductive reasoning. To define a constraint, use the following syntax:

constraint **`identifier`** *{*

name **"30 bytes descriptive text"** || names EN **"descriptive English text"**

, FR **"descriptive French text"**

objects: ? **a** is_a (300) **class-id**

where **?b** = **adt-cstic-on-a**

,? m is_object (material) (300) (nr= **material-id**)

condition:

?a.cstic-on-a <|| <=|| =|| >=|| = *value* || **specified** || **not specified**

or || and

part_of **(?m,?a)**

or || and

table **table-name (cstic-id = ?a.domain cstic-on-a)**

or || and

...see inline help for full list of potential functions/content

restrictions:

asserted facts

> **Note**
>
> For an explanation of the statements that can be used to assert facts, see the "Commands Used in the Restrictions Section of a Constraint" table below. The list, along with detailed syntax, is also available via the inline help for editing a constraint within the solution modeling environment.

inferences:

?a.cstic-on-a

,?m.cstic-on-a

,?a.facet cstic-on-a

explanations: ' **any descriptive text** '

}

Keyword/User Value	Meaning	Required/Optional
`constraint`	Denotes the start of a constraint definition encompasse within {...}	Required
`identifier`	This is the constraint ID used in the model to reference this constraint. Max. length 30 bytes	Required

Keyword/User Value	Meaning	Required/Optional
`name` or `names`	The descriptive, language-dependent text used as a label for this object. If not used, the identifier is used as the name. See the context-sensitive help for the full list of available two-character language identifiers.	Optional
`objects`	Keyword indicating the start of the objects section	Required
`?a/?m`	A user specified symbol to represent the object. Must start with a question mark (?) but can have any alphanumeric characters after the question mark.	Required
`is_a (300)`	Keyword indicating the object is a class	Required
`class-id`	Class identifier of object	Required
`is_object (material) (300) (nr= )`	Keyword indicating the object is a material	
`material-id`	Material identifier of object	
`condition`	Keyword indicating the start of the condition. Must be a single condition statement, but can have multiple clauses joined with Boolean operators “and”/ “or”	Optional
`condition content`	Expressions in the condition content can include a large range of functions. Use context help (CTRL + SPACE) to view the entire list	Optional
`restrictions`	Keyword indicating the start of the restrictions section	Required

Keyword/User Value	Meaning	Required/Optional
`restrictions content`	Restrictions are stated as assertions and, like condition content, can use a variety of functions. Restrictions can assert either the existence of an instance or the value of a characteristic facet	Required
`inferences`	Keyword indicating the start of the objects section	Optional
`inferences content`	Lists the asserted facts to be inferred	Optional
`explanation`	Keyword indicating the start of the explanation section	Optional
`explanation text`	Text entered here is available to help with debugging conflicts	Optional

Commands Used in the Restrictions Section of a Constraint

Command	Use
`false`	To raise a conflict, thereby making the configuration inconsistent
`find_or_create()`	To create a new instance in the configuration
`has_part()`	To add a part to a bill of material
`table()`	To look up values in a variant table
`type_of()`	To specialize a class object
`?class.cstic`	To assert facts about the value of a charactristic “cstic” on an instance of class “class”. `?class` is a reference to an object list in the objects section and `cstic` is a characteristic for that object.
`?class.facet cstic`	To assert facts about a facet of a characteristic “cstic” on an instance of class “class”. Examples of facets are domain, invisible, required, and noinput.

Command	Use
Mathematical functions	A large set of mathematical functions is available. Examples include the following functions: • rounded • sin • cos • tan • floor • log10 For a full list of the available functions, see the inline help.

Example

```
constraint PLACE_FUNIT_A_1 {
 objects:
  ?CA is_a (300)CABINET_A
 ,?FA is_a (300)FUNIT_A
 condition:
  ?FA.INST_NUM = 1
 restrictions:
  ?CA.FU_A = ?FA
 inferences:
  ?CA.FU_A
 explanations:
  "first FUNIT_A goes in CAB_A"
}
```

2.1.6.4 Dependency Net

Definition

A dependency net is a group of constraints or rules (also referred to as dependencies), which are defined within a solution model and specify the relationships between the characteristics and characteristic values of several classes.

Use

Constraints

Constraints are interdependencies between objects and their characteristics. They are used as a dependency in variant configuration. You can use them to set characteristic values or to check the consistency of assigned values. In variant configuration, you can use constraints to address configurable assemblies in a BOM, between which interdependencies exist.

Rules

Rules act as a point of reference for the classes, characteristics, materials, and variant tables created by the modeler. They are used when procedural logic is required, for example, counting or invoking procedural functions (pfunctions). Rules are grouped into rule nets, which are assigned to tasks. The tasks are then assigned to the knowledge bases.

Structure

The dependency net is displayed in the form of a hierarchy in the model graph.

More Information

For information about defining dependency nets, see Defining Solution Dependencies [page 44].

2.1.7 User-Defined Function

Definition

User-defined functions enable you to perform (complex) tasks using external program code. They can be used to check values and infer characteristic values. You may want to create functions for the following purposes, for example:

- Complex calculations based on characteristic values in the configuration
- Complex validity checks for allowed values
- Efficient operations, using side effects, that overcome limitations of the dependency syntax
- Arbitrary calls to external programs, with or without side effects

User-defined functions can be used in constraints and rules. In constraints, you can use functions to check the consistency of the values entered, perform arbitrary complex calculations on inputs to determine output values, and look up values in external systems. In rules, you can use functions to trigger arbitrary processing. When you invoke a user-defined function in a constraint or rule, you are referring to a Java method.

Use

There are two forms of user-defined functions: declarative functions and procedural functions (“pfunctions”). Both declarative functions and pfunctions have the same interface.

Structure

Declarative functions and pfunctions are defined using the following framework:

```
function <identifier> {
  characteristics
    <parameter list>
}
```

They use the following keywords:

Keyword/User Value	Meaning	Required/Optional
`function`	Denotes the start of a function definition	Required
`identifier`	Function ID used in the model to reference this function. Max. length 30 bytes. This must be the name of the class invoked by this function.	Required
`{`	Starts the set of keywords that define this object	Required
`}`	Ends object definition	Required
`characteristics`	Keyword indicating the start of the list of characteristics for this class	Required
`parameters`	List of characteristics that will be provided to the Java class as input. All characteristics listed are required when this function is invoked. i Note No values are returned from a pfunction.	Required
`primary`	Required for at least one parameter. It indicates that the parameter is “input only” to the invoked function. It does **not** indicate that the parameter is required.	Required for at least one parameter

More Information

2.1.7.1 Declarative Function

Definition

Declarative functions are invoked from constraints. In their arguments, they receive data structures representing input and output characteristics. Using these characteristics, they can read the values of the input characteristics in the configuration and they can return values for the output characteristics.

A declarative function is insulated from the actual configuration and can neither influence the value of any characteristic other than one that is passed to it as an output characteristic, nor create any other side effects.

More Information

Defining User-Defined Functions [page 78]

2.1.7.1.1 Example: Defining a Declarative Function

The following example shows how to define, implement, and invoke a declarative function.

Example Framework of a Declarative Function

To define a declarative function, use the following model:

```
function DETERMINE_LABEL_ID {
    characteristics
        CPU primary,
        HD primary,
        LABEL_ID
}
```

The function is implemented in a java class. The java class has to be defined as follows:

- The package of the class must be `com.sap.sce.user`.
- The class name must match the definition name (here: `DETERMINE_LABEL_ID`).
- The class must implement the interface `com.sap.sce.user.sce_user_fn`.

The interface defines the following method that needs to be implemented: `boolean execute(fn_args args, Object obj);`

The method has two parameters:

- `args` contains the sequence of arguments - import arguments with an existing binding and export arguments where the binding is to be set in the method implementation. All characteristics that have been specified as primary in the function definition become import parameters. Other characteristics become output parameters.
- The type of the second parameter `obj` depends on the context in which the function is executed. If this parameter is executed as a declarative function, the parameter refers to the knowledge base object, which provides access to static information such as the knowledge base name, version, and profile.

A declarative function returns a boolean value indicating success or failure. A declarative function invoked in the **condition** part of a constraint, or in the **if** part of a restriction must return a meaningful boolean value because the constraint is testing the return value.

Example Implementation of a Declarative Function

The following source code shows how to implement a declarative function.

```
public class DETERMINE_LABEL_ID implements sce_user_fn {
    public boolean execute(fn_args args, Object kbObj) {
        final sce_user_fn_logging log = new sce_user_fn_logging();

        // retrieve input characteristics
        String inputCPU = args.get_value("CPU");
        String inputHD = args.get_value("HD");

        // determine and set output characteristic
        String outputLabel = String
                .format("CPU: %s, HD: %s", inputCPU, inputHD);
        args.set_value("LABEL_ID", outputLabel);

        log.writeLogDebug(this, "Calculated label is " + outputLabel);

        return true;
    }
}
```

This implementation retrieves the characteristics `CPU` and `HD` from the configuration as input parameters. These two strings are concatenated and stored in the output parameter `LABEL_ID`.

> **Note**
>
> The implementation has only limited access to the current state of the configuration. Only the explicitly defined input and output parameters can be accessed.

Example Invocation of a Declarative Function

The defined declarative function can be used in a constraint as follows:

```
function DETERMINE_LABEL_ID {
        CPU      = ?PC.CPU,
        HD       = ?PC.HD,
         LABEL_ID = ?PC.LABEL_ID
}
```

The characteristics on the left-hand side are characteristics of the function. The characteristics on the right-hand side are characteristics of the instance (referred to with `?PC`) of the configuration.

2.1.7.2 Pfunction

Definition

A “pfunction” (procedural function) provides read and write access to the configuration and dynamic database (in contrast to a declarative function, which simply reads the knowledge base to derive export parameters). Pfunctions can be used only in rules but address all configuration objects.

Pfunctions make all changes as side-effects and are not declarative. No actions are tracked by the TMS (unless tracking is implemented manually). Unlike declarative functions, pfunctions can change the configuration directly. A pfunction can explicitly invoke the execution of dependencies using the “Check” API.

More Information

Defining User-Defined Functions [page 78]

Example: Defining a Pfunction [page 59]

2.1.7.2.1 Testing Pfunctions

Prerequisites

- You have created a knowledge base.
- You have created a launch configuration (see Creating a Launch Configuration [page 77]).

Context

When you open a knowledge base to test it, pfunctions may not be executed as expected. In this case, you can use this procedure to test and debug the pfunctions.

Procedure

1. Open the *Debug* perspective and choose ▶ *Run* ▶ *Debug Configurations...* ▶.
2. Expand the *Eclipse Applications* node.
3. Select a launch configuration.
4. Choose the *Debug* pushbutton.

   The test UI appears in a new window. You can then set breakpoints in the Java program from the *Debug* perspective in the solution modeling environment session.

2.1.7.2.2 Example: Defining a Pfunction

The following example shows how to define, implement, and invoke a pfunction.

Example Framework of a Pfunction

To define a pfunction, use the following model:

```
function NUMBER_OF_ITEMS {
    characteristics
        PC_REF primary,
        HOLDS_HDS primary,
}
```

The function is implemented in a Java class. The Java class has to be defined as follows:

- The package of the class must be `com.sap.sce.user`.
- The class name must match the definition name (here: `NUMBER_OF_ITEMS`).
- The class must implement the interface `com.sap.sce.user.sce_user_fn`.

The interface defines the following method that needs to be implemented: `boolean execute(fn_args args, Object obj);`

The method has two parameters:

- `args` contains the sequence of arguments - import arguments with an existing binding and export arguments where the binding is to be set in the method implementation. All characteristics that have been specified as primary in the function definition become import parameters. Other characteristics become output parameters.
- The type of the second parameter `obj` depends on the context in which the function is executed. If this parameter is executed as a pfunction, the parameter refers to the configuration itself, which can be used to perform changes in the configuration directly.

The boolean typed return parameter is of no use in the context of pfunctions. Whereas the return value of a declarative function carries meaning, the return value of a pfunction should always be true. If the return value is false, this can lead to errors in the engine and an inconsistent configuration.

Note

A function implementation (that is, the Java class) can be invoked as a test or for setting cstic values, and may even be used as the implementation of a pfunction. Such multiple use requires very careful implementation. It is permitted but not necessarily recommended.

Example Implementation of a Pfunction

The following source code shows how to implement a pfunction.

```
public class NUMBER_OF_ITEMS implements sce_user_fn {

  public boolean execute(fn_args args, Object configObj) {
    // retrieve input characteristic PC_REF. PC_REF is an ADT
    // characteristic and therefore can be converted to an ddb_inst
    // object
    ddb_inst instance = (ddb_inst) args.find("PC_REF").kb_get_binding();

    try {
      // retrieve the values of characteristic HOLDS_HDS
    kb_type instType = instance.ddb_get_inst_type();
    kb_cstic csticToCnt = instType.kb_has_cstic_p("HOLDS_HDS");
    read_only_sequence rs = instance.ddb_get_values(csticToCnt);

    // set the value of characteristic NUMBER_OF_HDS
    instance.ddb_set_or_replace_val(
        instType.kb_has_cstic_p("NUMBER_OF_HDS"),
        float_value_imp.get_float_value(rs.length()),
        ((cfg_imp) configObj).tms_get_generic_default_owner());

    } catch (Exception e) {
      final sce_user_fn_logging log = new sce_user_fn_logging();
      log.writeLogError(this, e.getMessage());
    }

    return true;
  }
}
```

This implementation counts the number of assigned values to characteristic `HOLDS_HDS`. The retrieved number is assigned to characteristic `NUMBER_OF_HDS`.

The implementation has access to the complete state of the configuration. Argument `configObj` is an instance of class `cfg_imp` that can be used to modify the state of the configuration in a non-declarative, procedural way. In the example, the characteristic `NUMBER_OF_HDS` is modified that is not part of the argument list of the pfunction.

The class `com.sap.sce.user.scelib` provides utility methods that can be used by pfunctions to read and write from/to the current configuration. Among others, this class provides the following functionality:

Method	Purpose
`scelib.get_value(inst, csticName)`	Reads assigned value of `<csticName>`

Method	Purpose
`scelib.get_values(inst, csticName)`	Reads assigned values of the multi-valued characteristic `<csticName>`
`scelib.set_value(inst, csticName, val)`	Sets value `<val>` to `<csticName>`
`scelib.vt_select_buffered(kBase, match, VTabName, condition, order)`	Selects lines of variant table `<VTabName>` that obey the `<condition>` , ordered by `<order>`
`scelib.get_domain(inst, csticName)`	Reads current domain of `<csticName>`
`scelib.restrict_dom(inst, csticName, dom, owner)`	Restricts the domain of `<csticName>` by the string-array `<dom>`

Example Invocation of a Pfunction

Pfunctions are invoked in the same way as declarative functions, for example:

```
function NUMBER_OF_ITEMS {
        PC_REF = ?PC.PC_REF
}
```

Pfunctions can be invoked only in the body of rules.

2.1.8 Interface Design

An interface design is an object that defines characteristic groups and assigns characteristics to those groups. The configuration UI uses this information to present characteristics on the screen. Interface design objects can be associated with classes or materials allowing different presentations of the characteristics to be displayed based on the class or material.

Interface designs are inherited by subclasses and can be overridden by superclasses. If a subclass or material extends (inherits from) more than one superclass with an interface design, the subclass **must** define its own interface design.

2.1.8.1 Defining an Interface Design

> i Note
>
> This description uses the following conventions to illustrate the syntax requirements:
>
> *Required keyword*

Optional keyword

`User-specified value`

`Mutually exclusive keyword`

To define an interface design, use the following syntax:

interfaceDesign **`identifier-1`** *{*

group **`identifier-A`** **{**

characteristics

`characteristic-id, characteristic-id,`

urls { **`"url"`** *label* **`"30 bytes descriptive text"`** **,** **`"url"`** *label* **`"30 bytes descriptive text"...`** *}*

}

group **`identifier-B`** **{**

characteristics

`characteristic-id, characteristic-id,`

urls { **`"url"`** *label* **`"30 bytes descriptive text"`** **,** **`"url"`** *label* **`"30 bytes descriptive text"...`** *}*

}

}

Keyword/User Value	Meaning	Required/Optional
`interfaceDesign`	Denotes the start of an interface design object encompassed within {...}	Required
`identifier-1`	This is the interface design ID used in the model to reference this interface design. Max. length 18 bytes.	Required
`group`	Denotes the start of a group definition	Required
`identifier-A`	This is the group ID, the technical name of the group of characteristics. Max. length 18 bytes.	Required
`name`	This is the group name used to name/ label the grouping on the UI. For example, if the UI presents groups as tabs, this is the tab label if no name is provided. Max. length 30 bytes.	Optional

Keyword/User Value	Meaning	Required/Optional
`characteristics`	Indicates the start of the list of characteristics in this group	Required
`cstic-1`	A valid characteristic name	Required
`urls`	Keyword associating a URL with this group. URLs are not accessible via the engine API. We suggest that you do not use them in this release.	Optional
`label`	A label for the URL	Optional

2.1.9 Bill of Material

Definition

A bill of material (BOM) is a list of materials, assemblies, components, and parts required to manufacture a product. These can be used to define simple relationships where one component comprises several subcomponents. If required, you can define a maximum and a minimum quantity for each material and class in the BOM.

More Information

For information about defining bills of material, see Defining Dynamic Bills of Material [page 63].

2.1.9.1 Defining Dynamic Bills of Material

Context

You use this procedure to create new bills of material (BOMs) in the Eclipse-based solution modeling environment.

Procedure

1. Choose *File* > *New* > *Empty Model File*

   The system displays a dialog prompting you to choose a folder and enter a file name for the new BOM.
2. Enter the required data.
3. Choose *Finish*.

   An empty file is displayed in the text editor.
4. Press CTRL + SPACEBAR and double-click *bom*.

   > **i Note**
   >
   > Alternatively you can type **`bom.`**
5. Enter the required data and save the new BOM.

   > **i Note**
   >
   > Optionally you can specify a minimum and a maximum value for each material or class.

Example

The bill of material shown below comprises one class and two materials:

```
bom SME_WORKPLACE {
10 class SME_COMPUTER min 0 max 9999,
20 material SME_DESKTOP min 0 max 9999,
30 material SME_DESKTOP min 0 max 9999
}
```

2.1.10 Model Syntax and Logical Validations

SME not only performs language syntax validation but extends it further to also do logical validations. These logical validations are performed at compile time which helps in providing feedback during the model development phase itself, thus avoiding the expensive test cycle process. These logical validations include:

- **Using default assignment operator “?=” is not allowed in constraint restriction.**
  The default assignment can be used in rules for assigning the default values, however, the same is not true for constraints. If it used in constraints, an error is thrown.
- **Warning for free variables in constraint declaration.**
  A free variable in a constraint is defined in the object section of a constraint, but is never used in the condition or restriction section. Such variables have an impact on the execution of the constraint. A warning message is shown in such scenarios. Refer to SAP Note 2877744 for more details.
- **Validation check in find_or_create**

If find_or_create is used in the restriction section using the‘with’ argument, SME does a logical check if the characteristics used in the ‘with’ argument are defined in a class, else a validation error is shown.

- **Validation check on assigning domain in constraint restriction**
  In a constraint, if a characteristic domain is assigned in the restriction section, then that characteristic should be of restrictable type ,else an error is shown.

2.1.11 Solution Model Samples

The Solution Modeling Environment is shipped with a few sample solution models that serve as points of reference when you are building live solution models.

For more information about creating live projects based on these samples, refer to the link below.

Related Information

Creating a Solution Sales Configuration Project [page 74]

2.1.11.1 KBO with MCI

SAP Solution Sales Configuration supports the application of predetermined rules to orchestrate between knowledge bases with multi-configuration instances (MCI). This sample model provides insight into how multi configuration instances can be created and managed across multiple configurations through the knowledge base orchestration process.

For more information about this, refer to SAP Note 2550572 (SAP SSC Knowledge Base Orchestration (KBO) and Multi Configuration Instances (MCI)).

2.1.11.2 FBS_SSC_CA

A model to demonstrate how the hardware, software, and services can be packaged together and how guided selling can be enabled.

The FBS_SSC_CA model is intended to configure a data center and its supply servers, supporting an application with various user types. Based on the number of each user type specified, the model will generate the number of servers required, with the appropriate memory, CPU speed, and number of CPU’s.

The model also allows the user to override recommended values.

Keywords and Key Concepts

Keywords	Description
Counting	Tallying, counting, and accumulating values
ADT's	Use of abstract data type pointers to represent relationships between solution components
Non-part instances	Adding content without using a BoM (Bill of Materials)
Guided selling	Rudimentary guided selling is illustrated here

Overriding Default Values in the System

1. Specify the number of users for one or more user type.
   The model provides a list of potential servers as per the requirements of the relevant user type.
2. Select the relevant server name.
   The model generates the required number of servers.
   Up to fourteen individual servers are contained in a single blade system. Each individual server is further configured with the appropriate memory and CPU's. You need to simply enter the number of users and a selection from a set of potential servers. The model executes the process automatically. If required, you may also override the default system behavior.
   For more information about this, refer to **Memory Override**.
3. Select additional software and/or services.
   These can be associated with the servers by selecting the servers listed in the *Runs on HW* characteristic in software instances or in the *Serves* characteristic on services instances. The association can also be made from the hardware instances by using the *Runs SW* and *Served by* characteristics.
   The model updates the line item quantity for the software and service instances based on the number of servers associated with them. For example, if the software runs on 15 servers, the line item quantity is set to 15. This process is controlled by the model and can also be configured otherwise.
   The user can override the memory values on any blade system by setting the *SPEC_MEM* characteristic to 'User' from the system default 'System'. When set to 'User', the memory field is available for update. You can specify a new value to allocate that memory across all the servers in this specific blade system. There is no impact on other blade systems in the configuration.

Overriding Other Default Behavior

The model was defined to automatically link all software and services to all server instances. This behavior of the model can be overridden by changing the *Auto Assign* characteristic in any of the following instances:

- Server
- Software
- Service

When the settings are changed, assignments are withdrawn and can be reassigned manually.

Related Information

Description of Modeling Techniques [page 67]

2.1.11.2.1 Description of Modeling Techniques

Related Information

Guided Selling Questionnaire [page 67]
Selecting Servers [page 68]
Adding Software and Services [page 69]
Linking Services, Hardware, and Software [page 70]
Counting and Setting a Line Item Quantity [page 70]

2.1.11.2.1.1 Guided Selling Questionnaire

The root instance of the configuration contains a set of prompts/questions through which you can specify your business needs. In this case, a single question about user volume (number of users per user type) gathers all the required information for the model, based on which the model determines the types of appropriate servers.

The user input for this question is supplied to the following fields:

- `STD_SELFSERV_USERS`
- `SCM_SALES_USERS`

Since both the fields work in a similar fashion, you can refer to the properties of `STD_SELFSERV_USERS` for further information.

Right click on the field and select *Find References* to find the constraints in which it is used:

- `DEMAND_STD_SELF`
  Here, the field `STD_SELFSERV_USERS` is used to compute the *Demand*, that is, the number of CPUs, CPU speed, and RAM required to support the type and number of users specified.
- `USER_MASTER_DATA`
  Here, the field `STD_SELFSERV_USERS` is used to create an instance in the configuration to hold data specific to a user type. In this constraint, you can see the `find_or_create` command and right click on the class name `USER_MASTER_DATA` to find references to it. The `GET_USER_MASTER_DATA` constraint pulls data from a variant table and stores it in the `USER_MASTER_DATA` instance.

Hence, the two constraints work together to retrieve data for a particular user type for which users have been specified. In other constraints, if user specific data is needed, the `User_Master_Data` instance is referenced.

- `VISIBLE_INST`
  Here, the field `STD_SELFSERV_USERS` is used to control visibility of the two other fields. If no value is specified, the list of server names is hidden along with the list of pointers to servers that are already added to the configuration.

2.1.11.2.1.2 Selecting Servers

The field containing the list of servers becomes visible after the number of users is entered in either or both the *Number of Users* fields. These lists of values are stored in form of variant tables and can be accessed through rules. This is discussed in more detail below:

`STD_SELF_SERVER_NAME` is the selection field for the server name and is restricted by the following constraints:

- `VISIBLE_INST`
  This constraint is discussed in the chapter **Guided Selling Questionnaire**.
- `SERVER_MASTER_DATA`
  This constraint uses the same techniques as is used for `USER_DATA`.
- `DMN_SERVER_AND_CPU_SPD`
  This constraint restricts the domain of both *Cpu_Speed* and *Server_Name* values by using a field computed in constraint `DEMAND_STD_SELF` based on *User_type*. Only those CPU speeds whose values are greater than or equal to the speed required for the user type:

  ```
  ?S.domain ACTL_STD_SELF_CPU_SPD >= ?S.REQ_STD_SELF_CPU_SPD
  ```

  The constraint further uses the restricted domain to restrict the allowed server name values using a variant table:

  ```
  table T_SERVER_DATA
                (SERVER_NAME = ?S.domain STD_SELF_SERVER_NAME
                 ,PROCESSOR_SPD = ?S.domain ACTL_STD_SELF_CPU_SPD
                        )
  ```
- `COMPUTE_REQ_SERVERS`
  This constraint computes the number of servers required to meet the memory demand and the required number of CPUs. It then compares these values and retrieves the number of servers as the greater of these two values.
- `STD_SELF_INST_1` and `STD_SELF_INST_N`
  These constraints work together to create the required number of servers, as discussed above.
  The following conditions are considered by the constraints for the calculations:
  - `STD_SELF_INST_1`: If it is true that there is at least 1 (more than 0) server required, then it is also true that there is a server with instance number = 1.
  - `STD_SELF_INST_N`: If it is true that many servers are required and a server exists with an instance number < the number required, then it is also true that there is a server with instance number 1 greater than the existing one.
    To illustrate the above, assume 3 servers are required.

1. STD_SELF_INST_1 will create a server instance with SSC_INSTANCE_NUM = 1.
2. STD_SELF_INST_N executes as its objects pattern is matched.
3. The number of servers required is 3 and a server instance exists with instance_num =1 (which is < 3).
4. It confirms that there is another server instance with SSC_INSTANCE_NUM one greater than the one found.
5. This constraint runs for each new server instance created, except for the one whose instance_num = 3 (only for instances with instance_num < 3).

Related Information

Guided Selling Questionnaire [page 67]

2.1.11.2.1.3 Adding Software and Services

Software is added by selecting values in field *SOFTWARE_SELECT* and services are added by selecting values either in *INST_SERVICE_SELECT* (installation services) or *MAINT_SERVICE_SELECT* (maintenance services).

Each of these are multivalued fields.

The constraint CREATE_SERVICE_INST adds service instances and will run once for each value in the multivalued field. Using the values in the field to set a characteristic value on the service instance ensures that each instance is unique.

```
constraint CREATE_SERVICE_INST {
          objects:
          ?S is_a (300) FBS_SSC_CA
         condition:
          ?S.SERVICE_SELECT specified
         restrictions:
           find_or_create
            ((300) SERVICE,
             with SERVICE_PROFILE = ?S.SERVICE_SELECT;
                 IS_PART_OF_SD_SOFT = ?S;
                 SERVICE_IN_SOL_ADT = ?S)
         explanations:
           "CREATE service instance. multiValue will find_create for each value."
        }
```

Note

If the SERVICE_PROFILE is not set on the service instance in the find_or_create statement. In this case, only one service instance would be created and it would be justified in the engine's truth maintenance system, once for each value in the *SERVICE_SELECT* field.

A similar technique is used for adding software instances.

2.1.11.2.1.4 Linking Services, Hardware, and Software

While each item added to the solution is a component of the solution, there are other relationships between the components. These relationships are expressed by using ADT's that function as pointers to other instances and are restricted by the following constraints:

- LINK_SW_HW, LINK_HW_SW
- LINK_HW_SV, LINK_SV_HW
- LINK_SV_SW LINK_SW_SV
  Each of the "LINK_..." constraints ensure that if item A points to item B, then item B also points back to item A. For example, `LINK_SW_HW` ensures that if a software is run on a server, then it is also true that the server runs that software.
- AUTO_ASSIGN_SW and AUTO_ASSIGN_SV
  These constraints use a switch (`AUTO_ASSIGN_TXT`) to determine whether the system should automatically establish these relationships, that is, assume that all software runs on all hardware, all software and hardware are 'serviced by' all services in the configuration, or whether no such relationships are assumed. If no relationships are assumed, then the user may assign the relationships manually by making selections in the ADT characteristic fields.

2.1.11.2.1.5 Counting and Setting a Line Item Quantity

The model includes a reference characteristic that allows the model to reference information that exists on the sales document, such as customer number, country, sales organization, etc. and for the model to set values of fields on the sales document, such as line item quantity. The reference characteristic in this model, `STOP_MENGE`, is used to set the line item quantity.

```
characteristic STPO_MENGE {
            names
                  EN 'Component quantity',
                  DE 'Komponentenmenge'
            numericLength 13 decimalPlaces 3
            negativeValues
            reference table 'STPO' field 'MENGE'
        }
```

To illustrate this, assume that a software's quantity depends on the number of things it runs on.

So, if 1 server runs the software, item quantity is 1; but if it runs on 10 servers, then the quantity is 10. A similar logic applies to service items.

For more information about how the number of software licenses is counted, see rule `AGGR_SW_LIC` and constraint `SET_SOFTWARE_STPO_MENGE` to understand how that value is assigned to the reference characteristic.

2.2 Setup of Solution Modeling Environment

Use

You use this process to define a solution model in the Eclipse-based solution modeling environment, test it, and transport it to a target system for use in the sales ordering process.

You perform these steps in Eclipse, using the *SAP Modeling* perspective. To change to a different perspective, choose *Window* > *Open Perspective* > *Other...*

> ℹ Note
>
> - During SME installation, the system on which Eclipse is running must have internet connectivity. Eclipse uses the default internet connectivity configured by proxy settings on OS level (for example, in *Windows* > *Internet Options* > *Connections* > *LAN Settings* or any other OS connectivity settings). If no internet connectivity is configured, you must configure the proxy settings in Eclipse under *Windows* > *Preferences* > *General* > *Network Connection*, as part of the standard Eclipse set up procedure.
> - If you are working as part of a team and are using a source code management system, you must first check out a file for editing, and then check it in after completing editing.
>   For more information, see chapter **Collaboration Between Modelers**.

Prerequisites

- You have configured the connection to the target system.
- You have imported the configuration master data (see Importing Configuration Master Data in the Solution Modeling Environment [page 100]).
- You have maintained your model templates (see Maintaining Modeling Templates [page 108]).

Process

1. You create or open the project for the solution model (see Creating Model Projects [page 74]).
2. You define the model master data, including characteristics and classes (see Defining Characteristics [page 24] and Defining Classes [page 15]).
3. You define materials (see Defining Materials [page 20]).
4. You define variant tables (see Defining Variant Tables [page 42]).
5. You define dynamic bills of material (BOMs) (see Defining Dynamic Bills of Material [page 63]).
6. You define dependencies and dependency nets (see Defining Solution Dependencies [page 44]).
7. You define a knowledge base (see Defining Knowledge Bases [page 10]).
8. You test the model locally (see Testing Models Locally [page 90]).
9. You export the model to the target system (See Exporting a Knowledge Base Runtime Version [page 82]).

Result

Your solution model is available for solution configuration in the target system.

Related Information

Collaboration Between Modelers [page 109]

2.2.1 SAP Modeling Perspective

The SAP Modeling perspective features a number of views that allow you to represent your solution model in different ways:

Component	Description
Project Explorer	The Project Explorer view allows you to explore the project as it exists in the file system, that is, as a set of folders and files. New files can be added, changed, or deleted just as they are in Windows Explorer. Updates here are also made in the local workspace. Adding a folder here, for example, adds a folder on the local disk in your workplace and deleting files removes them from the local disk. The operations available are shown in the menus or by right-clicking an object (context menu) in the Project Explorer. All operations, with the exception of those in the *SAP Modeling* menu, are standard Eclipse functions. For more information, see the Eclipse documentation.
Model Explorer	The Model Explorer view allows you to explore the project as it is seen by SAP Solution Sales Configuration, that is, as a set of objects - characteristics, classes, constraints, and so on. i Note Objects can be opened and edited from this view, but not added or deleted.

Component	Description
Model Graph	The Model Graph view shows the relationship between classes or dependencies in a graphical view. It is useful for understanding the relationships among classes and materials, or between dependencies. To view the class hierarchy, use the donut icon; to view the materials, click the hierarchy icon; to view the dependency structure, click the cube icon. When you set *Link to Editor*, the editor repositions to the definition of the object selected in the graph and the graph highlights the object based on the object definition being edited. The scope of objects shown in the graph can be controlled in two ways. First, in the configuration menu (which is accessible using the upside down triangle icon), choose *Workspace* to see objects from all open projects in your workspace, or choose *Selected Project* to see only objects in the project currently selected. You can also choose *Selected Project (Incl. References)* to see objects in the selected project and all of its open reference projects. The second way to control the scope of display in the Model Graph view is to right-click any of the displayed objects and set a filter. Setting a filter limits the view to this object and any subordinate objects. **i Note** Objects can be opened and edited from this view, but not added or deleted. If you open the file in which an object is defined and set *Link to Editor*, the view will reposition to the definition of the object selected in the graph, and the graph will reposition (highlight) to the object based on the object definition being edited.
Problems	The Problems view shows messages from invoked functions, such as *Export Knowledge Base* or *Validate Model*. Double-click a message to go to the error location.
Search	The Search view shows the results of a search. You can also initiate a search from the Search view or the *Search* menu. The yellow up and down arrows locate the next or previous match; the plus (+) and minus (–) icons expand or collapse all entries in the results tree. The refresh icon reruns the current search, and the search history icon shows a list of previous searches, which can be selected and rerun. The downward triangle icon can be used to change the view from a tree to a list, set a filter, and update general search preferences through a link.

Component	Description
Outline	The Outline view works in tandem with the editor. It shows all the objects defined within the active file in the editor. You can also sort the object definitions in this view.

2.2.1.1 Creating Model Projects

Use

You use this procedure to create a new SAP Solution Sales Configuration model project in the Eclipse-based solution modeling environment.

Procedure

> **i Note**
>
> Creating an empty model file is a quick way to start a new object definition. You can then use the context help to complete the definition.

1. Choose ▶ *File* › *New* › *SAP CPQ for Solution Sales Configuration Model Project* ◀.
2. Enter the project name and location.
3. Choose *Finish*.
   The new project is displayed in the *Project Explorer*.

2.2.1.1.1 Creating a Solution Sales Configuration Project

You use this procedure to create a solution configuration model from an example in the Eclipse-based solution modeling environment

Creating an SSC Model Project

1. Choose ▶ *File* › *New* › *Other* ◀
2. In the *Select a Wizard* window, expand the *SAP CPQ for Solution Sales Configuration* node and select *SAP CPQ for Solution Sales Configuration Model Project*
3. Choose *Next*
4. In the next window, enter the project name and a location for the project

5. Choose *Finish*

Result

You can see the new project in the *Project Explorer*.

Creating an Example SSC Model Project

1. Choose ▶ *File* › *New* › *Other* ◀
2. In the *Select a wizard* window, select *Examples*
   From the options, select the sample project you want to include in the workspace and click on *Next*
3. Click on *Finish*

Result

You can see the sample project in the *Project Explorer*.

2.2.1.1.2 Importing a Project into Your Workspace

Use

As a modeler, you can work with several workspaces, each with its own logically distinct set of projects. In some cases, you might need to copy a project from one workspace to another, or you might want to work on a model created by another colleague. You can do this by adding the project to your workspace.

Procedure

1. Right-click the white space of the *Project Explorer* view.
2. Choose *Import*.
3. In the *Import* window, open the *General* folder and choose *Existing Projects into Workspace*.
4. Choose *Next*.
5. Decide whether you want to import the project from the folder structure on your file system or from an archive file.

Importing from the File System

1. Choose the *Select Root Directory:* radio button.
2. Select the root directory containing the project(s) using the *Browse...* pushbutton.
3. Select the project to be imported.

> **i Note**
>
> Projects that are grayed out cannot be selected because they already exist in your workspace.

4. To make a copy of this project and copy it to your workspace, choose *Copy Projects into Workspace*. If you want to work on the files in the current location, do not choose this option.
5. Choose *Finish*.
   The project then appears in the *Project Explorer*.

Importing from an Archive File

1. Choose the *Select Archive File:* radio button and use the *Browse...* pushbutton to select the archive file containing the project.
2. Select the project to be imported. The *Copy Projects into Workspace* option is selected by default and is required.

   > i Note
   >
   > Projects that are grayed out cannot be selected because they already exist in your workspace.

3. Choose *Finish*.
   The project then appears in the *Project Explorer*.

2.2.1.2 Adding Reference Projects to an Existing Project

Use

It's common practice (and even a best practice) to create one or more projects containing frequently used routines that can be reused in other model projects. To reuse content from another project, you designate the project as a “reference project”. The solution modeling environment then treats the content as if it were included in a single project.

> i Note
>
> If any one of the following objects:
>
> - Class
> - Material
> - Characteristic
> - Constraint
> - Rule
> - Variant table
> - Constraint net
> - Rule net
> - Pfunction
>
> is defined in more than one project, it is flagged as a duplicate error. Reference projects allow you to reuse objects without having to define them more than once.

Procedure

You can view reference projects using the following procedure:

1. In the *Import* window, open the *General* folder and choose *File System*.
   Click on *Next*.
2. Specify the folder(s) and file(s) to be imported, using the *Browse* button.
   Click on *Finish*.

2.2.1.3 Exporting a Model Project

Context

You use this procedure to export a solution model project so that you can share it with your colleagues.

Procedure

1. Right-click the project folder in the *Project Explorer* view.
2. Choose *Export*.
3. In the *Export* window, open the *General* folder and choose *Archive File*.
4. Choose *Next*.
5. Choose the *Select All* pushbutton and specify the location and name of the archive file to be created.
6. Under *Options*, choose the options for the file format, directory structure, and compression.
7. Choose *Finish*.

   The project is then exported to the file you specified.

2.2.1.4 Creating a Launch Configuration

Context

You use this procedure to create a launch configuration from either the Java or Debug perspective.

Procedure

1. Choose *Open Perspective* > *Java Perspective*.
2. Choose *Run* > *Run Configurations...* or *Run* > *Debug Configurations*.
3. Expand the *Eclipse Application* node and enter a name for the configuration.

   Click on *Apply* and then *Run*.

Results

The new configuration is ready to be launched.

2.2.1.5 Defining User-Defined Functions

Use

You use this procedure to create user-defined functions in the Eclipse-based solution modeling environment.

Procedure

1. Choose *File* > *New* > *Empty Model File*
   The system displays a dialog box, prompting you to select a folder and enter a file name for the new function.
2. Enter the required data.
3. Choose *Finish*.
   An empty file is displayed in the text editor.
4. Press CTRL + SPACEBAR and then double-click *function*.
5. Enter the required data and save the new function.

> **i Note**
>
> The compiled Java class must be added to the class path of the EJB-IPC bundles (`com.sap.custdev.projects.fbs.slc.ejb-ipc`). This can be done using a fragment or the functionality `Eclipse-RegisterBuddy: com.sap.custdev.projects.fbs.slc.ejb-ipc`.
>
> The new fragment or plug-in with the function can be handed over to the Eclipse environment itself (MyEclipse\plugins or MyEclipse\dropins), or you can start the test UI in Eclipse in a runtime configuration, which contains all the necessary bundles.
>
> For more information about this, see SAP Note 1701098 (Documentation Update for Implementation of Variant Functions).

Example

```
function FUNCTION {
name "Name"
characteristics
CSTIC1 primary,
CSTIC2,
}
```

Related Information

Declarative Function [page 56]

Pfunction [page 58]

2.2.1.6 Exporting a Project

The Solution Modeling Environment allows you to export knowledge bases to different databases as well as a file to the system.

To make this work, you would need to setup different database connections and the required drivers for the database to which you are exporting the project, to allow successful exports.

For connections to local databases, the following factors are considered:

- MsSQL
- MySQL
- HANA
- Oracle
- MaxDB

Apart from the local databases, knowledge bases can also be exported to the *S/4HANA* systems.

2.2.1.6.1 Setup of Local Database Connection

Context

The SME can create connections to the following local databases:

- MsSQL
- MySQL

- HANA
- Oracle
- SAP MaxDB

You can setup a connection to the local database using the procedure below:

Procedure

1. Go to *Eclipse* > *Windows* > *Preferences* > *Expand SAP CPQ for Solution Sales* > *Connections*.
2. Select the *Add* > *Name*.

   Select the relevant radio button.
3. Select the *Database Type*.
4. Enter the *Server Name* and *Database Name*.
5. Enter the port and client, if this information is not populated automatically.
6. Enter the login name for the database.
7. You can add the drivers using the procedure below:
   a. Go to *Eclipse* > *Windows* > *Preferences* > *Expand SAP CPQ for Solution Sales*.
   b. Add the driver pertaining to the database connection that is being added.

      Click on *OK*.

Example

You can further refer to the example below for more information:

1. Select Microsoft SQL Server as the *Database Type*.
2. Set localhost as the *Server*.
3. Enter MsSQL Database as the *Database Name*.
   This value is user defined.
4. The port and client should be auto-populated as **1433** and **000**, respectively.
5. Enter *sa* as the login name.
   This value is provided during the DB creation process.
6. Add *Microsoft SQL Server JDBC4 Driver* in the *Driver* section.
   Click on *OK*.

2.2.1.6.2 Setup of CRM Connection

Context

You can use the following procedure to export a project to CRM:

Procedure

1. Go to *Eclipse* > *Windows* > *Preferences* > *Expand SAP CPQ for Solution Sales* > *Connections*.
2. Select the CRM radio button to add a name for the project.
3. The name will be user defined and relevant for the CRM connection.
4. Add the CRM *System Number* into the consideration for the connection.
5. Under the *Application Server*, enter the server details from the properties section of the CRM system.
6. Enter the *Client* number for the system.
7. Enter the login name defined above to connect to the CRM system.
8. You can refer to the example below for more information:
    a. *Application Name* = A valid name
    b. *System Number* = 11
    c. *Application Server* = Server details from the properties section
    d. *Client* = 700
    e. *Login Name* = UNIT_TEST

    > i Note
    >
    > No drivers are required for this connection.

2.2.1.6.3 Setup of ECC and SAP S/4HANA Connection

Context

You can export a project using the following procedure:

Procedure

1. Go to *Eclipse* > *Windows* > *Preferences* > *Expand SAP CPQ for Solution Sales* > *Connections*.
2. Select the ECC radio button to enter the relevant name.
3. Add a *Name* to the S/4HANA connection.

   This name is user defined.
4. Add the *System Number* of the S/4HANA system in the consideration for the connection.
5. Under the *Application Server*, enter the server details from the properties of the S/4HANA system.
6. Enter the *Client* number of the system.
7. Enter the login name to connect to the S/4HANA system.
8. You can refer to the example below for further information:
   a. *Application name* = A valid name
   b. *System number* = 11
   c. *Application server* = Server details from the properties section
   d. *Client* = 700
   e. *Login name* = UNIT_TEST

   > i Note
   >
   > No drivers are required for this connection.

2.2.1.6.4 Exporting a Knowledge Base to Database

Use

You use this procedure to export a knowledge base and create a runtime version from it.

You can export a knowledge base from the *File* menu, from the *Project Explorer* view, or from the *Model Explorer* view.

Procedure

From the File Menu

1. Choose *File* > *Export* > *SAP CPQ for Solution Sales Configuration* > *Export Knowledge Base*
2. Select the knowledge base you want to export
3. Choose *Next*
4. Select the connection for the target database
5. Optionally, you can also do the following:
   - Deactivate the validation option in the *Export Knowledge Bases* (multiple KB export) dialog

     > ⚠ Caution
     >
     > If you decide to deactivate the validation, the status of the data in the back end cannot be guaranteed after the export.

   - Activate the *Include local variant table content during export* option
6. Enter the password and choose *Finish*.

From the Project Explorer View

1. Right-click anywhere in the *Project Explorer* and choose *Export* > *SAP CPQ for Solution Sales Configuration* > *Export Knowledge Base*.
2. Select the knowledge base you want to export.
3. Optional in the *Export Knowledge Bases* dialog (multiple kb export dialog): Deactivate the validation option.

   > ⚠ Caution
   >
   > If you decide to deactivate the validation, the status of the data in the back end cannot be guaranteed after the export.

4. Choose *Next*.
5. Select the connection for the target database.
6. Enter the password.
7. Optional: Activate the *Include local variant table content during export* option.
8. Choose *Finish*.

From the Model Explorer View

1. Right-click the knowledge base definition and choose *Export Knowledge Base*.
2. Select the connection for the target database.
3. Enter the password and choose *Finish*.

> i Note
>
> Every time you export a knowledge base, the system asks you whether you want to start a test session in the testing perspective.

> ⚠ Caution
>
> If a validation error is detected in any of the dependent `.ssc` files when you export a knowledge base (for example, due to unrelated `class`, `cstic`, `material`, or `variant` tables present in that file, an error will be thrown that prevents you from exporting the knowledge base.

2.2.1.6.5 Exporting a Knowledge Base to a File

Follow this process to export a knowledge base from the workspace to a local folder.

Context

This procedure exports all the `.xml` files to the local folder of the knowledge base. You can export a knowledge base from the *File* menu either from the *Project Explorer* or the *Model Explorer* view.

Procedure

1. Setup the file connection.
    a. Go to ▶ *Eclipse* › *Windows* › *Preferences* › *Expand SAP CPQ for Solution Sales* › *Connections* ◀.
    b. Select *Add Name*.
    c. Select the relevant radio button for *File*.
    d. Provide the path to the selected local folder.
    e. Click on *OK*.
2. Export the knowledge base.
    - **Exporting from the file menu**
        1. Choose ▶ *File* › *Export* › *SAP CPQ for Solution Sales Configuration* › *Export Knowledge Base* ◀.
        2. Select the knowledge base you want to export.
        3. Optionally, you can also deactivate the option for validation in the *Export Knowledge Bases* dialog for multiple knowledge base export.
        4. Choose *Next*.
        5. Select the file connection created for the export.
        6. After the export is completed, the local folder should contain all the relevant `.xml` files of the project.
    - **Exporting the project explorer view**
        1. Right click anywhere in the project explorer and choose ▶ *Export* › *SAP CPQ for Solution Sales Configuration* › *Export Knowledge Base* ◀.
        2. Select the knowledge base you want to export.
        3. Optionally, you can also deactivate the option for validation in the *Export Knowledge Bases* dialog for multiple knowledge base export.
        4. Choose *Next*.
        5. Select the file connection created for export.
        6. After the export is completed, the local folder should contain all the relevant `.xml` files of the project.
    - **Exporting form the model explorer view**
        1. Right click on the knowledge base definition and choose *Export knowledge Base*.
        2. Select the file connection created for the export.

3. After the export is completed, the local folder should contain all the relevant `.xml` files of the project.

2.2.1.7 KB Admin Tool Support

2.2.1.7.1 Uploading Knowledge Bases

You use this procedure to upload a knowledge base into the local database. On the modeling user interface (UI), you can upload the knowledge base using the *Upload Knowledge Base* menu. You can upload the knowledge base as either flat files or xml files.

Procedure

1. Go to ▶ *SAP Modeling* ▶ *Upload Knowledge Base* ◀.
2. Choose the type of file that you want to upload.

Option	Description
Flat file	Choose *Flat File Upload* and browse for the knowledge base flat-file directory.
XML file	Choose *Xml File Upload* and browse for the knowledge base XML-file directory.

3. Choose *Next*.
4. Choose the connection and enter the required details.
5. Choose *Finish*.

More Information

While downloading a runtime version via `CU36` utility, some text formatting may be lost and replaced by hashes.

For more information about handling this issue, refer to SAP Note 2339258 (Japanese and Chinese texts are lost in runtime version.).

2.2.1.7.2 Deleting Knowledge Bases

You can use the *Delete Knowledge Base* option in the Solution Modeling Environment to delete knowledge bases from supported databases.

Procedure

1. Go to *SAP Modeling* > *Delete Knowledge Base*.
2. Select the relevant connection details and click *Next*.
3. Select the knowledge bases that you want to delete and click *Finish*.
4. Click *OK* to continue.

Results

The selected knowledge bases are deleted from the database and a confirmation message is displayed.

2.2.1.7.3 Uploading External Variant Tables

Context

You use this procedure to upload external variant tables to local databases that can be downloaded from the S/4HANA system.

> i Note
>
> Many models require the support of these external variant tables.

Procedure

1. Choose *File* > *Upload External Variant Tables*.
2. Browse the *External Variant Table* directory.
3. Choose *Next*.
4. Enter the connection details of the database to which the variant tables are to be uploaded.
5. Choose *Finish*.

2.2.2 SAP Testing Perspective

The *SAP Testing* perspective opens automatically when you open a knowledge base. This perspective has its own set of associated views and editors. These include the Configuration Editor, Characteristics view, Non-Part Instances view, Properties view, and a number of debug/analysis views.

Component	Description
Configuration Editor	Shows the content of the configuration session in a tree structure. The highest-level node shows the database name, knowledge base name, profile, and version of the knowledge base that was opened to start this configuration session. Directly under the knowledge base node is the configuration node, which shows the name of the configuration (same as the knowledge base name). ℹ Note Due to knowledge base orchestration (KBO), multiple configurations driven by their own knowledge bases can be active at the same time. These are 'orchestrated' to work together as one configuration. Below the configuration node are the class and material instances, under which are the characteristics with their assigned values. A green square indicates that all the required values have been assigned to the instance, that is, the instance is complete. A yellow triangle indicates that the instance is incomplete, that is, a required characteristic has not been assigned a value. A red circle indicates a conflict or inconsistent configuration. For help with debugging a conflict, open the *Conflicts* view and click any item in the configuration flagged with a red circle. Information about the conflict will be presented in the conflicts view. You can manipulate the configuration content in this editor by right-clicking any item or node. All of the permitted actions are then available for selection. To add an instance to the configuration, use the *Non-Part Instance* view.
Characteristics View	The Characteristics view lists the visible characteristics of the instance selected in the Configuration Editor. Values can be assigned to any characteristic, unless the model has marked it as 'no input' or a value has already been set by a constraint. Required characteristics are marked with a yellow triangle, even after a value has been assigned. Click the inverted triangle on the view header to view the menu. You can use the menu to show/hide invisible characteristics and to display language-dependent names and technical names.

Component	Description
Non-Part Instances View	The Non-Part Instances view is used to manually add instances of either classes or materials to the configuration. When you double-click any item in the Non-Part Instances view, it is added as an instance of that type in the form of a 'free-standing' independent instance. It does not have a 'part of' relation to the root instance, unless a constraint has been added to the model to establish a relationship. It is called a 'non-part instance' because it is not part of anything else. ℹ Note You can add components to a material at a given BOM position by right-clicking the material in the Configuration Editor. If a BOM has been defined for the material, the *Add Component* option is available for selection. If selected, a 'part instance' is added as a component of that material. Non-part instances cannot be added in this way. In this case, they must be added using the Non-Part Instances view.
Properties View	The Properties view shows information about all the properties of the current item (that is, the selected item in the active view). This can be an instance in the Configuration Editor or Characteristics view, or a constraint in the Justifications view. Any item you select in a view has properties, which are displayed in this view.

Debug and Analysis Views

These views are presented as a set across the bottom of the SAP Testing Perspective.

Component	Description
Test Runner	Tracks all actions and records them in a script file that can be saved and re-executed. The run pushbutton runs an imported script in its entirety. To run only to a particular step, select the last line to be executed, right-click, and choose *Run to Line*. The export pushbutton exports the current test runner content to a file. The system prompts you to enter the directory and file name. The import pushbutton imports a test script. The system prompts you to enter the directory and file name. The file must have the `.performer` suffix. The reset pushbutton resets the configuration session to its initial state. To add expected results to the script, select an object in the configuration editor, right-click and select one of the *Expect...* options. When replaying a script containing expected content, the SME compares the current content of the configuration with the recorded expected content. All matches are marked green, any unexpected content is marked red, and an explanation of the delta is provided.
Conflicts View	When two contradictory facts are detected, the engine raises a conflict. The Conflicts view lists the conflicts associated with the instance currently selected in the Configuration Editor. It also provides guidance for resolving the conflict in the form of "conflict assumptions".
Profiling View	Profiling is a function of the configuration engine that tracks the execution of dependencies and records the number of times each dependency is executed and the total time consumed. Profiling must be activated to start recording statistics and displays results when it is deactivated. To start or stop profiling, click the down arrow pushbutton in the Profiling view toolbar and choose *Start/Stop Profiling*.
Justifications View	Each instance and characteristic value is justified by a set of facts asserted by the user or by dependencies and is tracked by the Truth Maintenance System. The Justifications view shows all the facts and the dependencies that asserted them. Together, these facts and dependencies justify the item that is currently selected.

Component	Description
Trace View	The Trace view provides a detailed account of all engine activities. For more information, see Tracing [page 94].

2.2.2.1 Testing Models Locally

Context

You use this procedure to test your model prior to exporting it to the target system for use in the sales ordering process. In the test user interface (UI) and the test configuration engine, you can configure the solution to ensure that the model is performing as expected. You can either configure the solution manually or use a recorded test script. You can also measure the performance of constraints and dependencies.

Procedure

1. Choose *File* > *Open Knowledge Base...*
2. Choose the *Connection* and enter the required details.
3. Choose *Next*.
4. Choose a knowledge base and a profile.
5. Choose *Next*.

Results

The system opens the test UI. As you configure the solution in the test UI, the system displays the information on the following tab pages:

- **Test Runner**
  The system records all of the actions you perform in the test UI and displays the resulting script on the *Recorded Script* tab page. The script can be saved to a file. You can open a saved script on the *Executable Script* tab page and then run it. You can run the whole script, or you can run the script up to a particular line.
  If you run a saved script and then continue to configure the solution, the system adds your actions to the script.
- **Conflicts**
  The system displays the model conflicts that occur while testing the solution model.
- **Profiling**

The system displays all of the constraints executed during the configuration session. For each constraint, the system lists the number of calls and the execution time in milliseconds. You can select a column header to sort the data by that column.

- **Justifications**
  The system displays the justifications for configuration rules.
- **Trace**
  The system displays the steps taken during the configuration session.
- **Non-Part Instances**
  The view displays all of the non-part instances types. You can create a non-part instance by double-clicking a type and then add it to your configuration for testing.

> ℹ Note
>
> After testing your model locally, you can test the knowledge base in the sales transactions within your back-end SAP S/4HANA system. To do so, you must export the knowledge base to a S/4HANA database in the same way as you export it to the local database. In this case, however, you choose the S/4HANA connection instead of the local database connection. For more information, see Exporting a Knowledge Base Runtime Version [page 82].

2.2.2.1.1 Test Runner

The “Test Runner” view allows you to run the test scripts partially or completely. Furthermore, it provides actions for storing and loading test scripts. When a test script is run, the configuration tree is changed based on the actions in the test script. For expectations, the configuration tree is inspected. If the expectation does not match the actual state of the configuration tree, the deviation is reported in the “Test Runner” view. The *Loaded Files* tab displays the loaded performer scripts.

The test (performer) script is used to record all user commands as test steps with the purpose of storing and executing them at a later time. In the Solution Modeling Environment, you can record “expectations”. This allows you to inspect a model after executing user commands and verify that the expectations still hold, for example, after a model has been changed. Therefore, the test scripts and test-runner UI now serve as a simple tool for automated model tests.

2.2.2.1.1.1 Saving Test Scripts (Performer)

You use this procedure to save an SAP Solution Sales Configuration test script in the Eclipse-based Solution Modeling Environment.

Procedure

1. Open the knowledge base.

2. Modify the model configuration.
3. Choose *Save Script* in the Test Runner view.
4. Enter the file name and save.

   The test script is saved in the specified location on the file system, in the `.performer` format.

Next Steps

After you have saved the test script, you can load, run, and reset the script.

2.2.2.1.1.2 Loading Test Scripts (Performer)

You use this procedure to load an SAP Solution Sales Configuration test script (performer files) in the Eclipse-based Solution Modeling Environment.

Context

You can use this script to test the configuration for the model and can load a single or multiple test scripts to set the configuration.

Procedure

1. Choose *Load Script* in the *Test Runner* view.
2. Select the test-script files and choose *Open*.

   > i Note
   >
   > If multiple performer scripts are loaded, they must be of the same KB RTV.

Results

The performer test scripts are displayed on the *Loaded Files* tab.

2.2.2.1.1.3 Running Test Scripts (Performer)

You use this procedure to run an SAP Solution Sales Configuration test script in the Eclipse-based Solution Modeling Environment.

Procedure

1. Choose *Run Script* in the *Test Runner* view.
2. Double-click on the test script on the *Loaded Files* tab for details.

   The *Executable Script* tab opens with the test script details.

Results

The performer test scripts run with the status *SUCCESS* or *FAILURE*.

2.2.2.1.1.4 Resetting Test Scripts (Performer)

You use this procedure to reset an SAP Solution Sales Configuration test script in the Eclipse-based Solution Modeling Environment.

Context

The target configuration is set when a test script for a model is executed. You can choose to reset either one or multiple configuration scripts, as they are loaded in the test runner.

Procedure

1. Open the *Test Runner* view.
2. Choose either *Reset Configuration* (to reset a single configuration script) or *Reset All Configurations* (to reset multiple configuration scripts).

Results

The performer test script is reset to *Not Run*.

2.2.2.1.1.5 Deleting Test Scripts (Performer)

You use this procedure to delete SAP Solution Sales Configuration test scripts in the Eclipse-based Solution Modeling Environment.

Procedure

1. Select the test script performer on the *Loaded Files* tab.
2. Choose *Delete File* in the *Test Runner* view.

Results

The performer test script is deleted from the *Loaded Files* tab.

2.2.2.1.2 Problem Filter on Executable Script Tab

You use this procedure to filter actions in the Eclipse-based solution modeling environment. You can filter the actions based on the problem explanation to easily find out the failure actions encountered during the test script run.

Procedure

1. Open the *Executable Script* tab.
2. Enter text in the *Problems Filter* based on the explanation of the problem.

> i Note
>
> Multiple performer scripts loaded must be of the same KB RTV.

2.2.2.2 Tracing and Logging

Use

The tracing function enables you to record selected engine activities.

Prerequisites

- You have activated the trace by choosing the *View Menu* pushbutton in the *Trace* view and then choosing *Configure*. In the trace configuration window, choose *Enable Tracing*.
- You have selected the types of activity to be traced.

Types of Traceable Activity

Type	Description	Comments
DDB (Dynamic database)	The DDB is the engine-internal storage for facts maintained by the inference engine. Trace output of this type refers to changes in the DDB.	Tracks all assertion and retraction of facts in the dynamic database
PMS (Pattern matching system)	The PMS is responsible for matching patterns in a dependency against facts in the DDB. Dependencies are evaluated against the provided matches. In other words, the PMS is responsible for identifying the dependencies to be executed and for providing them with the objects (such as instances and characteristics) that are required for the dependency evaluation. Trace output of this type refers to activities inside the PMS.	Tracks all patterns in the configuration that are detected by the pattern matching system
CSTR (Constraint)	Trace output of this type refers to constraint evaluations.	Records the execution of any constraint
RULE	Trace output of this type refers to rule evaluations.	Records the execution of any rule
SCND (Selection condition)	Trace output of this type refers to the evaluation of selection conditions.	Traces the execution of any selection condition
PCND (Precondition)	Trace output of this type refers to the evaluation of preconditions.	Traces the execution of any preconditions
PROC (Procedure)	Trace output of this type refers to the evaluation of procedures.	Traces the execution of any procedures
FUNC (User-defined functions)	Trace output of this type refers to the evaluation of user-defined functions.	Records the execution of user-defined functions
TABL (Variant tables)	Trace output of this type refers to the access of variant tables.	Tracks interaction with a variant table

The trace result can be displayed in the configuration dialog. Each row is structured as follows:

```
[EngineTrace]:[ <Instance Number>- <Knowledge Base Name>] <Sequential Number for
Each Trace Row> <Related Engine Activity Type> <Message Number of the Text
Displayed for the Engine Activity Performed> <Engine Activity Performed>
```

> i Note
>
> The message number of the text displayed for the engine activity performed corresponds to the numbers in ABAP message class 34.

> Example
>
> ```
> [EngineTrace]:[1-HSS_SOLUTION_KB] 1 DDB 204 New fact "sf($1, SCM_SALES_USERS,
> VAL, 150.0)" inserted by "User"
> ```

Many expressions relating to the engine activity performed are self-explanatory. However, expressions about facts need further explanation.

A fact represents an elementary assertion about the state of the configuration. For example, if the user sets a characteristic `COLOR` to the value `BLUE`, a specific fact is created in the engine. The following types of facts can occur in the trace:

Fact Printed in Trace (Examples)	Explanation
`sf($1, COLOR, VAL, BLUE)`	This is a simple fact (sf). For the instance `$1`, this fact sets the value of the characteristic `COLOR` to `BLUE`. `VAL` refers to a specific facet of a characteristic. The available facets are described below.
`to($1, product: SAP_SYS)`	This is a type-of (to) fact which is used for classification. The instance `$1` is of the product type `SAP_SYS`
`rd($1, COLOR, BLUE, RED)`	This fact is about a restrictable domain (rd). For the instance `$1`, this fact sets the domain of values for the characteristic `COLOR` to the values `BLUE` and `RED`.

Facets of Characteristics

The tracing output of a simple fact (sf) always describes a certain facet of a characteristic. A facet is a property of a characteristic. The following facets exist:

- `VAL` (value facet): Value of a characteristic
- `DOM` (domain facet): Domain of a characteristic
- `REQ` (required facet): Specifies whether a characteristic is required
- `INV_P` (invisible facet): Specifies the visibility of a characteristic
- `NOINP_P` (no input facet): Specifies whether a characteristic is read-only

2.2.2.3 Import/Export Configuration

You can test your configuration on the test UI of the Solution Modeling Environment. In this way, you can mimic the restore scenario that is available on the user interface of SAP Solution Sales Configuration.

This feature in the Solution Modeling Environment enables you to perform the following tasks:

- Export configuration from the SME to XML
- Import configuration XML that has been exported from the SME
- Import configuration XML that has been exported from the user interface of SAP Solution Sales Configuration
- After the import, edit the configuration and export the updated configuration to XML

Export Configuration

On the *SAP Testing* user interface, you can export knowledge base configuration as a `*.cfg` file.

1. Open the knowledge base and complete configuration on the *SAP Testing* UI.
2. Right-click the tree view and go to ▶ *Configuration* › *Export* ◀.
3. Enter a file name for your configuration.

Restore Configuration

In the Solution Modeling Environment, you can restore the configuration XML as follows:

1. Go to ▶ *File* › *Restore Configuration* ◀.

   > i Note
   >
   > You must create a connection to the local database if a connection doesn't already exist.

2. Enter the database password and click *Next* to open the wizard.
3. Click *Browse* to select a configuration file. You can export files that have the formats `*.cfg` or `*.xml`.
4. Select the KB reference date and click *Finish* to see the restored configuration on the *SAP Testing* UI.

> i Note
>
> After the restore, you can edit the configuration on the *SAP Testing* UI. However, you cannot save a performer script for this updated configuration. (The *Save Script* button has been disabled in *Test Runner*).

2.2.2.4 DDB, PMS, and TMS Dumps

The Solution Modeling Environment provides dumps to support the user in analyzing modeling problems.

The following dumps are provided by the SME to help users analyze modeling problems:

- DDB (dynamic database) Dump
  The DDB is the internal storage for all facts maintained by the inference engine. The trace output for this dump refers to changes in the DDB.
- PMS (pattern matching system) Dump
  The PMS is responsible for matching patterns in a dependency against facts in the DDB. These dependencies are then evaluated against the provided matches. In other words, the PMS is responsible for identifying the dependencies to be executed and for providing them with the objects (such as, instances and characteristics) required for the dependency evaluation.
- TMS (truth maintenance system) Dump
  The TMS determines the status of facts given the known set of justifications. Any status changes are communicated to the TMS client. Every time a new justification is communicated to the TMS, the status of all facts and triggers is updated (incrementally). The TMS dump also performs an atms-label calculation for all facts and triggers, on demand.

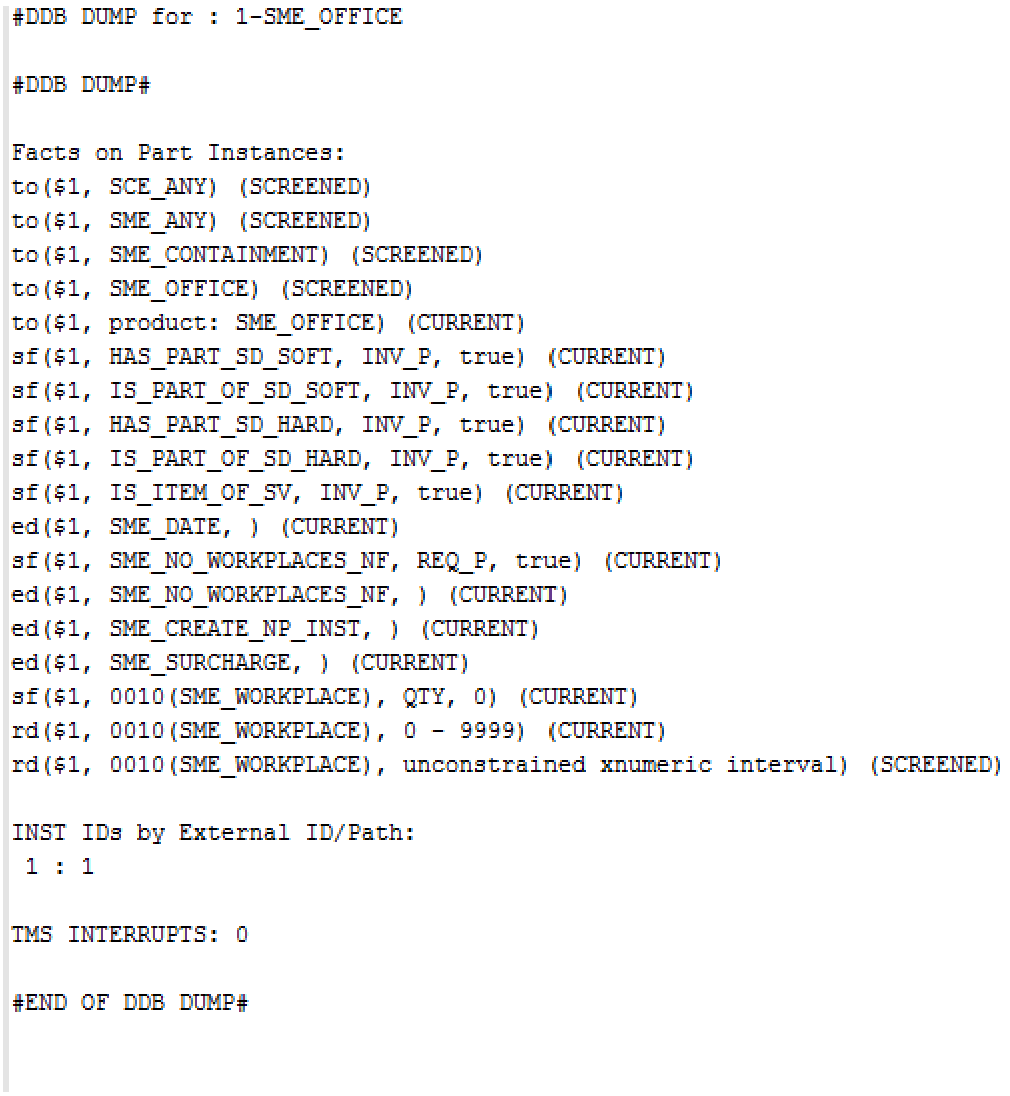
2.2.2.4.1 Exporting DDB Dumps

You use this procedure to export a DDB dump from the Eclipse-based Solution Modeling Environment.

Procedure

1. Right-click the knowledge base in the *Model* view and select ▶ *Configuration* ▶ *DDB Dump* ◀.
2. Choose the location in which you want to save the file and enter a file name.

The DDB dump is exported as shown below:



```
#DDB DUMP for : 1-SME_OFFICE

#DDB DUMP#

Facts on Part Instances:
to($1, SCE_ANY) (SCREENED)
to($1, SME_ANY) (SCREENED)
to($1, SME_CONTAINMENT) (SCREENED)
to($1, SME_OFFICE) (SCREENED)
to($1, product: SME_OFFICE) (CURRENT)
sf($1, HAS_PART_SD_SOFT, INV_P, true) (CURRENT)
sf($1, IS_PART_OF_SD_SOFT, INV_P, true) (CURRENT)
sf($1, HAS_PART_SD_HARD, INV_P, true) (CURRENT)
sf($1, IS_PART_OF_SD_HARD, INV_P, true) (CURRENT)
sf($1, IS_ITEM_OF_SV, INV_P, true) (CURRENT)
ed($1, SME_DATE, ) (CURRENT)
sf($1, SME_NO_WORKPLACES_NF, REQ_P, true) (CURRENT)
ed($1, SME_NO_WORKPLACES_NF, ) (CURRENT)
ed($1, SME_CREATE_NP_INST, ) (CURRENT)
ed($1, SME_SURCHARGE, ) (CURRENT)
sf($1, 0010(SME_WORKPLACE), QTY, 0) (CURRENT)
rd($1, 0010(SME_WORKPLACE), 0 - 9999) (CURRENT)
rd($1, 0010(SME_WORKPLACE), unconstrained xnumeric interval) (SCREENED)

INST IDs by External ID/Path:
 1 : 1

TMS INTERRUPTS: 0

#END OF DDB DUMP#
```

DDB Dump

2.2.2.4.2 Exporting PMS Dumps

You use this procedure to export a PMS dump from the Eclipse-based Solution Modeling Environment.

Procedure

1. Right-click the knowledge base in the Model view and choose *Configuration* > *PMS Dump*.
2. Choose a location in which to save the file and enter a file name.

2.2.2.4.3 Exporting TMS Dumps

You use this procedure to export a TMS dump from the Eclipse-based Solution Modeling Environment.

Procedure

1. Right-click the knowledge base in the Model view and choose ▶ *Configuration* › *TMS Dump* ◀.
2. Choose a location in which to save the file and enter a file name.

2.2.2.5 Importing a Model in Local Database

Use this process to import a knowledge base from a SAP S/4HANA system using the data loader.

Use

You use this procedure to import the interface characteristics from compatible-mode knowledge bases in the source SAP S/4HANA system to the solution modeling environment database. You perform these steps in the Eclipse-based solution modeling environment.

Prerequisites

You have setup the data loader connection in the Solution Model Environment.

Procedure

1. Choose ▶ *File* › *Import...* ◀.
   The system displays a dialog box prompting you to choose an import source.
2. Choose ▶ *SAP CPQ for Solution Sales Configuration* › *Data Loader* ◀.
3. Choose *Next*.
   The system displays a dialog box prompting you to choose a Data Loader configuration from a list of the configured connections.
4. Select the configuration for the system and choose *Next*.
5. Enter the required passwords and choose *Finish*.

> **i Note**
>
> Compatible mode knowledge bases imported from the SAP S/4HANA system can be tested using the test user interface; however they cannot be changed in the solution modeling environment.

For more information, see **Data Loader in the Solution Modeling Environment**.

More Information

Related Information

Data Loader in the Solution Modeling Environment [page 105]

2.2.2.6 Importing a Model into a File

Prerequisites

A java project needs to be created, under which the relevant `.ssc` file is created.

Context

This procedure is used to import a knowledge base present in a back-end database (S/4HANA, local database) into a file, in an existing java project. The imported file contains the following details:

- Material definition
- Class definition
- Characteristic definition
- Bill of materials definition

as per the selected KB.

Procedure

1. In the solution modeling environment, choose *File Import* > *SAP CPQ for Solution Sales Configuration* > *Import Model to File*.

   Choose *Next*.

2. Choose one of the existing connection types: *SAP S/4HANA*, or the local database.
3. Enter the *Client*, *User*, and *Password*.

   Choose *Next*.
4. Select the knowledge base runtime version for the product .

   You can sort the results by clicking on any one of the column headers.
5. Select the master data definitions you want to import.

   Choose *Include Dependent Bill of Materials* if you want to import the material definitions with the `boms` parameter. If you do not choose this option, the material definition will not contain a reference to a BOM.

   If you choose *Include Dependent Material Classifications*, the imported material definitions will include the `classes` parameter. If you do not choose this option, the material definition will not contain a reference to classes.

   > i Note
   >
   > These options affect only the material definition and not the import of bill of material or class definitions

   Choose *Next*.
6. Specify the folder and file names to be created, to store the definitions of the imported items.

   > i Note
   >
   > Do not select any advanced features.

   Choose *Finish*.

2.2.2.7 Importing a Model into a New Project

Context

This procedure is used to import a knowledge base present in a backend database (S/4HANA, local database) into a Java project. The project then created will have individual folders of bills-of-materials, classes, characteristic and materials. Each folder shall have the relevant material definition, class definition, characteristic definition and bill of material definition in an .ssc file as per the elements selected during creation

Procedure

1. In the solution modeling environment, choose ▶ *File* ▶ *Import SAP CPQ for Solution Sales Configuration* ▶ *Import Model to New Project* ◀.

ChooseNext.

2. Choose one of the available connections: *S/4HANA*, or the local database.
3. Enter the *Client*, *User*, and *Password*.

   Choose *Next*.
4. Select the knowledge base runtime version for the product.

   You can sort the results by clicking on any one of the column headers.
5. Select the master definitions you want to import.
6. Choose *Include Bill of Material* if you want to import the material definitions with the `boms` parameter. If you do not choose this option, the material definition will not contain a reference to a BOM.

   If you choose to *Include Dependant Material Classifications*, the imported material definitions will include the *classes* parameter. If you do not choose this option, the material definition will not contain a reference to classes.

   > i Note
   >
   > These options affect only the material definition and not the import of bill of materials definitions or class definitions.

   Choose *Next*.
7. Enter a project name under which the project folders will be created. The `.ssc` files with the selected parameter definitions will be saved here.

   Choose *Finish*.

2.2.2.8 Importing Master Data from SAP S/4HANA Systems

Use

At runtime in the SAP Solution Sales Configuration engine, the solution knowledge bases created in the solution modeling environment interact with the product knowledge bases created in SAP S/4HANA.

Most of the master data for the referenced products is already defined in the SAP S/4HANA system and can be reused in the solution modeling environment. These master data definitions are included in the knowledge base run time versions imported into the local SQL server database.

You use this procedure to import this master data to either a file or a new project.

Prerequisites

- You have configured a connection to your SAP S/4HANA system in the Eclipse-based solution modeling environment.

Procedure

Importing Master Data to a File

1. In the solution modeling environment, choose *File* › *Import* › *SAP CPQ for Solution Sales Configuration* › *Import Model to File*.
2. Choose *Next*.
3. Choose a S/4HANA connection.
4. Enter the client, user, and password.
5. Choose *Next*.
6. Select the knowledge base runtime version for the product (you can sort the results by clicking any column header).
7. Select the master data definitions you want to import.
   Choose *Include Dependent Bill of Materials* if you want to import the material definitions with the “boms” parameter. If you do not choose this option, the material definition will not contain a reference to a BOM.
   If you choose *Include Dependent Material Classifications*, the imported material definitions will include the “classes” parameter. If you do not choose this option, the material definition will not contain a reference to classes.

   > i Note
   >
   > These options affect only the material definition and not the import of bill of material definitions or class definitions.

8. Choose *Next*.
9. Specify the folder and file name to be created to store the definitions of the imported items.

   > i Note
   >
   > Do **not** select any advanced features.

10. Choose *Finish*.

Importing Master Data to a New Project

1. In the solution modeling environment, choose *File* › *Import* › *SAP CPQ for Solution Sales Configuration* › *Import Model to new Project*.
2. Choose *Next*.
3. Choose a S/4HANA connection.
4. Enter the client, user, and password.
5. Choose *Next*.
6. Select the knowledge base runtime version for the product (you can sort the results by clicking any column header).
7. Select the master data definitions you want to import.
   Choose *Include Dependent Bill of Materials* if you want to import the material definitions with the “boms” parameter. If you do not choose this option, the material definition will not contain a reference to a BOM.
   If you choose *Include Dependent Material Classifications*, the imported material definitions will include the “classes” parameter. If you do not choose this option, the material definition will not contain a reference to classes.

> **i Note**
>
> These options affect only the material definition and not the import of bill of material definitions or class definitions.

8. Choose *Next*.
9. Enter a project name.
10. Choose *Finish*.

2.2.3 Data Loader in the Solution Modeling Environment

Use

The data loader is a tool that allows you to download the configuration master data either from a S/4HANA system. The data loader registers with the SAP S/4HANA gateway, initiates download requests, and processes the call-backs from the SAP S/4HANA system. The download itself is a “push” mechanism for which two RFC connections are required:

1. Outbound connection from the data loader to the SAP S/4HANA system to read the SAP gateway parameters and initiate download requests.
2. Inbound connection from the SAP S/4HANA system in order to process the “pushed” data.

The data loader is used in the solution modeling environment to download configuration master data, for your solutions from the SAP S/4HANA system to the solution modeling environment database. It is installed as an Eclipse plug-in. Before you can download data, you must configure a connection to the source SAP S/4HANA system and then add the connection to the data loader configuration settings.

For more information about using the data loader in the solution modeling environment, see **Importing a Knowledge Base using Data Loader**.

Related Information

Importing a Model in Local Database [page 100]

2.3 SSC DevOps Wizard

To facilitate faster creation of automated setup for KB exports, a new SSC DevOps Project Wizard has been added in SME. You can create a new DevOps project from the file menu and select the knowledge bases and destinations for export. The wizard will generate a script for individual connections, and a master script for exporting to all the connections. The wizard uses the KB project and its dependent project information from the eclipse workspace to generate the scripts. These scripts can also be updated later to include more knowledge bases and connections.

To create a new DevOps project, perform the following steps:

1. Go to ▶ *File* › *New* › *SSC DevOps Project* ◀ and specify the *Project Name* and *Location*.
2. In the next window, select the knowledge bases to be exported, and in the subsequent window, select the connections to which the KBs need to be exported.
3. Upon finishing, one script for each individual connection, and a master script that can execute all the individual scripts will be generated.
4. This DevOps project can be executed using SSC DevOps run configurations. Results of the execution are displayed in the eclipse console.
5. To update the project, right click and select the *Update Devops Project* menu option. New SSC projects added in the workspace will be listed which can be then selected for script creation.

This DevOps project can be managed in Git and can be then used to setup headless KB export on a continuous integration server like Jenkins. For more infomartion, refer to the SSC Solution Modeling Environment Overview presentation on the http://help.sap.com. Search for ▶ *SAP Solution Sales Configuration* › *Relevant Version* › *Additional Information* › *SSC Solution Modeling Environment Overview* ◀.

2.4 Analyzing Pricing Traces

It is now possible to turn on pricing traces while testing the configuration. The pricing context parameters can be set in preferences, and the pricing context can be set during the opening of a knowledge base in SME.

When the pricing context is enabled, pricing parameters set in preferences will be passed onto the configuration session, and pricing will be enabled to turn on the pricing traces. It needs to be ensured that the material and pricing data is already downloaded in the database using which the tests are being performed.

You need to perform the following steps to use this feature:

1. Go to ▶ *File* › *Open Knowledge Base* ◀ and select the connection.
2. On the knowledge base selection page, select the kb.
3. Click the *Pricing Context Properties* button. You will be navigated to the pricing context properties preference page. Set the pricing context properties here and click on *Apply*.

   You can also go to this page by navigating to ▶ *Window* › *Preferences* › *SAP SSC* › *Pricing Context Properties* ◀
4. On the KB selection page, select the check box to *Set Pricing Context*. It will turn on pricing for this configuration.
5. Perform the configuration in the test perspective.
6. To view the pricing traces for the current configuration, right-click on the model tree and select *Show Pricing Traces*. This will open the pricing traces in the eclipse xml editor.
7. Similarly, to turn on pricing for Restore Configuration, we need to set the pricing context properties in the preferences, and select the *Set Pricing Context* checkbox on the restore configuration page

For more infomartion, refer to the SSC Solution Modeling Environment Overview presentation on the http://help.sap.com. Search for ▶ *SAP Solution Sales Configuration* › *Relevant Version* › *Additional Information* › *SSC Solution Modeling Environment Overview* ◀.

2.5 Data Exchange

Use

Master data must be exchanged between the solution modeling environment and the target sales system, for example, SAP S/4HANA system. The main data exchanges are as follows:

- Modelers import data such as characteristics and classes from the SAP S/4HANA system to the solution modeling environment.
- Modelers transfer the completed solution model (knowledge base runtime version) from the solution modeling environment to the SAP S/4HANA system.

Pricing information is not exported to the solution modeling environment.

> ℹ Note
>
> If you want to configure and export SME into a local database (for example, for use with SAP Solution Sales Configuration in Hybris), you must download and register the relevant database driver.

> ⚠ Caution
>
> The Data Loader does not import product data into the solution modeling environment. Therefore, you must ensure that the products defined in the solution models also exist in the target sales system(s) where the knowledge base is deployed.

Prerequisites

You have configured the connection between the solution modeling environment and the target sales system.

More Information

For more information about importing data, see Data Loader in the Solution Modeling Environment [page 105].

2.6 Maintaining Modeling Templates

Context

You use this procedure to maintain templates for model elements. When you open an empty text file in the solution modeling environment, and then use the auto-completion feature (CTRL + SPACEBAR), the system lists all of the available templates. Templates are stored as XML files.

Procedure

1. Choose *Window* > *Preferences*

   The system displays the preferences dialog.
2. Choose *SAP CPQ for solution sales configuration* > *Modeling Templates*

   The system displays the *Templates* dialog.

Results

The system lists all of the existing templates. A template is defined with the following information:

- Name
- Context (for example, Bill of Material, Class, Constraint, and so on)
- Description
- Pattern

If you select a template, the content of the template is displayed in the *Preview* area.

The *Templates* dialog provides the following functions:

Function	Description
New...	You use this function to create a new template.
Edit...	You use this function to edit the selected template.
Remove	You use this function to remove the selected template.
Import	You use this function to import a template from an XML file.
Export	You use this function to export the selected template to an XML file.

2.7 Developing SSC User Exits

To facilitate quick development of user exits, a new wizard has been added in SME. This wizard will create a skeleton for a pFunction project in which custom classes can be added. This pFunction can be then exported from SME to the local database or backend system for testing.

While executing a configuration in SME test perspective, pFunctions would be loaded from the database along with the KB. The existing pFunction jar files can also be exported using this project by placing the jar files in the target directory of the pFunction project.

You can perform the following steps to use this wizard:

1. Go to *File* > *New* > *SSC pFunction Project*.
2. Specify the *Project Name*, *Unit Name*, and *Unit Version* and create the project.
3. A project will be created in the eclipse workspace in which java classes can be added.
4. To export the pFunction, select the pFunction project and go to *File* > *Export* > *SAP SSC* > *Export SSC pFunction*.

In case a custom unit version has been used during project creation, the same needs to be specified in the engine settings preference page before start of the configuration in the testing perspective.

For more infomartion, refer to the SSC Solution Modeling Environment Overview presentation on the http://help.sap.com. Search for *SAP Solution Sales Configuration* > *Relevant Version* > *Additional Information* > *SSC Solution Modeling Environment Overview*.

2.8 Collaboration Between Modelers

Use

Complex solution models that combine different products are often maintained by more than one modeler. In addition to this, master data can be reused in different products. This means that different modelers may have to make changes to the same model elements. This function ensures that locally maintained models can be merged and synchronized by using a model repository that defines the rules and workflows for collaboration.

When changes are made to a solution model, the system does not directly store the model in the SAP S/4HANA system. It first merges the changes in a model repository that defines the rules and workflows for collaboration. A consistent model version is then transferred from the model repository to the relevant system.

> **i Note**
>
> The solution modeling environment does not contain its own repository technology; instead it is designed to be used with existing file-based Source Code Management Systems (SCMS).

Integration

The solution modeling environment is based on the Eclipse platform, and connector plug-ins are freely available for the most commonly used SCMS systems, such as Subversion, CVS, and Git. Most SCMS systems also come with the Eclipse Connector plug-in.

The SCMS system can be integrated with the solution modeling environment in the following ways:

- By downloading the required SCMS connector plug-in to the solution modeling environment using the integrated update manager.
- By downloading the solution modeling environment as a plug-in in an existing Eclipse-integrated development environment (IDE) with an existing SCMS connector.

2.9 Setting Context Properties

The configuration engine uses reference characteristics to obtain the context of the environment in which it is working. An example of this context could be a sales organization, a distribution channel, etc. In SAP S/4HANA orHybris, this context is passed on to the configuration engine through RFCs and APIs.

In SME, this context can be passed using a properties file. The format of the context properties file should be:

```
ref_cstic1=value1
ref_cstic2=value2
```

You can use this procedure to set context properties while opening a runtime knowledge base version or while restoring configuration.

Setting Context Properties When Opening a KB and Creating a New Configuration

1. Choose *File* › *Open Knowledge Base*.
2. Enter the connection details.
3. Choose *Next*.
4. Select the knowledge base and profile.
5. Select *Context Property*.
6. Browse the property file.
7. Choose *Finish*.

Setting Context Properties When Restoring an Existing Configuration

1. Choose *File* › *Restore Configuration*.

2. Enter the connection details.
3. Choose *Next*.
4. Browse the configuration file.
5. Select *Context Property*.
6. Browse the property file.
7. Choose *Finish*.

2.10 Automating Modeling Lifecycle

The solution modeling environment interface allows a user to use different features of the product.

Most of the features on the SME UI require human intervention for execution, for example, exporting a model project to a target system, restoring a configuration, execution of performer on a model project, etc.

The headless automated process not only allows automation of these features, but also allows the user to run regular automated jobs to ensure stability of models. The features mentioned below are currently supported in headless automation:

- Headless Export of Knowledge base
- Headless Performer Execution
- Headless XML Restore configuration
- Headless Upload of external Variant tables

2.10.1 Headless Export of Knowledge Bases

Use

You can use this process to export solution models to a target system, for example, a standalone database, SAP S/4HANA.

Prerequisites

- You have installed the solution modeling environment on the system from which you want to export the solution models
- You have set up an Eclipse workspace for the solution modeling environment
- You have defined the required back-end connections under *Window* › *Preferences* › *SAP Solution Sales Configuration* › *Connections*
- If solution models are to be exported to any of the supported database systems (see the Product Availability Matrix at http://support.sap.com/pam), such as Microsoft SQL Server, you have already registered the relevant database driver under *Window* › *Preferences* › *SAP Solution Sales Configuration* › *Drivers*

- You have configured all relevant settings in your solution modeling environment, such as source formatting and validation settings, under *Window* > *Preferences* > *SAP Solution Sales Configuration* > *Source Formatter / Validation*

Preparatory Steps

The knowledge base exporter requires the following information:

- The connection details of the target system
  The available connections are configured in the solution modeling environment under *Window* > *Preferences* > *SAP Solution Sales Configuration* > *Connections* and are stored under `<workspace_directory>\.metadata\.plugins\com.sap.custdev.projects.fbs.slc.conn.core\connections.xml`.
- The settings for the modeling language source formatter and model validation
  These settings are configured under *Window* > *Preferences* > *SAP Solution Sales Configuration* > *Source Formatter / Validation* and are stored under `<workspace_directory>\.metadata\.plugins\com.sap.custdev.projects.fbs.slc.sme\settings.xml`.

This information can be provided in multiple ways based on the mode of operation of the exporter:

- **Workspace mode**
  An Eclipse workspace is used to determine the settings and relevant modeling projects.
- **Directory mode**
  A minimal Eclipse workspace is used, that is, it should contain only registered database drivers. The settings and modeling projects are specified by parameters that are passed to the headless exporter executable.

2.10.1.1 Executing Headless Export

To start the headless knowledge base exporter, you must execute the `eclipse.exe` or `eclipsec.exe` file in your eclipse installation directory.

> → Recommendation
>
> Write a batch (`.bat`) or command (`.cmd`) file that calls the executable.

2.10.1.1.1 Eclipse-Specific Arguments

The behavior of the export can be controlled by the following arguments:

Eclipse-Specific Arguments

Argument Name	Argument Description
`--launcher.suppressErrors`	Suppress messages and error dialogs
`-application com.sap.custdev.projects.fbs.slc.sme.export.headless.application`	The application to run This must point to the headless export application.
`-noSplash`	Do not show the splash screen
`-console`	Show a console window
`-consoleLog`	Log messages to the console
`-vmargs`	Java VM arguments. This must be the last argument because all arguments after this one will be parsed as Java VM arguments. For example, `-vmargs -Xms128M -Xmx512M` , `-XX:PermSize=128M -XX:MaxPermSize=256M` , etc.

2.10.1.1.2 Export Application-Specific Argument

The behavior of the export can be controlled by the following arguments:

Export-Application-Specific Arguments

Argument Name	Example	Argument Description
`-data`	`-data "C:\path\to\workspace"`	The path to the Eclipse workspace. The workspace has to contain at least a JDBC driver registration (in directory mode). In workspace mode, the workspace also has to contain valid settings.
`-kbData`	`-kbData "C:\path\to\projects"`	(Relevant only in directory mode) The path to the directory with sub-directories containing all required `.ssc` files.
`-kbProject`	`-kbProject model.project.name`	**Workspace mode:** The name of the project that contains the KB to export. **Directory mode:** The directory names of all sub-directories in `-kbData` to be scanned. Use `-kbProject "subDir1;subDir2"`.

Argument Name	Example	Argument Description
`-kbName`	`-kbName KB_NAME`	The name of the knowledge base to be exported
`-kbValidate`	Not applicable	If specified, the knowledge base is validated before the export. i Note Always specify this parameter to prevent inconsistent knowledge bases from being exported.
`-conFile`	`-conFile "path\to\connections.xml"`	**Workspace mode:** Optional absolute path to connections to be used instead of the connections defined in the workspace. **Directory mode:** Required path to connections to be used. It can be absolute or relative to `-kbData`.
`-conName`	`-conName SYSTEM_1`	The name of the export connection to be used. If it is not specified, the export uses the default connection.
`-conPassword`	`-conPassword secret`	The password to use for the chosen connection.
`-settingsFile`	`-settingsFile "path\to\settings.xml"`	**Workspace mode:** Optional absolute path to the settings to be used instead of the settings defined in the workspace. **Directory mode:** Required path to settings to be used. It can be absolute or relative to `-kbData`.
`-kbConsoleLog`	Not applicable	Log messages and errors to the console.
`-kbEclipseLog`	Not applicable	Log messages and errors to the Eclipse runtime logging framework.
`-kbExcludeVariantTableContents`	Not applicable	Variant tables are exported without content.
`-kbExcludeLocalizations`	Not applicable	Localized short and long texts of model elements are not exported.

2.10.1.1.3 Example: Headless Knowledge Base Export

If the headless export code is put into a command file called `deployKb.cmd`, it can be used as follows:

- **Workspace mode**
  In case of the workspace mode, a typical call to the exporter appears as follows:

  ```
  eclipse\eclipsec -vm --launcher.suppressErrors -application
  com.sap.custdev.projects.fbs.slc.sme.export.headless.application -noSplash
  – console
  -consoleLog -kbValidate -kbProject -kbName -kbConsoleLog %* -vmargs
  -Xms128M -Xmx1024M - XX:PermSize=128M -XX:MaxPermSize=256M
  echo EXIT CODE: %ERRORLEVEL%
  ```

  > → Remember
  >
  > If the default eclipse workspace is not used, then `-data` argument should be used to specify the workspace path in headless export mode.

- **Directory mode**

  ```
  eclipsec --launcher.suppressErrors -application
  com.sap.custdev.projects.fbs.slc.sme.export.headless.application -noSplash
  -console
  -consoleLog -kbValidate -kbConsoleLog %* -vmargs -Xms128M -Xmx1024M -
  XX:PermSize=128M -XX:MaxPermSize=256M echo EXIT CODE: %ERRORLEVEL% -data
  "C:\path\to\workspace" -kbData
  "C:\path\to\projects_root"
  -kbProject your.project.with.kb -kbName KB_NAME -conFile
  "C:\path\to\connections.xml" -settingsFile "C:\path\to\settings.xml"
  ```

The application returns an exit code that can be accessed through variable `ERRORLEVEL`.

For more information about this, a sample batch script is available in SAP Note 2174488 (Headless Knowledge Base Export - Sample Script).

2.10.1.1.4 Exit Codes for Knowledge Base Export

Exit Codes for Knowledge Base Export

Exit Code	Description
0	All OK. KB was exported.
200	Invalid or insufficient arguments or parameters.
201	Given project (`-kbProject`) not found.

Exit Code	Description
202	Given KB (`-kbName`) not found.
203	Given KB is invalid (validation issues).
204	Connection not found or invalid.
205	Cannot execute export (internal problems; see log).

2.10.2 Headless Performer Execution

Headless performer execution allows a user to run performer scripts on the exported models in the target system, using an automated process.

You can use either the workspace scenario or the directory scenario to provide data for headless performer execution.

Workspace Scenario

This is an Eclipse workspace with a project and defined connections and a performer script directory.

The headless performer script execution requires the following data for the workspace scenario:

- Project workspace with projects
- Valid defined connections
- Performer scripts directory (performers specific to the workspace)

The workspace should contain the projects. You can set up an export connection by choosing *Window* > *Preferences* > *SAP Solution Configuration* > *Connections*.

> i Note
>
> When you close the workspace, the connections are stored under:
>
> `<workspace_dir.metadata>\.plugins\com.sap.custdev.projects.fbs.slc.conn.core\connections.xml`

> ⚠ Caution
>
> Add a connection password to the XML at your own risk. The password is stored as unsecure plain text. To do so, add the `password` attribute to the `clientSettings` element.

Directory Scenario

This is a simple directory containing sub-directories with the `.performer` script files and the required pfunctions (if any) in the respective directories and paths given by parameters to a valid `connection.xml`.

The headless performer script execution requires the following data for the directory scenario:

- One or more directories as a container for the `.performer` script files
- An XML file for the performer connection (database) to use (`connections.xml`)
- Path to the connection driver `.jar` file (for example, `D:\jars\sqljdbc4.jar`).

The application needs a path to a (parent) directory (`-psData “dir/parent”`) and it will scan all subdirectories for the available performer scripts. The scenario also requires a path to the connections via an XML file (`-conFile “path/connections.xml”`).

In both the workspace and directory scenarios, the performer run requires the `pFunction.jar` to be present in the same directory as the respective script.

2.10.2.1 Executing Headless Performer

To execute the “headless performer script”, call the **`eclipse.exe`** or **`eclipsec.exe`** (without additional console) in the Eclipse installation directory with the required arguments.

> → Recommendation
>
> We recommend using a batch (`.bat`) or command (`.cmd`) file.

2.10.2.1.1 Eclipse-Specific Arguments

Argument	Description
--launcher.suppressErrors	Suppress errors argument must be set before `vmargs`; normally one of the first arguments
-application com.sap.custdev.projects.fbs.slc.config.performer.headless.performerapplication	The application to start. Value (application ID) is separated by a space from the argument.
-noSplash	Do not show the splash.
-console	Show a console.
-consoleLog	Log to the console.

Argument	Description

-vmargs	Java VM arguments. Must be the last Eclipse argument because all of the following arguments will be parsed as Java VM arguments, for example:
	`-vmargs -Xms128M -Xmx512M`
	`-XX:PermSize=128M -XX:MaxPermSize=256M`

> **i Posting Instructions**
>
> For a full list of the supported arguments, see the Eclipse documentation .

2.10.2.1.2 Performer-Application-Specific Arguments

> **i Note**
>
> Use quotation marks ("") for values that contain spaces such as `-psdata “D: \Performer Data”`.

Parameter	Workspace Scenario	Directory Scenario
-projectWorkspace	Workspace project name. -projectWorkspace ssc_office_example	Not applicable
-psData	The path to the directory with the subdirectories containing all required `.performer` files. -psData “C:\parentDirectory”	The path to the directory with the subdirectories containing all required `.performer` files. -psData “C:\parentDirectory”
-conFile	Not applicable	Required path to connections to use. -conFile “C:\path\connections.xml”
-conDriver	Not applicable	The path to the connection driver jar file. -conDriver “D:\jars\sqljdbc4.jar”
-psConsoleLog	Print logs on console. -consoleLog	Print logs on console. -consoleLog
-conName	Not applicable	Connection name. -conName “localCon”

Parameter	Workspace Scenario	Directory Scenario
-conPassword	Connection password (optional if specified in `connections.xml` file). -conPassword “mypassword”	Connection password (optional here if specified in `connections.xml` file). -conPassword “mypassword”
-pContextPropertyFile	Path for the context properties to be set. This is an optional field and can be used as required by user. For example, C:\Users\Admin\ContextProperty\sme_office_prop.properties.	Path for the context properties to be set. This is an optional field and can be used as required by user. For example, C:\Users\Admin\ContextProperty\sme_office_prop.properties.
-pFunction	Specify the required pfunctions(if any). For example, -pFunction "C:/users/dir1/pFunction1.jar;C:/users/dir2/pFunction2.jar".	Specify the required pfunctions(if any). For example, -pFunction "C:/users/dir1/pFunction1.jar;C:/users/dir2/pFunction2.jar".

2.10.2.1.3 Example: Headless Performer Execution

→ Recommendation

We recommend using a batch file (`performer.bat`) for this process.

The application will return an exit code (see echo `%ERRORLEVEL%` in example cmd).

Workspace Mode

A batch or command script could appear as follows:

Sample Code

```
@echo off
eclipse\eclipsec -vm “%JRE_HOME%\bin” --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.config.performer.headless.performerapplicatio
n -noSplash -console -psConsoleLog -projectWorkspace “ssc_office_example”
-psData “C:\Users\Public\PROJECTWORK\Enhancements\TestRunner in Headless
Mode\PerformerData” -conName “localCon” -conPassword “mypassword”
```

Directory Mode

A batch or command script could appear as follows:

Sample Code

```
@echo off
eclipse\eclipsec -vm "%JRE_HOME%\bin" --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.config.performer.headless.performerapplicatio
n -noSplash -console -psConsoleLog -conDriver
"C:\Users\Public\SOFTWARES\sqljdbc4.jar" –conFile
"C:\Users\Public\PROJECTWORK\Enhancements\TestRunner in Headless
Mode\PerformerData\config\connections.xml" –psData
"C:\Users\Public\PROJECTWORK\Enhancements\TestRunner in Headless
Mode\PerformerData"REM -Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005@echo EXIT CODE:
%ERRORLEVEL%
```

2.10.2.1.4 Exit Codes for Performer Execution

Exit Codes for Performer Execution

Exit Code	Description
0	OK. Performer scripts have been executed.
300	Invalid or insufficient arguments or parameters.
301	Given performer scripts (`-psData`) not found.
302	Given workspace project not found.
303	Knowledge base not found.
304	Connection not found or invalid.
305	Cannot execute performer scripts (internal problems; see log).

2.10.3 Headless XML Configuration Restore

The headless restore configuration feature is part of the solution modeling environment that allows configuration restore using the .cfg/xml files through the headless automated process for the exported models on the target system.

There are two possible scenarios to provide the data for headless restore configuration:

- **Workspace scenario**
  An Eclipse workspace with project and its defined connections. The headless restore configuration needs the following data for workspace scenario:
  - Project workspace with projects
  - Valid defined connections
  - Configuration files directory (containing .cfg/.xml files)

  The workspace should contain all the relevant projects. You can setup an export connection from here:
  Window › *Preferences* › *SAP Solution Configuration* › *Connections*.

  > i Note
  >
  > When you close the workspace, the connections are stored under:
  >
  > `<workspace_dir.metadata>\.plugins\com.sap.custdev.projects.fbs.slc.conn.core\connections.xml`

  > ⚠ Caution
  >
  > Add a connection password to the XML at your own risk. The password is stored as unsecure plain text. To do so, add the `password` attribute to the `clientSettings` element.

- **Directory scenario**
  A simple directory with sub-directories with the .cfg/.xml files and paths given by parameters to a valid `connection.xml`. The headless restore configuration script execution needs the following data for directory scenario:
  - Configuration files directory (containing .cfg/.xml files)
  - An XML file (`connections.xml`) for the connection (DB)
  - A path to the connection driver jar file, for example, `D:\jars\sqljdbs4.jar`

  The application needs a path to a parent directory (`-rsData “dir/parent”`) and it will scan all sub-directories for the available configuration files (.cfg/.xml). The scenario requires also a path to the connections via XML file (`-conFile “path/connections.xml”`).

2.10.3.1 Executing Headless XML Configuration Restore

To execute the headless restore configuration script, call the `eclipse.exe` or `eclipsec.exe` (without additional console) in the Eclipse installation directory with the required arguments.

> → Recommendation
>
> Write a batch (`.bat`) or command (`.cmd`) file that calls the executable.

2.10.3.1.1 Eclipse-Specific Arguments

The behavior of the export can be controlled by the following arguments:

Eclipse-Specific Arguments

Argument Name	Argument Description
`--launcher.suppressErrors`	Suppress errors and arguments must be set before `vmargs` as it is one of the first arguments.
`-application com.sap.custdev.projects.fbs.slc.config.restore.headless.restoreapplication`	For the application to start. The Value (application ID) is separated from the argument by a single space.
`-noSplash`	Do not show the splash.
`-console`	Show a console.
`-consoleLog`	Log on to the console.
`-vmargs`	Java VM arguments. Should be the last Eclipse argument as all the arguments after this will be parsed as Java VM arguments. For example, `-vmargs -Xms128M -Xmx512M`, `-XX:PermSize=128M -XX:MaxPermSize=256M`, etc.

> **i Posting Instructions**
>
> For a complete list of the supported arguments, see the Eclipse documentation .

2.10.3.1.2 Restore Configuration Application-Specific Arguments

The behavior of the export can be controlled by the following arguments:

Restore Configuration Application-Specific Arguments

Parameters	Workspace Scenario	Directory Scenario
`-projectWorkspace`	Workspace project name `-projectWorkspace ssc_office_example`	N/A

Parameters	Workspace Scenario	Directory Scenario
`-rsData`	The path to the directory with the subdirectories containing all required configuration files `-rsData “C:\parentDirectory”`	The path to the directory with the subdirectories containing all the required configuration files `-rsData “C:\parentDirectory”`
`-conFile`	N/A	Required path to the relevant connections. `-conFile “C:\path\connections.xml”`
`-conDriver`	N/A	The path to the connection driver jar file. `-conDriver “D:\jars\sqljdbc4.jar”`
`-rsConsoleLog`	Print logs on console. `- rsConsoleLog`	Print logs on console. `- rsConsoleLog`
`-conName`	N/A	Connection name. `-conName “localCon”`
`-conPassword`	Connection password (optional unless specified in connections.xml file). `-conPassword “mypassword”`	Connection password (optional unless specified in connections.xml file). `-conPassword “mypassword”`

Note

“” are used for values containing spaces, for example, `-rsdata “D: \Configuration Data”`.

2.10.3.1.3 Example: Headless XML Configuration Restore

It is recommended to use a batch file, `restoreConfig.bat` for the restore process and the application returns an exit code once the restore is complete.

For more information about this, see `echo %ERRORLEVEL%` in example cmd.

Workspace mode

```
eclipse\eclipsec -vm "%JRE_HOME%\bin" --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.config.restore.headless.restoreapplication
-noSplash -console -rsConsoleLog  -projectWorkspace "ssc_office_example" -rsData
"C:\Users\Demo\PROJECTWORK\Enhancements\RestoreConfigData" -conName "localCon"
-conPassword "mypassword"
```

Directory mode

```
@echo off
eclipse\eclipsec -vm "%JRE_HOME%\bin" --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.config.restore.headless.restoreapplication
-noSplash -console -rsConsoleLog
-conDriver "C:\Users\Demo\SOFTWARES\sqljdbc4.jar" –conFile
"C:\Users\Demo\PROJECTWORK\Enhancements\RestoreConfigData\config\connection.xml"
–rsData "C:\Users\Demo\PROJECTWORK\Enhancements\ RestoreConfigData"
REM -Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005 @echo EXIT CODE:
%ERRORLEVEL%
```

2.10.3.1.4 Exit Codes for Configuration Restore

Exit Code	Description
0	All OK. Configuration files restored.
300	Invalid or insufficient arguments or parameters.
301	Given configuration files (`-rsData`) not found.
302	Given workspace project not found.
303	Knowledge base not found.
304	Connection not found or invalid.
305	Cannot restore configuration (internal problems; see log).
306	Cannot restore configuration (internal problems; see log).
307	Database driver not found (internal problems; see log).

2.10.4 Headless Upload External Variant Table

The headless upload external variant table feature is the part of solution modeling environment that allows upload of external variant table to a target system through headless automated process.

There are two possible scenarios to provide the data for headless upload of external variant table:

- **Workspace scenario**
  An Eclipse workspace with a project and its defined connections. The headless upload external variant table needs the following data for workspace scenario:
  - Project workspace with projects
  - Valid defined connections

- Data directory for external variant table downloaded from `S/4HANA` system.

The workspace should contain all the relevant projects. You can setup an export connection from *Window* › *Preferences* › *SAP Solution Configuration* › *Connections*.

> **i Note**
>
> When you close the workspace, the connections are stored under:
>
> `<workspace_dir.metadata>\.plugins\com.sap.custdev.projects.fbs.slc.conn.core\connections.xml`

> **⚠ Caution**
>
> Add a connection password to the XML at your own risk. The password is stored as unsecure plain text. To do so, add the `password` attribute to the `clientSettings` element.

- **Directory scenario**
  A simple directory containing the external variant table data files and their paths, given by parameters to a valid `connection.xml`. The headless external variant table script execution needs the following data for directory scenario:
  - Data directory for external variant table downloaded from S/4HANA system
  - An XML (`connections.xml`) file for the connection (DB)
  - A path to the connection driver jar file, for example, `D:\jars\sqljdbc4.jar`

The application needs a path to a parent directory (`-vtData “dir/parent”`), the data downloaded from S/4HANA external variant table download utility. The scenario requires also a path to the connections via XML file (`-conFile “path/connections.xml”`).

2.10.4.1 Executing Headless Upload External Variant Table

To execute the headless upload external variant table script call `eclipse.exe` or `eclipsec.exe` (without additional console) in the Eclipse installation directory with the required arguments.

> **→ Recommendation**
>
> Write a batch (`.bat`) or command (`.cmd`) file that calls the executable.

2.10.4.1.1 Eclipse-Specific Arguments

The behavior of the export can be controlled by the following arguments:

Eclipse-Specific Arguments

Argument Name	Argument Description
`-launcher.suppressErrors`	Suppress errors argument must be set before vmargs; normally one of the first arguments.
`-application com.sap.custdev.projects.fbs.slc.sme.extvartable.upload.headless.externalvarianttableapplication`	The application to start. Value (application ID) is separated by a space from the argument.
`-noSplash`	Do not show the splash.
`-console`	Show a console.
`consoleLog`	Log on to the console.
`-vmargs`	Java VM arguments. This must be the last Eclipse argument as all the arguments after this will be parsed as Java VM arguments. For example, `-vmargs-Xms128M-Xmx512M`, `-XX:PermSize=128`, etc.

i Posting Instructions

For a full list of the supported arguments, see the Eclipse documentation .

2.10.4.1.2 Upload External Variant Table Application-Specific Arguments

The behavior of the export can be controlled by the following arguments:

Upload External Variant Table Application-Specific Arguments

Parameter	Workspace Scenario	Directory Scenario
-projectWorkspace	Workspace project name. -projectWorkspace ssc_office_example	Not applicable
-vtData	The path to the directory with the subdirectories containing all required configuration files. -vtData “C:\extVarDirectory”	The path to the directory with the subdirectories containing all required external variant table files. -vtData “C:\extVarDirectory”
-conFile	Not applicable	Required path to the relevant connections. -conFile “C:\path\connections.xml”

Parameter	Workspace Scenario	Directory Scenario
-conDriver	Not applicable	The path to the connection driver jar file. -conDriver “D:\jars\sqljdbc4.jar”
-vtConsoleLog	Print logs on console. -vtConsoleLog	Print logs on console. -vtConsoleLog
-conName	Not applicable	Connection name. -conName “localCon”
-conPassword	Connection password (optional if specified in `connections.xml` file). -conPassword “mypassword”	Connection password (optional here if specified in `connections.xml` file). -conPassword “mypassword”

> **i Note**
>
> “” are used for values containing spaces, for example, `-vtdata “D: \External Variant Data”`.

2.10.4.1.3 Example: Headless Upload External Variant Table

It is recommended to use a batch file, `restore Config.bat` for the restore process and the application returns an exit code once the restore is complete.

For more information about this, see `echo %ERRORLEVEL%` in example cmd.

Workspace mode

```
eclipse\eclipsec -vm "%JRE_HOME%\bin" --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.sme.extvartable.upload.headless.externalvariantt
ableapplication -noSplash -console -vtConsoleLog  -projectWorkspace
"ssc_office_example" -vtData
"C:\Users\Demo\PROJECTWORK\Enhancements\ExternalVariantTableData" -conName
"localCon" -conPassword "mypassword"
```

Directory mode

```
@echo off
eclipse\eclipsec -vm "%JRE_HOME%\bin" --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.sme.extvartable.upload.headless.externalvariantt
ableapplication -noSplash -console -vtConsoleLog -conDriver
"C:\Users\Demo\SOFTWARES\sqljdbc4.jar" –conFile
"C:\Users\Demo\PROJECTWORK\Enhancements\ExtVarTabData\config\connection.xml" –
vtData "C:\Users\Demo\PROJECTWORK\Enhancements\ ExtVarTabData" REM -Xdebug
-Xnoagent -Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005 @echo EXIT CODE:
%ERRORLEVEL%
```

2.10.4.1.4 Exit Codes for External Variant Tables

Exit Codes for External Variant Tables

Exit Code	Description
0	All OK. External table variant files restored.
300	Invalid or insufficient arguments or parameters.
301	Given data files (`-vtData`) not found.
302	Given workspace project not found.
303	Knowledge base not found.
304	Connection not found or invalid.
305	Cannot execute external table script (internal problems; see log).
306	External variant tables uploaded with errors.
307	Database driver not found (internal problems; see log)

2.10.5 Headless Upload of Knowledge Bases

The process of headless upload of knowledge bases allows upload of `.xml/.txt` files pertaining to a certain knowledge base into a target system through headless automated process.

There are two possible scenarios to provide the data for headless upload of knowledge bases:

- **Workspace scenario**
  An Eclipse workspace with a project and its defined connections. The headless upload external variant table needs the following data for workspace scenario:
  - Project workspace with projects
  - Valid defined connections
  - `.xml/.txt` files data directory for external variant table downloaded from `S/4HANA` system.

  The workspace should contain all the relevant projects. You can setup an export connection from *Window* > *Preferences* > *SAP Solution Configuration* > *Connections*.

> i Note
>
> When you close the workspace, the connections are stored under:
>
> `<workspace_dir.metadata>\.plugins\com.sap.custdev.projects.fbs.slc.conn.core\connections.xml`

> ⚠ Caution
>
> Add a connection password to the XML at your own risk. The password is stored as unsecure plain text. To do so, add the `password` attribute to the `clientSettings` element.

- **Directory scenario**
  A simple directory containing `.xml/.txt` files and their paths, given by parameters to a valid `connection.xml`. The headless upload of knowledge base script execution needs the following data for directory scenario:
  - `.xml/.txt` files directory downloaded from S/4HANA system
  - An XML (`connections.xml`) file for the connection (DB)
  - A path to the connection driver jar file, for example, `D:\jars\sqljdbc4.jar`

The application needs a path to a parent directory (`-kbData` absolute path to the `.xml/.txt` files), the data downloaded from S/4HANA by download utility. The scenario also requires a path to the connections via XML file (`-conFile “path/connections.xml”`).

2.10.5.1 Executing Headless Upload

To execute the script for headless upload of knowledge bases, you must call the `eclipse.exe` or `eclipsec.exe` (without additional console) in your eclipse installation directory, with the required arguments.

> → Recommendation
>
> Write a batch (`.bat`) or command (`.cmd`) file that calls the executable.

2.10.5.1.1 Eclipse-Specific Arguments

The behavior of the export can be controlled by the following arguments:

Eclipse-Specific Arguments

Argument Name	Argument Description
`--launcher.suppressErrors`	Suppress errors Arguments must be set before `vmargs` (normally one of the first arguments)
`-application com.sap.custdev.projects.fbs.slc.sme.extvartable.upload.headless.externalvarianttableapplication`	The application to run Value (application ID) is separated from the argument by a single space.
`-noSplash`	Do not show the splash screen
`-console`	Show a console window
`-consoleLog`	Log messages to the console
`-vmargs`	Java VM arguments. This must be the last Eclipse argument because all arguments after this one will be parsed as Java VM arguments. For example, `-vmargs -Xms128M -Xmx512M`, `-XX:PermSize=128M -XX:MaxPermSize=256M`, etc.

i Posting Instructions

For a full list of the supported arguments, see the Eclipse documentation .

2.10.5.1.2 Headless Upload-Specific Arguments

i Note

Use quotation marks ("") for values that contain spaces such as `-kbData “D: \KnowledgeBaseData”`.

Parameter	Workspace Scenario	Directory Scenario
-projectWorkspace	Workspace project name. -projectWorkspace ssc_office_example	Not applicable
-conFile	Not applicable	Path for connections to be used -conFile "C:\path\connections.xml"
-conDriver	Not applicable	Path to the connection driver jar file -conDriver "D:\jars\sqljdbc4.jar"

Parameter	Workspace Scenario	Directory Scenario
-uploadConsoleLog	Print logs on console -uploadConsoleLog	Print logs on console.
-conName	Not applicable	Connection name. -conName “localCon”
-conPassword	Connection password (optional if specified in `connections.xml` file). -conPassword “mypassword”	Connection password (optional here if specified in `connections.xml` file). -conPassword “mypassword”
-kbData	Path to the directory containing `.xml/.txt` files	The argument mentions the path to the directory containing `.xml/.txt` files of the knowledge base

2.10.5.1.3 Example: Headless Knowledge Base Upload

> → Recommendation
>
> We recommend using a batch file (`restoreConfig.bat`) for this process.

The application will return an exit code (see echo `%ERRORLEVEL%` in example cmd).

Workspace Mode

A batch or command script could appear as follows:

Sample Code

```
eclipse\eclipsec -vm "%JRE_HOME%\bin" --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.sme.uploadKb.headless.uploadapplication
-noSplash– console -consoleLog -
uploadConsoleLog  -projectWorkspace "ssc_office_example" -kbData
"C:\Users\Administrator\xmlModels\TestModel " -conName "localCon"
-conPassword "mypassword"

echo

EXIT CODE: %ERRORLEVEL%
```

Directory Mode

Sample Code

```
@echo off

eclipse\eclipsec -vm "%JRE_HOME%\bin" --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.sme.uploadKb.headless.uploadapplication
-noSplash -console -
uploadConsoleLog -conDriver "C:\Users\Demo\SOFTWARES\sqljdbc4.jar" –conFile

"C:\Users\Demo\PROJECTWORK\Enhancements\configconnection.xml" –kbData
"C:\Users\Administrator\xmlModels\TestModel "

REM -Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005

@echo EXIT CODE: %ERRORLEVEL%
```

2.10.5.1.4 Exit Codes

Exit Codes for Uploading Knowledge Bases

Exit Code	Description
0	OK. KB was exported.
301	Invalid or insufficient arguments/parameters.
302	No project found.
303	Connection not found or invalid.
304	Upload failure
305	Database driver not found.

2.10.6 Headless Dataloader Execution

Use

Some scenarios required a data download from SAP backend system into the local DB on frequent basis and thus manual trigger of data loader is sub optimal. In such situations, data download can be automated by running SME data loader in the headless mode.

Installation

The Headless Dataloader is part of SME and will be installed with SME dataloader feature installation. Hence there is no additional step required.

Setup of Headless Dataloader Execution

Before setup of the SME in headless mode, the dataloader in SME must be setup and initial download triggered from the UI which can help verify if dataloader is working as desired. Files generated as part of UI setup can then be used to trigger dataloader in headless mode.

Setup of headless mode requires that the absolute path for the settings file is known and provided in the scripts. The following are paths for the files:

- *Datalaoder.xml*. This is present in D:\DBdrivers\dataloader.xml directory. This file is generated in your eclipse workspace when you add dataloader connection settings in eclipse preferences. The same file can be used.
- The jdbc driver jar file for the destination database, for example, *D:\Database Drivers\mssql\sqljdbc4-4.0.jar*

Executing the Headless Eclipse Application

To execute SME in the headless mode, *eclipse.exe* or *eclipsec.exe* (without additional console) must be invoked in the Eclipse installation directory with the required arguments. This command can be written in a batch script *(.bat/.cmd)*.

Eclipse Specific Arguments

-launcher.suppressErrors	Suppress errors, arguments must set before vmargs, normally one of the first arguments
-application com.sap.custdev.projects.fbs.slc.dataloader.headless.dataloaderapplication	The application to start. Value (application ID) is separated by a space from the argument.
-noSplash	Does not show the splash screen
-console	Displays a console window
-consoleLog	Log messages to the console
-vmargs	Java VM arguments. Must be the last Eclipse argument because all following arguments will be parsed as Java VM arguments, eg: *-vmargs -Xms256m -Xmx1024m* *XX:PermSize=128M -XX:MaxPermSize=256M*

A full list of supported arguments can be found here:http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fmisc%2Fruntime-options.html

Application Specific Arguments

> Note
>
> use *""* for values containing spaces like *conDriver D:\Database Drivers\mssql\sqljdbc4-4.0.jar*.

Parameters	Sample Value
-conFile	Dataloader settings file to use. For example conFile *D:\DBdrivers\dataloader.xml* You can find dataloader settings xml file created from SME inside eclipse workspace for example, *<Workspace>\.metadata\.plugins\com.sap.custdev.projects.fbs.slc.dataloader.core*
-conDriver	JDBC driver jar file path. For e.g. conDriver *D:\Database Drivers\mssql\sqljdbc4-4.0.jar*
-psConsoleLog	Prints logs on console. *psConsoleLog*

Parameters	Sample Value
-conName	Display name of the dataloader connection setting to be used. You must specify the following attribute from dataloader connection file. *displayName="con"* conName *con*
-conPassword	Password for the source backend system is specified in the dataloader connection setting. conPassword *mypassword*
-backendPassword	Password for the user specified for source backend system in the dataloader connection settings. backendPassword *backendpassword*
-createTables	Specify this argument if create table needs to be run while running dataloader. There is no value required for this argument. To run the full initial download, it is recommended to specify this argument. createTables
-initialDownload	Dataloader runs in initial and delta download mode. In SME only initial mode is supported. Hence *initialDownload* must be specified. No value is required for this argument.
-eccCfgExtractorFlag	This flag enables download from *Comm Tables* and must be specified with true argument. -eccCfgExtractorFlag *TRUE*

Example

The usage of a batch file (dataloader.bat) file is recommended.

The application will return an exit code *(see echo %ERRORLEVEL% in example cmd)*.

A batch/command script could look like:

```
@echo off

eclipse\eclipsec -vm "%JRE_HOME%\bin" --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.dataloader.headless.dataloaderapplication -noSplash -console
-psConsoleLog -conDriver "D:\DBdrivers\Database_Drivers\mssql\sqljdbc4-4.0.jar" -conName "con"
-conPassword "password" -backendPassword "bkpassword" -conFile "D:\DBdrivers\dataloader.xml"
-createTables -initialDownload
```

```
REM -Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005

@echo EXIT CODE: %ERRORLEVEL%
```

Exit Codes

0	All ok, Dataloader executed successfully
303	Dataloader settings file not found. Please Specify conFile
304	Issues while loading dataloader settings
307	Database driver not found
308	BackendPassword not specified
309	ConPassword not specified

2.10.6.1 Executing Headless Dataloader

To execute the script for headless dataloader, you must call the `eclipse.exe` or `eclipsec.exe` (without additional console) in your eclipse installation directory, with the required arguments.

> → Recommendation
>
> Write a batch (`.bat`) or command (`.cmd`) file that calls the executable.

2.10.6.1.1 Eclipse-Specific Arguments

Argument Name	**Argument Description**
`--launcher.suppressErrors`	Suppress errors Arguments must be set before `vmargs` (normally one of the first arguments)
`-application com.sap.custdev.projects.fbs.slc.sme.extvartable.upload.headless.externalvarianttableapplication`	The application to run Value (application ID) is separated from the argument by a single space.

Argument Name	Argument Description
`-noSplash`	Do not show the splash screen
`-console`	Show a console window
`-consoleLog`	Log messages to the console
`-vmargs`	Java VM arguments. This must be the last Eclipse argument because all arguments after this one will be parsed as Java VM arguments. For example, `-vmargs -Xms128M -Xmx512M` , `-XX:PermSize=128M -XX:MaxPermSize=256M` , etc.

> ℹ Posting Instructions
>
> For a full list of the supported arguments, see the Eclipse documentation .

2.10.6.1.2 Dataloader Application-Specific Arguments

> ℹ Note
>
> Use quotation marks ("") for values that contain spaces such as `-conDriver "D:\Database Drivers\mssql\sqljdbc4-4.0.jar`.

Argument Name	Example	Argument Description
`-conFile`	`-conFile "D:\DBdrivers\dataloader.xml"`	The required path to the dataloader settings file to use.
`-conDriver`	`-conDriver "D:\DBdrivers\Database_Drivers\mssql\sqljdbc4-4.0.jar"`	The path to the connection driver `jar` file.
`-psConsoleLog`	`-psConsoleLog`	Print logs on console.
`-conName`	`-conName "con"`	Display name of the dataloader connection setting to be used.
`-conPassword`	`-conPassword "mypassword"`	Connection password for the destination database to be used for data download.
`-backendPassword`	`-backendPassword "backendpassword"`	Password for the user specified for source backend system in the dataloader connection settings.

Argument Name	Example	Argument Description
`-createTables`	`-createTables`	Specify this argument if create table needs to be run while running the dataloader. There is no value required for this argument.
`-initialDownload`	`-initialDownload`	Specify this argument if initial download needs to be run while running the dataloader. No value required for this argument.
`-eccCfgExtractorFlag`	`-eccCfgExtractorFlag “TRUE”`	This argument is used to specify the value for ECC cfg extractor flag from SME. (com.sap.sxe.dataloader.GET_KB_FROM_CFGEXTRACTOR)

2.10.6.1.3 Example: Headless Dataloader Execution

> → Recommendation
>
> We recommend using a batch (`dataloader.bat`) file.

The application will return an exit code (refer to `echo %ERRORLEVEL%` in example command.

A batch or command script could appear as follows:

```
@echo off
eclipse\eclipsec -vm "%JRE_HOME%\bin" --launcher.suppressErrors -application
com.sap.custdev.projects.fbs.slc.dataloader.headless.dataloaderapplication
-noSplash
-console
-psConsoleLog -conDriver "D:\DBdrivers\Database_Drivers\mssql\sqljdbc4-4.0.jar"
-conName "con" -conPassword "password" -backendPassword "bkpassword"
-conFile "D:\DBdrivers\dataloader.xml"-createTables
-initialDownloadREM -Xdebug -Xnoagent -Djava.compiler=NONE-
Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005@echo EXIT CODE:
%ERRORLEVEL%
```

2.10.6.1.4 Exit Codes for Dataloader Execution

The exit codes for headless dataloader execution are given in the table below:

Exit Code	Description
0	All OK; Dataloader executed successfully.
303	Dataloader settings file not found. Please specify conFile.
304	Issues while loading dataloader settings.
307	Database driver not found.
308	`backendPassword` not specified.
309	`conPassword` not specified

3 Solution Configuration Environment

Use

The solution configuration environment is part of the SAP Solution Sales Configuration application. It is used to define the configuration of a solution in a business document, such as a sales quotation or a sales order, and is integrated with sales order processing transactions in the following systems:

- Hybris Commerce Suite
- SAP S/4HANA

Runtime Component

At run time, SAP Solution Sales Configuration comprises of the following components:

- A configuration UI: Launched from a sales application within the context of a sales document. It interacts with the engine via a command layer with well-documented APIs.
- KBO - knowledge base orchestration layer: Enables multiple discrete knowledge bases to be accessed dynamically as required during a configuration session.
- SPE - Sales Pricing Engine: Accesses the pricing master data to calculate pricing.
- SCE - Sales Configuration Engine: Accesses knowledge base run time versions in the back-end database (if not already cached in memory). These can be solution knowledge bases generated from the solution modeling environment as well as product knowledge bases from.

Components of the SCE:

Component	Description
PMS - pattern matching	Responsible for identifying the objects in the configuration that are "in scope", as defined by the objects/condition sections of constraints and rules. This process is called pattern matching. In addition, this component is responsible for forward chaining. That is, "firing" the constraints whose patterns are matched and altering the content of the configuration session, adding instances, or setting characteristic values.

Component	Description
TMS - truth maintenance	Tracks the facts asserted into the configuration session, along with their justifications. In addition, this component is responsible for retraction. That is, if a particular fact A is the justification for another fact B, and A is no longer true (for example, the user changes an entered value), the justification must be removed/retracted. Facts that are no longer justified must be removed from the configuration session. This may mean changing a characteristic value setting or deleting an instance.
DDB - dynamic database	This is the content of the configuration session. The PMS and TMS invoke methods on the `ddb_inst` class to request action in the configuration session, to set a value, or to create or remove an instance.

> i Note
>
> An event is logged for each action taken by these three components. The tracing and profiling capabilities of the engine support recording of these events, which are used in the solution modeling environment and, if enabled, in the production run time engine.

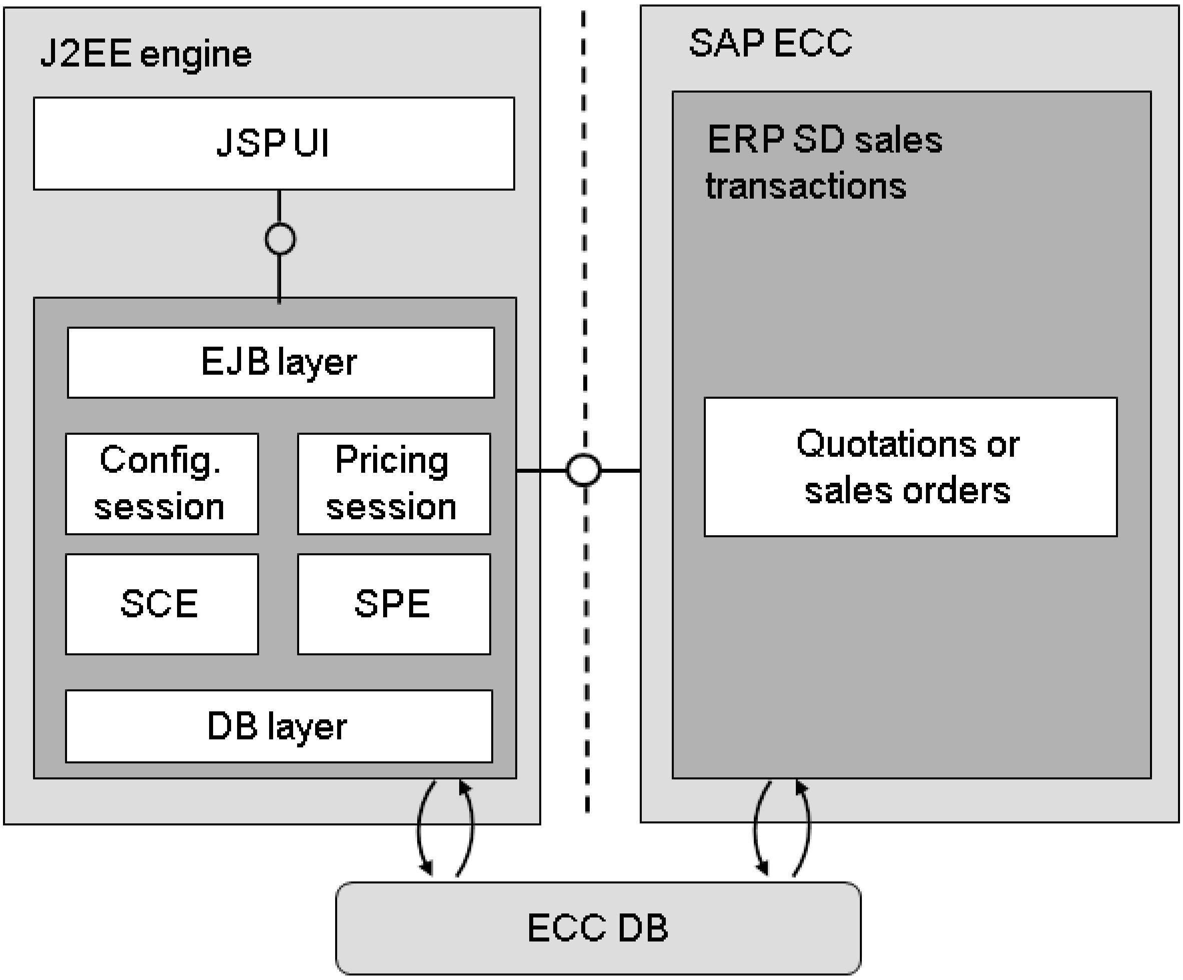
Integration

The solution configuration environment is built and packaged as an Enterprise Java Beans (EJB) component based on Java 2 Enterprise Edition (J2EE) standards. It is primarily intended to be deployed on an SAP J2EE NetWeaver Server.

The solution configuration environment includes the following components:

- Database (DB) Layer
- Sales Configuration Engine (SCE)
- Sales Pricing Engine (SPE)
- Configuration Session: This component encapsulates all of the configuration engine-related functionality and exposes the configuration engine commands to calling applications.
- Pricing Session: This component encapsulates all of the pricing-related functionality and exposes the pricing engine commands to calling applications.
- EJB Layer
- JavaServer Pages (JSP) User Interface (UI)

The following figure illustrates the architecture of the solution configuration environment when it is integrated with a SAP S/4HANA system:



Architecture for Integrating Solution Configuration Environment with SAP S/4HANA

The solution configuration environment for SAP Commerce is built and packaged as a Java Archive (.jar), which is added to the SAP Commerce class path. The Hybris service layer directly communicates with the Solution Configuration Engine (ConfigSession) "in-process", giving the best performance using Java native "call-by-reference".

The solution configuration environment integrates with the following business processes:

- Configure-to-Order in SAP S/4HANA
- Vehicle Management in SAP S/4HANA
- Procurement in SAP S/4HANA

When you choose to edit a configurable product in a sales document, the system opens the solution configuration UI. When you accept your configuration and close the UI, you are returned to the transaction document (quote, order, and vehicle), and the configuration result is reflected in the document, with any sales-relevant components listed as sub-line items. A saved configuration can be reopened, changed, and saved in the sales document, as necessary.

Features

- **Advanced Mode Configuration**
  SAP systems provide two ways of modeling solutions: compatible mode and advanced mode. Compatible mode is sufficient to model simple product configurations; however, advanced mode is required to model more complex solution configurations.
  SAP Internet Pricing and Configurator (IPC) is a component that is integrated into several standard SAP systems, including SAP CRM, SAP ECC, and SAP S/4HANA. While the IPC can generate an advanced mode configuration, the configuration result is not integrated with the processes outside the IPC. Therefore, the advanced mode configuration result generated by the IPC cannot be used in the downstream standard business processes. SAP Solution Sales Configuration overcomes this limitation by providing advanced mode configuration capabilities that are fully integrated with the standard configure-to-order business processes. For more information, see .
- **Starting a Configuration from a Component**
  In addition to the top-down approach, where you add a solution to a sales document, and then configure the component parts of the solution, you can also begin a solution configuration by adding a component of the solution to a sales document. When you choose to configure the component, the system recognizes that it is part of a solution (based on your Customizing settings), and initializes the configuration using the knowledge base for the solution.
- **Complex Relationships Between Components**
  A bill of material (BOM) can be used to define simple relationships where one component comprises several subcomponents. SAP Solution Sales Configuration allows you to define more complex relationships between the components of a solution, for example, one-to-one and one-to-many relationships. These relationships are defined in the solution modeling environment as abstract data types (ADTs). Moreover, simple relationships between components can also be defined using ADTs. Therefore, SAP Solution Sales Configuration eliminates the need to maintain BOMs. The configuration engine interprets the ADTs and reflects them in the sales document. For more information, see .
- **Reference Characteristics**
  The system allows you to define reference characteristics as part of the solution model. The value of a reference characteristic can be used in the following ways:
  - Passed to the configuration as a context
  - Passed from the configuration to the sales document in the target system
- **Hard and Soft Ties in Follow-Up Documents**
  When you create follow-up documents for a quotation in SAP CRM, you can split the components of the solution into separate documents. The system allows you to control which components can be split into separate sales documents and which must remain together. For more information, see .
- **Interactive Pricing**
  During the solution configuration process, the pricing engine recalculates the total price of the solution after each change to the configuration. The total price of the current configuration and the delta price for each characteristic value are displayed in the user interface. Interactive pricing is an option that can be enabled or disabled in the configuration user interface. For more information, see Interactive Pricing and Delta Pricing [page 153].
- **Enhanced Solution Configuration User Interface (JSP UI)**
  The JSP UI is designed as an accordion with collapsible layers. It features a *Configuration* layer that contains a layer for each component in the solution.
  The following icons are used to indicate the status of each component and characteristic:
  - A green square indicates that the configuration is complete.

- A yellow triangle indicates that the configuration is incomplete.
- A gray diamond indicates that the field cannot be edited.

You can enable or disable the display of these icons using the *Settings* option in the *Configuration* layer. For more information about the functions in the JSP UI, see Creating Solution Configurations [page 170].

Customizing the Solution Configuration Engine

As a system administrator, you can also customize the behavior of the Solution Configuration Engine, by modifying the engine parameters. For more information about working with engine parameters, please refer to SAP Note 2291607 (Engine Parameters for Hybris SSC, SAP Solution Sales Configuration, and Solution Modeling Environment).

Related Information

Configure-To-Order in SAP S/4HANA [page 144]

3.1 Configure-To-Order in SAP S/4HANA

Use

You use this process to add a solution to a sales document and configure it in the SAP S/4HANA system.

> i Note
>
> SAP S/4HANA has an integrated configuration engine called the Variant Configurator, which supports only compatible-mode configurations, however. If you install SAP Solution Sales Configuration, the SAP S/4HANA system opens the SAP Solution Sales Configuration JavaServer Pages (JSP) user interface (UI) or runtime UI Composer UI depending on the setup, instead of the standard Variant Configurator user interface when you choose to configure a product (a solution) during the sales ordering process.
>
> An alternative to the Variant Configurator for compatible-mode configurations is the use of the SSC.

Prerequisites

- You have entered product master data and configuration master data.
- You have set up the product to be configured with the SAP Solution Sales Configurator via the transaction/ `SLCE/CFG_MAT`.

For detailed information regarding setting up the UI for a material, see **UI Composer Runtime UI**. You must follow this process even when you are only using the JSP UI to launch configuration.

Process

1. You create a sales document, such as a quotation.
2. You add a solution to the sales document.
   The system displays the solution as a line item in the sales document.
3. You choose to configure the solution.
   The system displays the configuration user interface for the solution and all of its component parts.
4. You configure the solution.
5. You accept the configuration.
   The system closes the configuration user interface and displays the sales document. Each sales-relevant component of the solution is displayed as a line item.

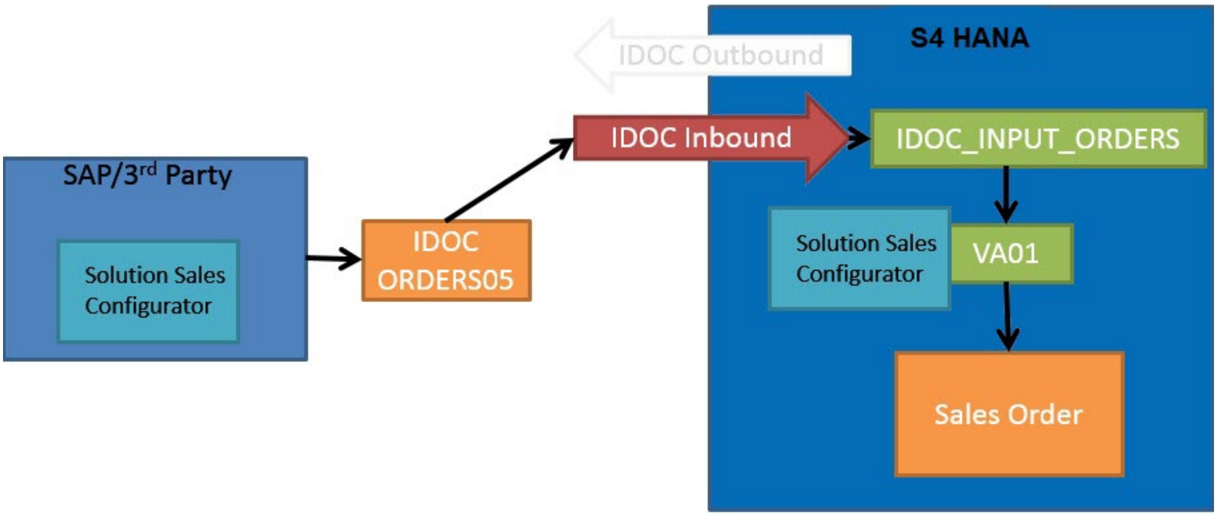
More Information

For more information about the functions in the configuration user interface, see Creating Solution Configurations [page 170].

3.1.1 IDOC Inbound Interface for Sales Order Creation

The business scenario for enhancement of S/4HANA IDOC inbound interface for sales order creation has the following steps:

1. You send `IDOC`s from a source to a target system where the source system may be an SAP- or third-party system with SSC add-on installed and the target system is an SAP S/4HANA system with the SSC add-on installed.
2. You use `ORDERS IDOC` inbound interface in the target S/4HANA system to create sales documents with the configured items.
3. You use basic `IDOC` type `/SLCE/ORDERS05` and the function module for `IDOC_INPUT_ORDERS` standard inbound for `IDOC` processing.



IDOC Processing

A S/4HANA solution configuration created by SSC operates in the following two modes at the same time:

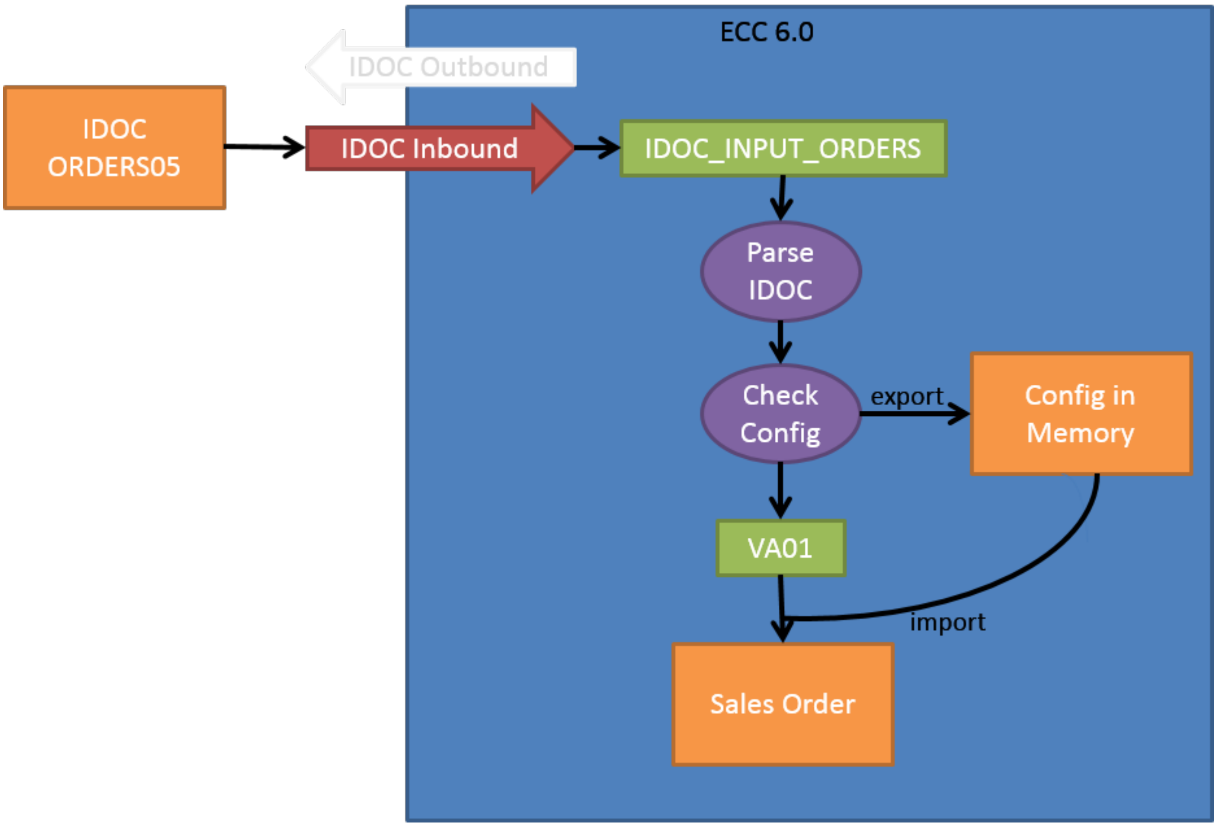
- The **Advanced Mode** model that contains the complete solution configuration, for example, including ADT characteristics
- The **Classic Mode** model, also known as the **Passive Receiving Structure (PRS)**, that contains only those parts of the configuration that are relevant for low-level production processes in SAP S/4HANA. The PRS is returned and stored by SSC in the `CBase` format.
  The results of the advanced mode configuration are returned and stored by SSC in the rich-config XML format.

Basic `IDOC` type `ORDERS05` provides a set of segments to transfer configuration results, such as, `E1CUCFG`, `E1CUINS`, etc. These segments must be used to transfer the PRS. As of today, the rich-config XML cannot be transferred by a standard IDOC type or BAPI.

If you create sales orders by IDOCs that contain only the PRS configuration results but not the rich-config XML, then the users cannot see the full advanced mode configuration when they try to open the configuration manually in that sales order.

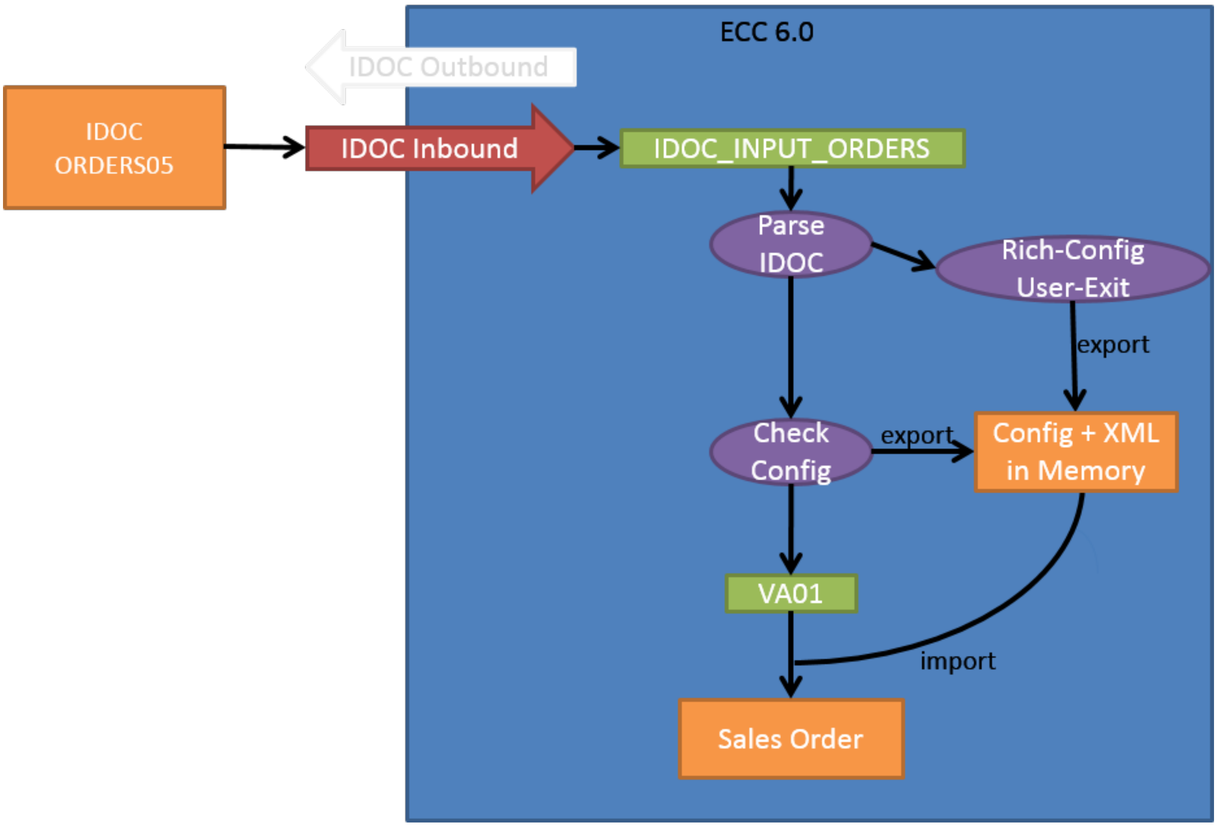
To solve this problem, SAP Notes 1996874 (Orders IDOC Inbound for SSC - config XML extension) and 2003665 (Orders IDOC Inbound for SSC - config XML extension (II)) enhance IDOC inbound processing in the following way:

- Enhancement of inbound function module `IDOC_INPUT_ORDERS`:
  - New user-exit that allows customers to read rich-config XML from own IDOC segment(s) or fetch the XML from an external source, for example, by an own web service
  - Functionality to export the retrieved XML(s) to ABAP memory
- Enhancement of sales document creation process:
  - If an XML is found in ABAP memory, it will be imported from memory
  - The XML will be assigned to the corresponding sales item



Before Implementation of Solution

After implementation of solution (Read: SAP Note 1996874 (Orders IDOC Inbound for SSC - config XML extension)).



After Implementation of Solution

You can use this process to create a sales document with items configured by IDOC. You will be able to view the advanced mode model by opening the configuration in SSC UI.

For more information about creating a sales order, refer to **Enhancement of S/4HANA IDOC Inbound Interface for Sales Order Creation**.

Related Information

Enhancement of S/4HANA IDOC Inbound Interface for Sales Order Creation [page 148]

3.1.1.1 Enhancement of S/4HANA IDOC Inbound Interface for Sales Order Creation

> i Note
>
> You need to use the enhancements provided in SAP Notes 1996874 (Orders IDOC Inbound for SSC - config XML extension) and 2003665 (Orders IDOC Inbound for SSC - config XML extension (II)).

Prerequisites

The following prerequisites should be met for creation of a sales order:

- FBS Solution Sales Configurator (`SSC`) add-on for `SAP S/4HANA` (`SLCE`) has been installed with the correct version and latest support package.
- SAP Note 2003665 (Orders IDOC Inbound for SSC - config XML extension (II)) has been implemented, including pre- and post-installation steps.
- The following notes need to be implemented as well:
  - SAP Note 1991156 (IDoc order creation: Sub-item without configuration tie)
  - SAP Note 1996874 (Order IDOC Inbound for SSC - config XML extension)
  - SAP Note 2006212 (Additional enhancement point in function group V45CU)
  - SAP Note 1923474 (Items on the Sales Document are deleted and recreated.)
  - SAP Note 1880466 (ECC:Create With Reference results in Exception on Config UI)

Assumptions

The following assumptions are considered during the process of enhancement:

- You have already customized `ORDERS IDOC` inbound interface in the target S/4HANA system and are able to create sales documents by basic `IDOC` type `ORDERS05`.
- The sales documents are already created with the correct PRS configuration and only the rich-config XML is missing

Out of Scope

The out of scope processes and functions are especially but not limited to the following:

- Transfer and storage of custom relations in table /SLCE/IRT
- IDOC outbound interface
- IDOC interface for order change
- Other IDOC interfaces or BAPIs

Related Information

IDOC Inbound Interface for Sales Order Creation [page 145]

3.1.2 Light Engineer-to-Order

Use

Light engineer-to-order (“light ETO”) enables you to split a complex configuration between the sales level and the engineering level, making the configuration process clearer. The processor of the production order BOM can also change the configuration on an individual basis in cases where a required change is too specific to be included in the configuration rules.

Process

1. The sales representative creates a sales order that contains a configurable material, either directly in SAP S/4HANA (transaction `VA01`).
2. The sales representative starts the sales configurator by choosing the *Item details: Configuration* pushbutton.
3. The sales configurator explodes the sales order BOM components.
4. The sales representative configures the components in the sales order BOM. The component may already be partly configured as the result of rules that have been defined in the system (see Defining Solution Dependencies [page 44]).

   > **i Note**
   >
   > The component configured here is also the header of the production order BOM and is, therefore, visible in both the solution sales configurator and the variant configurator.

5. The sales representative accepts the configuration to transfer the result to the production order. The sales order is then saved and the sales order number communicated to the production engineer.
6. The production engineer opens the order BOM, for example, in transaction `CU51` by entering the order number and the item number that he or she wants to configure.

   > **i Note**
   >
   > All standard engineering-to-order processes can be used (for example, transactions `CSKB`, `CS6x`, and `CU51`).

   The variant configurator opens and displays the header level of the engineering structure.
7. The production engineer navigates to the subitems and configures the relevant data.
8. The production engineer clicks the configuration tree structure to explode the BOM again. If there are any dependencies between the data (for example, the engineer has specified a component but not the component type), the system requests the missing data.
9. The production engineer saves the data.

Example

You have a material (`MY_SYSTEM`) that has been configured with the components `RACK`, `SUBRACK`, and `DEVICE`. The device slots into the subrack, which in turn slots into the rack.

The sales representative creates a sales order for the material `MY_SYSTEM` and starts the sales configurator. After the sales representative has specified the general data, the system displays three more dropdown areas - one for `RACK`, one for `SUBRACK`, and one for `DEVICE` (whereby `SUBRACK` and `DEVICE` are classes only). The sales representative configures the data for `RACK` and saves the order.

The production engineer opens the order BOM with the item number that is to be configured (in this example, item 20). The engineer navigates to `SUBRACK` in the configuration tree structure and enters the required data. He or she then explodes the BOM again and repeats the process to configure the required data for `DEVICE`.

3.1.2.1 Changing Bills of Material

Use

In certain cases, a production engineer may want to change the data in a bill of material for a specific production order. The required change is too specific to be included in the configuration rules, and so the engineer must change the data manually.

Procedure

1. Display the order BOM in transaction `CU51` and open the configuration result by choosing the *Result* pushbutton (`CTRL` + `F9`).
2. Select the checkbox for the component that you want to change and choose the *Item in Full* pushbutton (`SHIFT` + `F4`).
3. Change the required data, return to the configuration result, and save.

> **i Note**
>
> You can also add further items to the BOM by choosing the *Insert* pushbutton (`SHIFT` + `F1`). Examples of items that you can add include documents, texts, and compatible units.

3.2 Configure-To-Order in Hybris

Use

You use this process to configure a solution in Hybris and then add it to your shopping cart.

Prerequisites

- You have created product master data and configuration master data.
- You have created the solution product in your Hybris catalog.
- You have downloaded the master data from the SAP back-end system to the Hybris database using the Data Loader.
  For more information on using the Data Loader in Hybris, see the Hybris help portal and *Integrations and Data Management* > *SAP Integrations* > *SAP Product Configuration (On-Premise Edition)* > *Installing Product Configuration* > *Post Installation Steps* > *Configuring and Running the Data Loader* > *Loading Configuration Master Data through Data Loader*.

Process

1. You search for the solution product you require in the catalog.
2. You choose to configure the solution.
   The system displays the configuration user interface.
3. You configure the solution.
4. You add the solution to your shopping cart.
   The system closes the configuration user interface and displays your shopping cart.
5. You start the checkout process.

More Information

For more information about the functions in the configuration user interface, see Creating Solution Configurations [page 170].

3.2.1 Adding Related Products to the Solution

Context

SAP Solution Sales Configuration enables you to manually add related products to your solution.

Procedure

1. Open the *Find Related Products* dialog box.

   The system displays a list of related products. The default implementation of the search filter for “related products” uses the potential “non-part instances”of the solution model. You can enhance this with your own logic.
2. Select the product that you want to add to the solution.

   The system reports that the product has been added.
3. Continue to add further products or return to the solution.

Results

The system shows the added product in the section *Selected Products Related to This Solution*.

3.3 Interactive Pricing and Delta Pricing

Use

During the solution configuration process, the system calculates and displays the price of the solution after each change in the configuration, for example, when the value of a characteristic is changed, when an instance is deleted, and when an instance is added. This is known as interactive pricing.

To inform the user about the impact of a characteristic selection on the total price, the system also displays the surcharge price or the price reduction next to each characteristic value in the configuration user interface. This is known as delta pricing.

Integration

The solution configuration process uses configuration and pricing data from the source system. To access the data, therefore, the system must be connected to the relevant database. This connection is implemented as a static destination setting between the J2EE environment, where the configuration engine is deployed, and the source system.

To optimize performance, pricing and configuration data is cached at Java stack level.

Prerequisites

- You have maintained prices (list prices) for the required products.
- You have maintained one-to-one variant conditions in the solution configuration model to reflect the surcharge or the price reduction for each characteristic value.

> → Recommendation
>
> To optimize system performance, SAP recommends that you maintain a pricing procedure that is dedicated to interactive pricing purposes. This pricing procedure should hold only the necessary condition types and condition tables relevant for interactive pricing.

Activities

Interactive Pricing

To calculate and display interactive prices, the system performs the following steps:

1. The order mapper interprets the solution configuration and converts it into a sales product structure that can be used by the sales pricing engine (SPE).
   The sales product structure is represented in the configuration result by the following types of instance relationship:
   - Sales abstract data type (ADT) characteristics (see Characteristic [page 23])
   - Compatible mode sales bill of material (BOM) explosions of the super BOM
2. The system passes the sales product structure to the SPE.
3. The system triggers the pricing determination process.
   The SPE calculates the total price of the solution, which is a summation of all the instance (product) list prices and all the variant conditions attached to the selected characteristic values.
4. The system passes the result of the pricing determination process to the configuration user interface and updates the display.

> ⚠ Caution
>
> The pricing procedure used for interactive pricing should not propagate the prices of subitems to the root item price.

Delta Pricing

To calculate and display delta prices, the system performs the following steps:

1. The surcharge or reduction price for each pricing-relevant characteristic value is retrieved and displayed next to each characteristic value in the user interface.
2. After each change to the configuration, the surcharge or reduction price is recalculated based on the user's last selection.
   It is assumed that a one-to-one characteristic value/variant condition assignment is used. Therefore, the new delta prices only need to be calculated for the values of the characteristic that was changed last.
   The following formula is used to calculate the delta price:
   characteristic value price = variant condition value price - selected characteristic value price
   If the value is positive, the delta price is a surcharge. If the value is negative, the delta price is a reduction.

More Information

> **i** Note
>
> You can enable and disable interactive pricing and delta pricing using a checkbox in the configuration user interface. Unlike JSP UI, UI Composer runtime UI offers only header level price display as of now.

For more information about using the configuration user interface, see Creating Solution Configurations [page 170].

3.3.1 Pricing Formula and User Exits

Pricing is a highly customizable and configurable engine. However, in some cases, the regular features and functionalities provided by the pricing engine are not sufficient. In such cases, it is possible to meet the special business requirements by using **Pricing Formulas** and **User Exits**. These are custom functions that allow customization of the default behavior of existing pricing conditions.

For more information about this, refer to the SAP Help Portal and navigate to *Basic Functions and Master Data in SD Processing (SD-BF)* › *Basic Functions in SD* › *Pricing and Conditions*.

3.3.1.1 Available User Exits and APIs

Here, you can find all the relevant information related to the available pricing-related user-exit types. First, the standard features are explained and then the different types of user exits. The parameters that form the interface between pricing and user exits are also described briefly.

3.3.1.1.1 Logging Capabilities

For customer pricing user exits, there is an easy way to include fast logging. The `com.sap.spe.base.logging.UserexitLogger` class implements two methods for logging debug messages or error messages. Logging is fast and done only if the appropriate log level is reached, which you can define at runtime.

ZSpecialRoundingValueFormula (shorten)

```
package your.company.pricing.userexits;

import com.sap.spe.base.logging.UserexitLogger;
```

```
[..]
public class ZSpecialRoundingValueFormula extends ValueFormulaAdapter {

    private static UserexitLogger userexitlogger =
        new UserexitLogger(ZSpecialRoundingValueFormula.class);

    public BigDecimal overwriteConditionValue(IPricingItemUserExit item,
            IPricingConditionUserExit condition) {
[..]
        userexitlogger.writeLogDebug("old cond value: "
                + val.getValueAsString());
[..]
    }
}
```

Line No.	Description
8	Create a static instance of the `UserexitLogger` class. As constructor parameter, pass the actual class.
13	Use `writeLogDebug(String s)` or `writeLogError(String s)` to log the string s in the log.

3.3.1.1.2 Condition Base Formula

The condition base formula can be used to overturn the automatically calculated base value of a condition. This type of user exit must be assigned in Customizing to the user exits type `BAS` (condition base formula).

This user exit is called after the condition base value has been calculated for each pricing condition. The user exit class must be inherited from `BaseFormulaAdapter` and implement method `overwriteConditionBase`. The `overwriteConditionBase` method has the parameters `pricingItem` and `pricingCondition`, which represents the item and the actual condition.

If this method returns a null object reference, pricing will keep the base value that is called automatically.

ZSpecialBaseFormula

```
package your.company.pricing.userexits;

import java.math.BigDecimal;

import com.sap.spe.base.logging.UserexitLogger;
import com.sap.spe.conversion.IDimensionalValue;
import com.sap.spe.pricing.transactiondata.userexit.IPricingConditionUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingItemUserExit;
import com.sap.spe.pricing.transactiondata.userexit.BaseFormulaAdapter;

public class ZSpecialBaseFormula extends BaseFormulaAdapter {

    private static UserexitLogger userexitlogger =
        new UserexitLogger(ZSpecialBaseFormula.class);
    public BigDecimal overwriteConditionBase(IPricingItemUserExit pricingItem,
            IPricingConditionUserExit pricingCondition) {

            BigDecimal result;

            userexitlogger.writeLogDebug("old cond base: "
                    + pricingCondition.getConditionBase().getValueAsString());

            // double the base value
            result = pricingCondition.getConditionBase().getValue().
                         multiply(new BigDecimal("2"));

            userexitlogger.writeLogDebug("new cond base: " + result);

            return result;
        }
}
```

Line No.	Description
11	Extend/subclass the API `BaseFormulaAdapter`.
16	Overwrite the implementation of the `overwriteConditionBase` method.
25	Change the value of the automatically determined condition base.
30	Return the changed condition base value.

3.3.1.1.3 Item Calculation Begin Formula

This seldom-used user exit is available to change the document and item if necessary before item pricing takes place. This type of user exit must be assigned in Customizing to user exit type `CAB` (Item Calculation Begin Formula).

The user exit class must be inherited from `PricingItemCalculateBeginFormulaAdapter`. It passes a reference to the pricing document (`prDocument`) and the item (`prItem`).

ZSpecialCalculationBeginFormula

```
package your.company.pricing.userexits;

import com.sap.spe.pricing.transactiondata.userexit.IPricingDocumentUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingItemUserExit;
import
com.sap.spe.pricing.transactiondata.userexit.PricingItemCalculateBeginFormulaAdap
ter;

public class ZSpecialCalculationBeginFormula extends
PricingItemCalculateBeginFormulaAdapter {
{
    private int stepNumber, counter;

    public void calculationBegin(IPricingDocumentUserExit prDocument,
         IPricingItemUserExit prItem) {

    stepNumber = 10;
    counter = 1;

    prDocument.setZeroPriceActive(true);

    }
}
```

Line No.	Description
7	Extend the API `PricingItemCalculateBeginFormulaAdapter`
12	Overwrite the implementation of the `calculationBegin` method
18	Set the document to accept zero prices as valid prices

3.3.1.1.4 Item Calculation End Formula

This seldom-used user exit is available to change the document and item if necessary after item pricing has taken place. This type of user exit must be assigned in Customizing to user exit type `CAE` (Item Calculation End Formula).

The user exit class must be inherited from `PricingItemCalculateEndFormulaAdapter`. It passes a reference to the pricing document (`prDocument`) and the item (`prItem`).

ZSpecialCalculationEndFormula

```
package your.company.pricing.userexits;

import com.sap.spe.pricing.transactiondata.userexit.IPricingDocumentUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingItemUserExit;
import
com.sap.spe.pricing.transactiondata.userexit.PricingItemCalculateEndFormulaAdapte
r;

public class ZSpecialCalculationEndFormula extends
PricingItemCalculateEndFormulaAdapter {

    private int stepNumber, counter;

    public void calculationEnd(IPricingDocumentUserExit prDocument,
        IPricingItemUserExit prItem) {

    stepNumber = 10;
    counter = 1;
    prItem.findPricingCondition(stepNumber, counter).setConditionControl('A');

    }
}
```

Line No.	Description
7	Extend the API `PricingItemCalculateEndFormulaAdapter`
11	Overwrite the implementation of the `calculationEnd` method
16	Set the condition control of the pricing condition at `stepNumber` 10 to Automatic A

3.3.1.1.5 Configuration Formula

This seldom-used user exit is called when the product configuration process creates subitems. This type of user exit must be assigned in Customizing to user exit type `CFG` (Configuration Formula), which is called for subitems created by SCE.

The user exit class must be inherited from `SPCSubItemCreatedByConfigurationFormulaAdapter`. For each subitem, method `isRelevantForPricing` is called and a reference to the new subitem and the configuration instance is passed.

ZSpecialConfigurationFormula

```
package your.company.pricing.userexits;

import com.sap.spc.document.userexit.ISPCItemUserExitAccess;
import com.sap.sce.front.base.Instance;
import
com.sap.spc.document.userexit.SPCSubItemCreatedByConfigurationFormulaAdapter;

public class ZSpecialConfigurationFormula extends
    SPCSubItemCreatedByConfigurationFormulaAdapter {

    public boolean isRelevantForPricing(ISPCItemUserExitAccess subItem,
            Instance instance) {

    return subItem.isRelevantForPricing();
    }
}
```

Line No.	Description
8	Extend the API `SPCSubItemCreatedByConfigurationFormulaAdapter`
10	Implement the `isRelevantForPricing` method
13	Set the configuration subitem to pricing-relevant or not

3.3.1.1.6 Condition Init Formula

After a pricing condition has been initialized, it can be changed with this user exit, which is called whenever an internal condition (a transactional object or business entity) is created. This type of user exit must be assigned in Customizing to user exit type `CNI` (Condition Init Formula).

The user exit class must be inherited from `PricingConditionInitFormulaAdapter` and must overwrite the method `init`. It allows the new condition to be changed (parameter `prCondition`).

ZSpecialConditionInitFormula

```
package your.company.pricing.userexits;

import java.math.BigDecimal;

import com.sap.spe.pricing.transactiondata.userexit.IPricingConditionUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingDocumentUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingItemUserExit;
import
com.sap.spe.pricing.transactiondata.userexit.PricingConditionInitFormulaAdapter;

public class ZSpecialConditionInitFormula extends
PricingConditionInitFormulaAdapter {

    public void init(IPricingDocumentUserExit prDocument, IPricingItemUserExit
prItem,
            IPricingConditionUserExit prCondition) {

    if (prCondition.getConditionTypeName() != "0PR0"
                     && prCondition.getChangeOfRateAllowed())
    prCondition.setConditionRateValue(new BigDecimal("2"));
    }
}
```

Line No.	Description
10	Extend the API `PricingConditionInitFormulaAdapter`

Line No.	Description
12	Implement the `init` method
17	Set the condition rate to 2

3.3.1.1.7 Copy Formula

While a document is being copied, the pricing condition can be fixed or other changes can take place if required. This type of user exit must be assigned in Customizing to user exit type `CPY` (Copy Formula).

This user exit is called during the copying process. The user exit class must be inherited from class `PricingCopyFormulaAdapter` and implement method `pricingCopy`. Parameters `pricingDocument`, `pricingItem`, and `pricingCondition` are references to the target document, item, and condition. The pricing type describes what should happen to the pricing result when new pricing takes place. The parameter `copyType` is a reference to the Customizing used for the copy process; `sourceSalesQuantity` contains the old quantity of the source item.

ZSpecialCopyFormula

```
package your.company.pricing.userexits;

import com.sap.spe.conversion.IQuantityValue;
import com.sap.spe.pricing.customizing.ICopyType;
import com.sap.spe.pricing.customizing.IPricingType;
import com.sap.spe.pricing.transactiondata.userexit.IPricingConditionUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingDocumentUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingItemUserExit;
import com.sap.spe.pricing.transactiondata.userexit.PricingCopyFormulaAdapter;

public class ZSpecialCopyFormula extends PricingCopyFormulaAdapter {

    public void pricingCopy(IPricingDocumentUserExit pricingDocument,
IPricingItemUserExit
                            pricingItem, IPricingConditionUserExit
pricingCondition,
                            IPricingType pricingType, ICopyType copyType,
IQuantityValue
                            sourceSalesQuantity) {

      // fix condition value and base
      pricingCondition.setConditionControl('E');
        }
}
```

Line No.	Description
11	Extend the API `PricingCopyFormulaAdapter`

Line No.	Description
13	Overwrite the implementation of the `pricingCopy` method
19	Fix the conditions value and base by setting the `conditionControl` to E

3.3.1.1.8 Document Init Formula

After a pricing document has been initialized, it can be changed with this user exit, which is called when a new pricing document is created. This type of user exit must be assigned in Customizing to user exits type `DOI` (Document Init Formula).

The user exit class must be inherited from class `PricingDocumentInitFormulaAdapter` and implement method `init`. A reference to the new document is passed.

ZSpecialDocumentInitFormula

```
package your.company.pricing.userexits;

import com.sap.spe.pricing.transactiondata.userexit.IPricingDocumentUserExit;
import
com.sap.spe.pricing.transactiondata.userexit.PricingDocumentInitFormulaAdapter;

public class ZSpecialDocumentInitFormula extends
PricingDocumentInitFormulaAdapter {

    public void init(IPricingDocumentUserExit prDocument) {
      if (!prDocument.isAlwaysPerformingGroupConditionProcessing())
        prDocument.setAlwaysPerformingGroupConditionProcessing(true);
    }
}
```

Line No.	Description
6	Extend the API `DocumentInitFormula`
8	Overwrite the implementation of the `init` method
10	Set group condition processing to active

3.3.1.1.9 Group Key Formula

This user exit can be used to replace the automatically determined condition value. This type of user exit must be assigned in Customizing to user exit type `VAL` (Condition Value Formula).

This user exit is called after the condition value has been calculated for each pricing condition.
The user exit class must be inherited from class `ValueFormulaAdapter` and implement at least `overwriteConditionValue`. If group condition processing is enabled for the condition type, the implementation of method `overwriteGroupConditionValue` is possible. Both methods can return null to indicate that the original value is to be taken.

ZSpecialRoundingValueFormula

```
package your.company.pricing.userexits;

import java.math.BigDecimal;

import com.sap.spe.base.logging.UserexitLogger;
import com.sap.spe.conversion.ICurrencyValue;
import com.sap.spe.pricing.transactiondata.userexit.IGroupConditionUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingConditionUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingDocumentUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingItemUserExit;
import com.sap.spe.pricing.transactiondata.userexit.ValueFormulaAdapter;

public class ZSpecialRoundingValueFormula extends ValueFormulaAdapter {

    private static UserexitLogger userexitlogger =
        new UserexitLogger(ZSpecialRoundingValueFormula.class);

    public BigDecimal overwriteConditionValue(IPricingItemUserExit item,
            IPricingConditionUserExit condition) {
        BigDecimal result;

        ICurrencyValue val = condition.getConditionValue();
        userexitlogger.writeLogDebug("old cond value: "
                + val.getValueAsString());

        result = val.getValue().setScale(0, BigDecimal.ROUND_HALF_UP);

        BigDecimal qnt = item.getProductQuantity().getValue();
        qnt = qnt.divide(new BigDecimal("100"), 2, BigDecimal.ROUND_HALF_UP);

        userexitlogger.writeLogDebug("new cond value: " + result.subtract(qnt));

        return result.subtract(qnt);
    }

    public BigDecimal overwriteGroupConditionValue(
            IPricingDocumentUserExit item, IGroupConditionUserExit condition) {
        // do nothing
        return null;                    }
}
```

Line No.	Description
13	Extend the API `ValueFormulaAdapter`
18	Overwrite the implementation of the `overwriteConditionValue` method
33	Change the value of the automatically determined condition value
33	Return the changed condition value
36	Overwrite the implementation of the `overwriteGroupConditionValue` method
39	Return null to keep the automatically calculated value

3.3.1.1.10 Item Init Formula

After the pricing item has been initialized, it can be changed with this user exit, which is called when a new pricing item is created. This type of user exit must be assigned in Customizing to user exit type `ITI` (Item Init Formula).

The user exit class must be inherited from class `PricingItemInitFormulaAdapter` and implement method `init`. A reference to the document and to the new item is passed.

ZSpecialItemInitFomula

```
package your.company.pricing.userexits;

import com.sap.spe.pricing.transactiondata.userexit.IPricingDocumentUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingItemUserExit;
import
com.sap.spe.pricing.transactiondata.userexit.PricingItemInitFormulaAdapter;

public class ZSpecialItemInitFomula extends PricingItemInitFormulaAdapter {

    public void init(IPricingDocumentUserExit prDocument, IPricingItemUserExit
prItem) {
      if (prItem.isStatistical())
        prItem.setExclusionFlag('$');
    }
}
```

Line No.	Description
7	Extend the API `ZSpecialItemInitFomula`

Line No.	Description
9	Overwrite the implementation of the `init` method
11	Set the item exclusion flag

3.3.1.1.11 Pricing Init

In previous releases, this user exit was called `CRMDocumentStandardExit` where it was used mainly to pass header attributes to be used in method `initializeDocument`. As of Release 2.0 SP5, these attributes can be customized. Pricing Init user exits can now be used only to set the unit of rounding to the smallest unit of a currency. This type of user exit must be assigned in Customizing to the user exit type `PRI` (Pricing Init).

This user exit is called when a new pricing document is created. The user exit class must be inherited from class `PricingInitFormulaAdapter` and must implement method `initializeDocument`. This method has parameter `documentUserExitAccess`, which represents the pricing document.

ZPricingInit

```
package your.company.pricing.userexits;

import com.sap.spe.document.userexit.IDocumentUserExitAccess;
import com.sap.spe.document.userexit.PricingInitFormulaAdapter;

public class ZPricingInit extends PricingInitFormulaAdapter {

    public void initializeDocument(IDocumentUserExitAccess
documentUserExitAccess) {

      documentUserExitAccess.setUnitToBeRoundedTo(20);
    }
}
```

Line No.	Description
6	Extend the API `PricingInitFormulaAdapter`
8	Overwrite the implementation of the `initializeDocument` method
10	Set the rounding unit to 20

3.3.1.1.12 Pricing Prepare

In previous releases, this user exit was called `CRMItemStandardExit`. Pricing Prepare user exits can be used to add header and/or item attributes to be used during the pricing process. These attributes can now be customized. This type of user exit must be assigned in Customizing to the user exit type `PRP` (Pricing Prepare).

The Pricing Prepare user exit is called when creating a new pricing item and when new pricing takes place. The user exit class must be inherited from class `PricingPrepareFormulaAdapter` and must implement method `addAttributeBindings`. This method has parameter `itemUserExitAccess`, which represents the pricing item.

ZPricingPrepare

```
package your.company.pricing.userexits;

import com.sap.spe.document.userexit.IItemUserExitAccess;
import com.sap.spe.document.userexit.PricingPrepareFormulaAdapter;

public class ZPricingPrepare extends PricingPrepareFormulaAdapter {

  public void addAttributeBindings(IItemUserExitAccess itemUserExitAccess) {

    itemUserExitAccess.addAttributeBinding("ZLAND", "DE");
    }
}
```

Line No.	Description
6	Extend the API `PricingPrepareFormulaAdapter`
8	Overwrite the implementation of the `addAttributeBindings` method
10	Set the attribute `ZLAND` to the value “DE”.

3.3.1.1.13 Requirement

This user exit is used during condition determination at pricing procedure step/counter level and at condition access step level. This type of user exit must be assigned in Customizing to the user exit type `REQ` (Requirement).

The user exit class must be inherited from `RequirementAdapter` and implement method `checkRequirement`. If this method returns false, the actual access is not made.

ZSpecialRequirement

```
package your.company.pricing.userexits;

import com.sap.spe.base.logging.UserexitLogger;
import com.sap.spe.condmgnt.customizing.IAccess;
import com.sap.spe.condmgnt.customizing.IStep;
import com.sap.spe.condmgnt.finding.userexit.IConditionFindingManagerUserExit;
import com.sap.spe.condmgnt.finding.userexit.RequirementAdapter;

public class ZSpecialRequirement extends RequirementAdapter {

    private static UserexitLogger userexitlogger =
        new UserexitLogger(ZSpecialRequirement.class);

    public boolean checkRequirement(IConditionFindingManagerUserExit item,
        IStep step, IAccess access)
    {
        String zland = item.getAttributeValue("ZLAND");
        if (zland == null || zland.equals("")) {
          userexitlogger.writeLogError("ZLAND attribute missing");
             return false;
        } else {
             return zland.equals("US");
        }
     }
}
```

Line No.	Description
9	Extend the API `RequirementAdapter`
14	Overwrite the implementation of the `checkRequirement` method
17	Retrieve an attribute value to be used for the check
22	Return the check result: “true” to make the access, “false” not to make the access

3.3.1.1.14 Scale Base Formula

This user exit can be used to replace the automatically determined scale base. This type of user exit must be assigned in Customizing to the user exit type `SCL` (Scale Base Formula).

This user exit is called after the condition-scale base value has been calculated for a pricing condition. The user exit class must be inherited from class `ScaleBaseFormulaAdapter` and implement at least `overwriteScaleBase`. If group condition processing is enabled for the condition type, method `overwriteGroupScaleBase` can also be implemented. Both methods can return null to indicate that the original value is to be taken.

ZSpecialScaleBaseFormula

```
package your.company.pricing.userexits;

import java.math.BigDecimal;

import com.sap.spe.base.logging.UserexitLogger;
import com.sap.spe.pricing.transactiondata.userexit.IGroupConditionUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingConditionUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingDocumentUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingItemUserExit;
import com.sap.spe.pricing.transactiondata.userexit.ScaleBaseFormulaAdapter;

public class ZSpecialScaleBaseFormula extends ScaleBaseFormulaAdapter {

    private static UserexitLogger userexitlogger =
        new UserexitLogger(ZSpecialScaleBaseFormula.class);

    public BigDecimal overwriteScaleBase(IPricingItemUserExit item,
            IPricingConditionUserExit condition,
            IGroupConditionUserExit groupCondition) {

        userexitlogger.writeLogDebug("Old scale: " +
                       groupCondition.getConditionScale().getValueAsString());

        if (groupCondition.getConditionScale() != null) {
             return groupCondition.getConditionScale().getValue().setScale(0,
                           BigDecimal.ROUND_FLOOR);
        }
        else
        {
             return null;
        }
        }

        public BigDecimal overwriteGroupScaleBase(IPricingDocumentUserExit
document,
            IGroupConditionUserExit groupCondition) {

          if (groupCondition.getConditionScale() != null) {
              return groupCondition.getConditionScale().getValue().setScale(0,
                        BigDecimal.ROUND_FLOOR);
          }
          else
          {
              return null;
          }
        }
}
```

Line No.	Description
12	Extend the API `ScaleBaseFormulaAdapter`
17	Overwrite the implementation of the `overwriteScaleBase` method
25	Return the changed scale base value

Line No.	Description
34	Overwrite the implementation of the `overwriteGroupScaleBase` method
38	Return the changed scale base value

3.3.1.1.15 Condition Value Formula

This user exit can be used to replace the automatically determined condition value. This type of user exit must be assigned in Customizing to user exit type `VAL` (Condition Value Formula).

This user exit is called after the condition value has been calculated for each pricing condition.
The user exit class must be inherited from class `ValueFormulaAdapter` and implement at least `overwriteConditionValue`. If group condition processing is enabled for the condition type, the implementation of method `overwriteGroupConditionValue` is possible. Both methods can return null to indicate that the original value is to be taken.

ZSpecialRoundingValueFormula

```
package your.company.pricing.userexits;

import java.math.BigDecimal;

import com.sap.spe.base.logging.UserexitLogger;
import com.sap.spe.conversion.ICurrencyValue;
import com.sap.spe.pricing.transactiondata.userexit.IGroupConditionUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingConditionUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingDocumentUserExit;
import com.sap.spe.pricing.transactiondata.userexit.IPricingItemUserExit;
import com.sap.spe.pricing.transactiondata.userexit.ValueFormulaAdapter;

public class ZSpecialRoundingValueFormula extends ValueFormulaAdapter {

    private static UserexitLogger userexitlogger =
        new UserexitLogger(ZSpecialRoundingValueFormula.class);

    public BigDecimal overwriteConditionValue(IPricingItemUserExit item,
            IPricingConditionUserExit condition) {
        BigDecimal result;

        ICurrencyValue val = condition.getConditionValue();
        userexitlogger.writeLogDebug("old cond value: "
                + val.getValueAsString());

        result = val.getValue().setScale(0, BigDecimal.ROUND_HALF_UP);

        BigDecimal qnt = item.getProductQuantity().getValue();
        qnt = qnt.divide(new BigDecimal("100"), 2, BigDecimal.ROUND_HALF_UP);

        userexitlogger.writeLogDebug("new cond value: " + result.subtract(qnt));

        return result.subtract(qnt);
    }

    public BigDecimal overwriteGroupConditionValue(
            IPricingDocumentUserExit item, IGroupConditionUserExit condition) {
        // do nothing
        return null;
    }
}
```

Line No.	Description
13	Extend the API `ValueFormulaAdapter`
18	Overwrite the implementation of the `overwriteConditionValue` method
33	Change the value of the automatically determined condition value
33	Return the changed condition value
36	Overwrite the implementation of the `overwriteGroupConditionValue` method
39	Return null to keep the automatically calculated value

3.3.2 Modifying Pricing Context

If you want to change the pricing context for the header or item of a document, you can implement the `/SLCC/MODIFY_PRICING_CONTEXT` BadI. SAP delivers a default implementation of this BadI and customers can have their own implementation as this is a multiple use BadI.

3.4 Creating Solution Configurations

Use

This procedure explains how to use the main functions in the advanced-mode configuration user interface. The user interface is presented as an accordion with collapsible layers. Within the *Configuration* layer, there is a layer for each component in the solution.

Prerequisites

You have created a sales document, added an advanced mode product, and chosen to edit it.

Procedure

To access individual functions shown in the table, use the *Configuration* layer of the user interface:

Function	Navigation	More Information
Add a non-part instance	Choose *Add Non Part Instance*	The system displays a list of all the non-part instances for the solution. You can add one or more non-part instances by choosing the *Add Component* pushbutton next to the relevant component.
Enable or Disable Interactive Pricing	Select or deselect the *Interactive Pricing* checkbox	The system enables or disables both interactive pricing (display of the total price) and delta pricing (display of surcharges and price reductions).
Display the sales structure for the solution	Select the *Sales Structure* checkbox	The system displays the solution configuration in one of the following ways: • **Flat Structure** This view displays the non-part instances in a list and the bill of material (BOM) explosion as a hierarchical structure. The relationships between the non-part instances are not displayed. • **Sales Structure** This view displays the hierarchical relationships between the instances, which are interpreted from the sales abstract data types (ADTs) and the BOM explosion.

Function	Navigation	More Information
View or specify the configuration settings	Choose *Settings*	The system displays the following information: • Extended Configuration Management (XCM) Application Configuration • Knowledge Base Description • Knowledge Base Version • Knowledge Base Profile • Knowledge Base Build Number Here, you can also enable/disable the following options: • Display Invisible Characteristics • Display Language-Dependent Descriptions • Show Characteristic Groups • Display All Options • Show Status Lights • Evaluate Characteristics Online • Render Values in Multiple Columns • Enable JQuery Controls • Show Assignable Values only • Indent Components • Select Price Type • Enable Customization List on Main Screen • Show position number of each Sub-Component • Show Component Quantities

To access individual functions shown in the table, use the layer for the relevant component:

Function	Navigation	More Information
Manually add a component to the configuration	Choose *Add Component*	The system displays the list of components that can be added to the configuration manually, with the following information: • Minimum quantity (*How Many Must I Have?*) • Maximum quantity (*How Many Can I Have?*) • Current quantity (*How Many Do I Have?*) This information is maintained in the super BOM. Choose the *Add Component* pushbutton next to the component you want to add. When you have completed your entries, choose *Return to Configuration* to continue.
Delete a component	Choose *Delete Component*	You cannot delete a component that is related to another component by a constraint.
Specialize a component	Choose *Specialize*	The system displays a list of the possible products. The list is maintained as part of the knowledge base for the solution.
Unspecialize a component	Choose *Unspecialize*	You use this function to reverse a specialization.

3.5 Restoring Solution Configuration

The Configuration Engine processes data used during configuration of products. It also supports configuration processes and the final configuration can be save and restored later, for editing.

The restore process creates/modifies facts, fires dependencies, and then deletes the facts used while configuring the products. For more information about this process, refer to SAP Note 2365244 (Configuration restore based on the user inputs).

4 Integration with Vehicle Management System (VMS)

Use

SAP Solution Sales Configurator is an S/4HANA add-on used to process complex system and solution configurations. It can process the LO-VC compatible product configurations with the same functionality as the SAP Internet Pricing Configurator (IPC).

> i Note
>
> The **SAP Internet Pricing Configurator (SAP IPC)** has been discontinued in SAP S/4HANA according to SAP Note 2242322 (IPC does not exist in SAP S/4HANA).
>
> You can use the SAP Solution Sales Configurator as a compatible alternative to the IPC.

Prerequisites

- You have installed the SAP Solution Sales Configurator.
- You have connected the **Vehicle Management System (VMS)** to SAP Solution Sales Configuration for SAP S/4HANA by defining an RFC destination to the SAP Solution Sales Configuration for SAP S/4HANA engine.
- You have connected VMS to SAP Solution Sales Configuration for SAP S/4HANA by defining an https destination to the SAP Solution Sales Configuration for SAP S/4HANA user interface.
- You have defined an Extended Configuration Management (XCM) configuration and maintained it in table `TCUUISCEN`.
- You have configured the vehicle material to be used with SAP Solution Sales Configuration for SAP S/4HANA in the transaction `VELO`.

For more information, see the SAP SSC Administration Guide

Features

- You can change the calling parameters for SAP Solution Sales Configuration for SAP S/4HANA by using the BAdI `VLC_SCE_PARAMETERS`.
- SAP Solution Sales Configuration for SAP S/4HANA is called for every configuration transaction in VMS, when you click on the icon *Configure when you execute an action*.
- You can implement the user parameter `VELO_SCE_FULLSCREEN` to define that SAP IPC is called in full-screen mode and not integrated into the *Action* tab page.

- You can use the new SAP Solution Sales Configuration for SAP S/4HANA UI Composer user interface by setting the user parameter `VELO_CU50_ACTIVE` to the value 'X'. When using this parameter, as with `CU50` and `LO-VC`, you will not be able to provide configuration and pricing context information to SAP Solution Sales Configuration for SAP S/4HANA. BAdIs such as `ERP_CFG_ADAPT` and `VLC_SCE_PARAMETERS` will not be called in that case.

5 Compressed Storage of XML Configuration Results

Use

Technical Data

Technical Name of Business Function	/SLCE/BF_XML_COMPRSSION
Type of Business Function	Enterprise Business Function
Available From	SAP Solution Sales Configuration for SAP S/4HANA 1907
Application Component	*FBS Solution Configuration* (LO-SLC)
Required Business Function	Not relevant

You can use this business function to store XML configuration results in the database in compressed form. If the business function is deactivated, the configuration results are stored in uncompressed form.

Integration

The business function can be activated separately in SAP S/4HANA. You can activate the business function in one system but leave it deactivated in the other system.

Prerequisites

- You have installed the following components as of the version mentioned:

Type of Component	Component	Required for the Following Features Only
Software Component	SLCE (for SAP S/4HANA)	

- If sales documents with configurable products already exist before you activate the business function, you must migrate the data manually from table /SLCE/CFG_XML to /SLCE/XML_BLOB (for SAP S/4HANA). Unless you do so, the system cannot restore the configurations for existing sales document items.
  The data migration is necessary because XML configuration results for sales document items are no longer stored in table /SLCE/CFG_XML after you have activated the business function. Instead, they are stored in table /SLCE/XML_BLOB. Configuration results are read either from table *CFG_XML or *XML_BLOB (for example, when the configuration screen of an existing item is opened).

Features

Compression of XML Configuration Results

Configuration results for sales document items are stored as XML in the back-end system. In releases of SAP Solution Sales Configuration earlier than Support Package 3, the configuration results are stored in an uncompressed form. This is also the case in Support Package 3 if this business function is deactivated. If you activate this business function, the configuration results are stored in a compressed form.

If you do not activate the business function, configuration results are stored in table `/SLCE/CFG_XML` for SAP S/4HANA. If you activate the business function, the configuration results are stored in table `/SLCE/XML_BLOB` for SAP S/4HANA.

The business function is reversible and can be deactivated again if it has been activated. Note that some manual migration activities are required for existing configuration results.

6 Operations Information

Certain administrative activities are required to use SAP Solution Sales Configuration. For more information about the use and administration of the application, see the following sections:

Related Information

User Exit Deployment [page 178]

6.1 User Exit Deployment

Use

SAP Solution Sales Configuration offers an alternative way of providing user-defined functions that are to be consumed by the SSC engine running on SAP Application Server Java. User-defined functions are declarative functions and pfunctions in the configuration engine and pricing formulas in the pricing.

The following sections discuss using Apache Maven from the command line to deploy a user exit .jar file without existence of a Maven project (`pom.xml`).

Deployment using the Command Line

When user exits are deployed with Maven from the command line, multiple parameters need to be specified.

The examples discussed in the succeeding sections presume the existence of the following parameters:

```
SET PATH_TO_FILE=user_exit_jar_file.jar
SET UNIT_NAME="My First User Exit Unit"
SET HOST=hostname_of_as_java
SET PROVIDER_URL=%HOST%:50004
SET USERNAME=user
SET PASSWORD=secret
SET SIMULATE=false
```

Deploying a File

Code for Deploying a File

You can deploy a file using the following code:

```
call mvn ^
-Duserexit.jarFile=%PATH_TO_FILE% ^
-Duserexit.userExitUnitName=%UNIT_NAME% ^
-Duserexit.providerUrl=%PROVIDER_URL% ^
-Duserexit.username=%USERNAME% ^
-Duserexit.password=%PASSWORD% ^
-Duserexit.simulate=%SIMULATE% ^
com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:deploy-file
```

Output

Below you can see the possible output from the code mentioned above:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------------------------------
[INFO] Building Maven Stub Project <No POM> 1
[INFO] -------------------------------------------
[INFO]
[INFO] –- ssc-user-exit-maven-plugin:2.0.0:deploy-file <default-cli> @
standalone-pom –-
[INFO] Deploying user-exit file to server...
[INFO] File successfully deployed to server in 2590 ms.
[INFO] -------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------
[INFO] Total time: 3.54s
[INFO] Finished at: Tue Mar 03 13:44:21 CET 2015
[INFO] Final Memory: 8M/241M
[INFO] -------------------------------------------
```

Undeploying/Removing a File

Code for Undeploying a File

You can remove a file using the following code:

```
mvn ^
-Duserexit.userExitUnitName=%UNIT_NAME% ^
-Duserexit.providerUrl=%PROVIDER_URL% ^
-Duserexit.username=%USERNAME% ^
-Duserexit.password=%PASSWORD% ^
-Duserexit.simulate=%SIMULATE% ^
com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:undeploy-file
```

Output

Below you can see the possible output from the code mentioned above:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------------------------------
```

```
[INFO] Building Maven Stub Project <No POM> 1
[INFO] -------------------------------------------
[INFO]
[INFO] –- ssc-user-exit-maven-plugin:2.0.0:undeploy-file <default-cli> @
standalone-pom –-
[INFO] Deleting user-exit unit from server...
[INFO] Unit successfully deleted from server in 2542 ms.
[INFO] -------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------
[INFO] Total time: 3.260s
[INFO] Finished at: Tue Mar 03 13:49:41 CET 2015
[INFO] Final Memory: 9M/304M
[INFO] -------------------------------------------
```

Retrieving Information about Deployed Files

Code for Retrieving Information about Deployed Files

You can retrieve information on a deployed files using the following code:

```
mvn ^
-Duserexit.providerUrl=%PROVIDER_URL% ^
-Duserexit.username=%USERNAME% ^
-Duserexit.password=%PASSWORD% ^
-Duserexit.simulate=%SIMULATE% ^
com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:info
```

Output

Below you can see the possible output from the code mentioned above:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------------------------------
[INFO] Building Maven Stub Project <No POM> 1
[INFO] -------------------------------------------
[INFO]
[INFO] –- ssc-user-exit-maven-plugin:2.0.0:info <default-cli> @ standalone-pom –-
[INFO] Retrieving user-exit unit information from server...
[INFO] User-exit unit information successfully retrieved from server in 2901 ms.
[INFO] -------------------------------------------
[INFO] Number of available units: 1
[INFO]   -  My First User Exit Unit <9 resources>
[INFO] -------------------------------------------
[INFO] -------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------
[INFO] Total time: 3.967s
[INFO] Finished at: Tue Mar 03 13:51:49 CET 2015
[INFO] Final Memory: 8M/241M
[INFO] -------------------------------------------
```

For information on deployment based on a Maven project, refer to **Deployment Using the Solution Modeling Environment**

Related Information

Deployment Using the Solution Modeling Environment [page 181]

6.1.1 Deployment Using the Solution Modeling Environment

Context

The User-Exit-Maven-Plugin enables you to automatically deploy self-developed pfunctions with Maven.

Here, you can follow an alternative process for user exit deployment, using an existing Maven project. This process also allows deployment from within the Solution Modeling Environment.

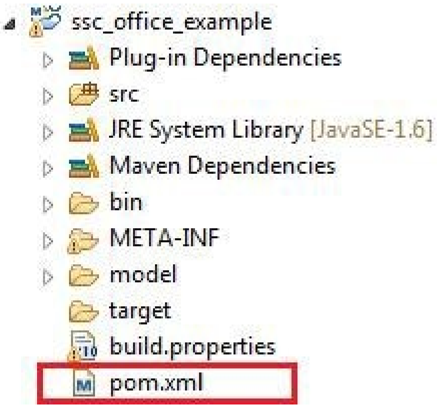
> i Note
>
> When the Solution Modeling Environment plugin is installed in an eclipse environment, there will be two windows command/batch files that are created in the installation directory of Eclipse that shall be used later in the procedure below:

Procedure

1. Create an example project from *File* > *Other*.
2. Select *SAP CPQ for Solution Sales Configuration* > *Examples* > *Office Model Example (with pfunctions)*.

   Click *Next* to finish.
3. A new project *ssc_office_example* will be created, whose root directory contains the file `pom.xml`.

   This file contains a sample Maven project configuration that you can use as a starting point for your projects.

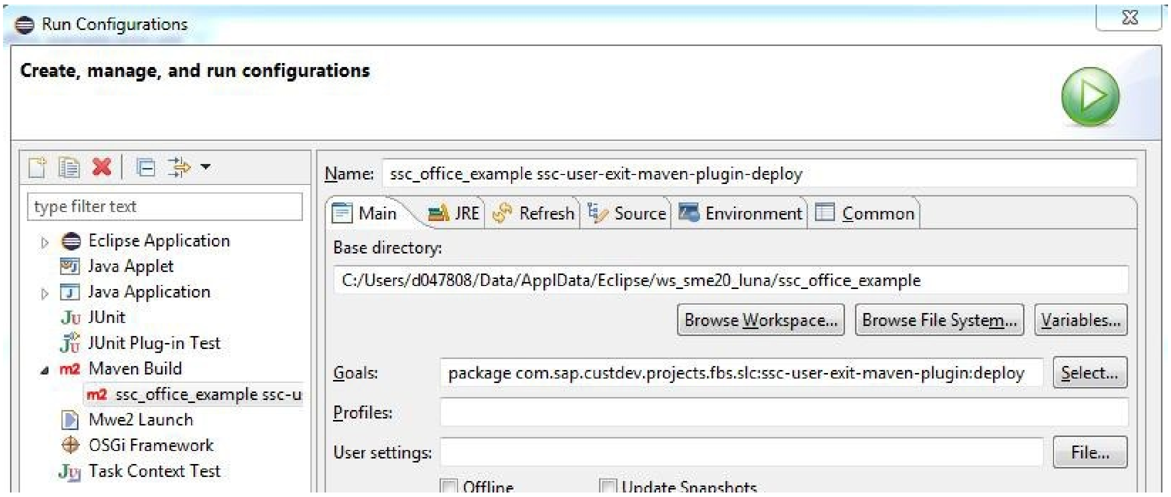


ssc_office_example
- Plug-in Dependencies
- src
- JRE System Library [JavaSE-1.6]
- Maven Dependencies
- bin
- META-INF
- model
- target
- build.properties
- pom.xml

Maven Project Configuration

4. In order to start a simulation of the deployment, switch to the *Java Perspective* of Eclipse and choose ▶ *Run* › *Run Configurations* ◀ from the menu.
5. Create a new entry below the Maven *Build.*
6. Enter the following text in the *Goals* input field:

```
package com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:deploy
```



Run Configurations

Create, manage, and run configurations

Name: ssc_office_example ssc-user-exit-maven-plugin-deploy

Base directory: C:/Users/d047808/Data/ApplData/Eclipse/ws_sme20_luna/ssc_office_example

Goals: package com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:deploy

Profiles:

User settings:

□ Offline □ Update Snapshots

Input Field for Goals

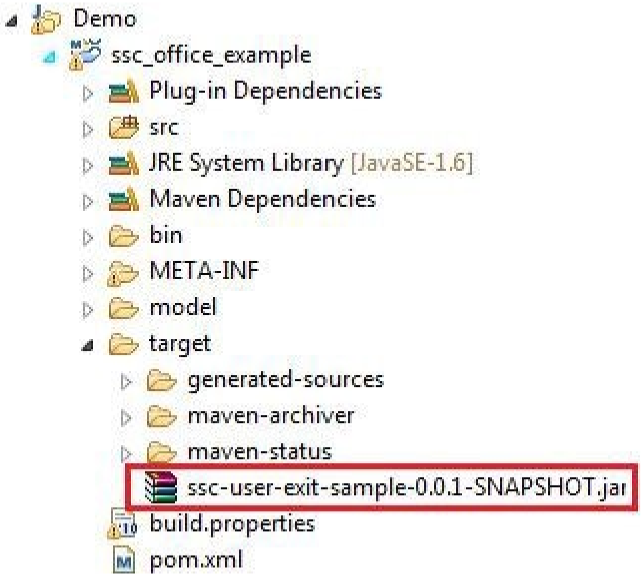
7. Click *Run* to start the user exit deployment.

   Two Maven goals will be executed during this step. The goal *package* compiles Java source files, for example, pfunction class files that are in the *src* folder of the project and generates a jar file in the project's *target* folder
8. The goal deploy (of the *ssc-user-exit-maven-plugin* plug-in) performs the deployment of the generated jar file to SAP AS Java. The connection details of SAP AS Java are in the `pom.xml` file.
9. In the `pom.xml` file, enter the user name, password, and the provider URL of the server in which the plugin needs to be deployed

> **i Note**
>
> If the parameters discussed above are provided, you can delete the *artifactId* input.

10. By default, the simulation mode is activated (see the `pom.xml` file) and can be used to test the proper installation of Maven and the user exit deployment plug-in, without having to set up SAP AS Java
11. Now, run the windows command files present in the eclipse installed directory

    The `ssc-mvn-install.cmd` file installs the required artifacts in your local Maven repository and deploys all necessary artifacts in a repository of your choice
12. If everything works, you will see an output similar to the following in the *Console* view of Eclipse:



- Demo
  - ssc_office_example
    - Plug-in Dependencies
    - src
    - JRE System Library [JavaSE-1.6]
    - Maven Dependencies
    - bin
    - META-INF
    - model
    - target
      - generated-sources
      - maven-archiver
      - maven-status
      - ssc-user-exit-sample-0.0.1-SNAPSHOT.jar
    - build.properties
    - pom.xml

Output Jar File

```
[INFO] Scanning for projects...
...
[INFO]
------------------------------------------------------------------------
[INFO] Building ssc-user-exit-sample 0.0.1-SNAPSHOT
[INFO]
------------------------------------------------------------------------
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ ssc-user-
exit-sample --- [INFO] Changes detected - recompiling the module!
...
[INFO]
[INFO] --- ssc-user-exit-maven-plugin:2.0.0:deploy (default-cli) @ ssc-user-
exit-sample --- [INFO] Deploying user-exit file to server...
[INFO] Deployment to server skipped (simulation mode).
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
...
```

6.1.2 SSC-USER-EXIT-MAVEN-PLUGIN Reference

Goals Available for this Plug-In

Goal	Description
`ssc-user-exit:deploy`	Deploys an SSC user-exit unit to SAP Application Server Java. An SSC user-exit unit can be one of the following: • a JAR archive that contains declarative functions and pfunctions to be used by the SAP Solution Sales Configuration for SAP S/4HANA configuration engine, or • a JAR archive that contains pricing formulas to be used by the SAP Solution Sales Configuration for SAP S/4HANA pricing engine
`ssc-user-exit:deploy-file`	Deploys an SSC user-exit unit to SAP Application Server Java.
`ssc-user-exit:info`	Retrieves information about all SSC user-exit units that are installed on SAP Application Server Java.
`ssc-user-exit:undeploy`	Undeploys/deletes an SSC user-exit unit from SAP Application Server Java.
`ssc-user-exit:undeploy-file`	Undeploys/deletes an SSC user-exit unit from SAP Application Server Java.

System Requirements

Maven	3.1
JDK	1.7

Goal Details

For further information on the goals discussed above, refer to the topics below:

- SSC-USER-EXIT:DEPLOY

- SSC-USER-EXIT:DEPLOY-FILE
- SSC-USER-EXIT:INFO
- SSC-USER-EXIT:UNDEPLOY
- SSC-USER-EXIT:UNDEPLOY-FILE

Related Information

SSC-USER-EXIT:DEPLOY-FILE [page 185]
SSC-USER-EXIT:DEPLOY [page 189]
SSC-USER-EXIT:INFO [page 193]
SSC-USER-EXIT:UNDEPLOY-FILE [page 197]
SSC-USER-EXIT:UNDEPLOY [page 201]

6.1.2.1 SSC-USER-EXIT:DEPLOY-FILE

Full Name

```
com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:deploy-file
```

Description

This goal deploys an SSC user-exit unit to SAP Application Server Java.

Prerequisites

- Maven should run in online mode.

Required Parameters

Name	Type	Description

`initialContextFactory`	`String`	The initial context factory.
		See also:
		`javax.naming.Context.INITIAL_CONTEXT_FACTORY`
		Default value:
		`com.sap.engine.services.jndi.InitialContextFactoryImpl`
		User property:
		`userexit.initialContextFactory`

`jndiLookupString`	`String`	The JNDI lookup string.
		See also:
		`javax.naming.InitialContext.lookup(String)`
		Default value:
		`ejb:/appName=sap.com/cdev~fbs_slc_java, jarName=sap.com~cdev~fbs_slc_exit_mgr_ejb.jar, beanName=UserExitBean, interfaceName=com.sap.custdev.projects.fbs.slc.exit.api.UserExitBeanRemote`
		User property:
		`userexit.jndiLookupString`

Name	Type	Description

`provideUrl`	`String`	The service provider URL.
		See also:
		`javax.naming.Context.PROVIDER_URL`
		Default value:
		`localhost:50004`
		User property:
		`userexit.providerUrl`

`urlPkgPrefixes`	`String`	The URL package prefixes.
		See also:
		`javax.naming.Context.URL_PKG_PREFIXES`
		Default value:
		`com.sap.engine.services`
		User property:
		`userexit.urlPkgPrefixes`

Optional Parameters

Name	Type	Description

`jarFile`	`File`	The path to a jar file that should be deployed.
		Default value:
		`${project.build.directory}/${project.build.finalName}.jar`
		User property:
		`userexit.jarFile`

Name	Type	Description

`password`	`String`	Password used to authenticate in SAP Application Server Java. If not specified, it defaults to no password (empty string).
		See also:
		`javax.naming.Context.SECURITY_CREDENTIALS`
		User property:
		`userexit.password`

`serverId`	`String`	The server ID in `settings.xml` is used when connecting to SAP Application Server Java. If specified, it overrides the values specified in the username and password properties. **User property:** `userexit.serverId`
`settings`	`Settings`	The Maven settings. **User property:** `settings`

`simulate`	`Boolean`	The flag that decides whether the deployment should be simulated or not. If set to true, no server communication takes place.
		Default value:
		`false`
		User property:
		`userexit.simulate`

Name	Type	Description

`userExitUnitName`	`String`	The user-exit unit name.
		Default value:
		`${project.artifact.artifactId}`
		User property:
		`userexit.userExitUnitName`

`username`	`String`	Username used to authenticate with SAP Application Server Java.
		See also:
		`javax.naming.Context.SECURITY_PRINCIPAL`
		Default value:
		`Administrator`
		User property:
		`userexit.username`

6.1.2.2 SSC-USER-EXIT:DEPLOY

Full Name

```
com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:deploy
```

Description

This goal deploys an SSC user-exit unit to SAP Application Server Java. An SSC user-exit unit can be any one of the following:

- a JAR archive that contains declarative functions and pfunctions to be used by the SSC configuration engine, or

- a JAR archive that contains pricing formulas to be used by the SSC pricing engine

Prerequisites

- A Maven project should be executed
- Maven should run in online mode

Required Parameters

Name	Type	Description

`initialContextFactory`	`String`	The initial context factory.
		See also:
		`javax.naming.Context.INITIAL_CONTEXT_FACTORY`
		Default value:
		`com.sap.engine.services.jndi.InitialContextFactoryImpl`
		User property:
		`userexit.initialContextFactory`

`jarFile`	`File`	The path to a jar file that should be deployed.
		Default value:
		`${project.build.directory}/${project.build.finalName}.jar`
		User property:
		`userexit.jarFile`

Name	Type	Description

`jndiLookupString`	`String`	The JNDI lookup string.
		See also:
		`javax.naming.InitialContext.lookup(String)`
		Default value:
		`ejb:/appName=sap.com/cdev~fbs_slc_java, jarName=sap.com~cdev~fbs_slc_exit_mgr_ejb.jar, beanName=UserExitBean, interfaceName=com.sap.custdev.projects.fbs.slc.exit.api.UserExitBeanRemote`
		User property:
		`userexit.jndiLookupString`

`providerURL`	`String`	The service provider URL.
		See also:
		`javax.naming.Context.PROVIDER_URL`
		Default value:
		`localhost:50004`
		User property:
		`userexit.providerUrl`

`urlPkgPrefixes`	`String`	The URL package prefixes to use.
		See also:
		`javax.naming.Context.URL_PKG_PREFIXES`
		Default value:
		`com.sap.engine.services`
		User property:
		`userexit.urlPkgPrefixes`

`userExitUnitName`	`String`	The user-exit unit name. **Default value:** `${project.artifact.artifactId}` **User property:** userexit.userExitUnitName

Optional Parameters

Name	Type	Description

`password`	`String`	Password used for authentication in SAP Application Server Java. If not specified, it defaults to no password (empty string).
		See also:
		`javax.naming.Context.SECURITY_CREDENTIALS.`
		User property:
		`userexit.password`

`serverID`	`String`	The server ID in `settings.xml` to use when connecting to SAP Application Server JAVA. If specified, it overrides the values specified in the username and password properties. **User property:** `userexit.serverId`
`settings`	`Settings`	The Maven settings. **User property:** `settings`

`simulate`	`Boolean`	The flag that decides whether the deployment should be simulated or not. If set to true, no server communication takes place.
		Default value:
		`false`
		User property:
		`userexit.simulate`

Name	Type	Description

`username`	`String`	Username used to authentication in SAP Application Server Java.
		See also:
		`javax.naming.Context.SECURITY_PRINCIPAL`
		Default value:
		`Administrator`
		User property:
		`userexit.username`

6.1.2.3 SSC-USER-EXIT:INFO

Full Name

```
com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:info
```

Description

This goal retrieves information about all SSC user-exit units that are installed on SAP Application Server Java.

Prerequisites

- The Maven should run in online mode

Required Parameters

Name	Type	Description

`initialContextFactory`	`String`	The initial context factory.
		See also:
		`javax.naming.Context.INITIAL_CONTEXT_FACTORY.`
		Default value:
		`com.sap.engine.services.jndi.InitialContextFactoryImpl`
		User property:
		`userexit.initialContextFactory`

`jndiLookupString`	`String`	The JNDI lookup string.
		See also:
		`javax.naming.InitialContext.lookup(String)`
		Default value:
		`ejb:/appName=sap.com/cdev~fbs_slc_java, jarName=sap.com~cdev~fbs_slc_exit_mgr_ejb.jar, beanName=UserExitBean, interfaceName=com.sap.custdev.projects.fbs.slc.exit.api.UserExitBeanRemote`
		User property:
		`userexit.jndiLookupString`

Name	Type	Description

`providerUrl`	`String`	The service provider URL.
		See also:
		`javax.naming.Context.PROVIDER_URL`
		Default value:
		`localhost:50004`
		User property:
		`userexit.providerUrl`

`urlPkgPrefixes`	`String`	The URL package prefixes to use.
		See also:
		`javax.naming.Context.URL_PKG_PREFIXES`
		Default value:
		`com.sap.engine.services`
		User property:
		`userexit.urlPkgPrefixes`

Optional Parameters

Name	Type	Description

`password`	`String`	Password used for authentication in SAP Application Server Java. If not specified, it defaults to no password (empty string).
		See also:
		`javax.naming.Context.SECURITY_CREDENTIALS`
		User property:
		`userexit.password`

Name	Type	Description
`serverId`	`String`	The server ID in settings.xml is used when connecting to SAP Application Server JAVA. If specified, it overrides the values specified in the username and password properties. **User property:** `userexit.serverId`
`Settings`	`Settings`	The Maven settings. **User property:** `settings`

`simulate`	`Boolean`	This flag decides whether the deployment should be simulated or not. If set to true, no server communication takes place.
		Default value:
		`false`
		User property:
		`userexit.simulate`

`username`	`String`	Username used for authentication in SAP Application Server Java.
		See also:
		`javax.naming.Context.SECURITY_PRINCIPAL`
		Default value:
		`Administrator`
		User property:
		`userexit.username`

6.1.2.4 SSC-USER-EXIT:UNDEPLOY-FILE

Full Name

```
com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:undeploy-file
```

Description

This goal removes (undeploys) an SSC user-exit unit from SAP Application Server Java.

Prerequisites

- The Maven runs in online mode

Required Parameters

Name	Type	Description

`initialContextFactory`	`String`	The initial context factory.
		See also:
		`javax.naming.Context.INITIAL_CONTEXT_FACTORY`
		Default value:
		`com.sap.engine.services.jndi.InitialContextFactoryImpl`
		User property:
		`userexit.initialContextFactory`

Name	Type	Description

`jndiLookupString`	`String`	The JNDI lookup string.
		See also:
		`javax.naming.InitialContext.lookup(String)`
		Default value:
		`ejb:/appName=sap.com/cdev~fbs_slc_java, jarName=sap.com~cdev~fbs_slc_exit_mgr_ejb.jar, beanName=UserExitBean, interfaceName=com.sap.custdev.projects.fbs.slc.exit.api.UserExitBeanRemote`
		User property:
		`userexit.jndiLookupString`

`provideUrl`	`String`	The service provider URL.
		See also:
		`javax.naming.Context.PROVIDER_URL`
		Default value:
		`localhost:50004`
		User property:
		`userexit.providerUrl`

`urlPkgPrefixes`	`String`	The URL package prefixes to use.
		See also:
		`javax.naming.Context.URL_PKG_PREFIXES`
		Default value:
		`com.sap.engine.services`
		User property:
		`userexit.urlPkgPrefixes`

Optional Parameters

Name	Type	Description

`jarFile`	`File`	The path to a jar file that should be deployed.
		Default value:
		`${project.build.directory}/${project.build.finalName}.jar`
		User property:
		`userexit.jarFile`

`password`	`String`	Password used to authenticate with SAP Application Server Java. If not specified, it defaults to no password (empty string).
		See also:
		`javax.naming.Context.SECURITY_CREDENTIALS`
		User property:
		`userexit.password`

`serverId`	`String`	The server ID in `settings.xml` is used when connecting to SAP Application Server Java. If specified, it overrides the values specified in the username and password properties. **User property:** `userexit.serverId`
`settings`	`Settings`	The Maven settings. **User property:** `settings`

Name	Type	Description

`simulate`	`Boolean`	This flag decides whether the deployment should be simulated or not. If set to true, no server communication takes place.
		Default value:
		`false`
		User property:
		`userexit.simulate`

`userExitUnitName`	`String`	The user exit name.
		Default value:
		`${project.artifact.artifactId}`
		User property:
		`userexit.userExitUnitName`

`username`	`String`	The username used to authenticate with SAP Application Server Java.
		See also:
		`javax.naming.Context.SECURITY_PRINCIPAL`
		Default value:
		`Administrator`
		User property:
		`userexit.username`

6.1.2.5 SSC-USER-EXIT:UNDEPLOY

Full Name

```
com.sap.custdev.projects.fbs.slc:ssc-user-exit-maven-plugin:undeploy
```

Prerequisites

- The Maven project should be executed
- The Maven runs in online mode

Required Parameters

Name	Type	Description

`initialContextFactory`	`String`	The initial context factory.
		See also:
		`javax.naming.Context.INITIAL_CONTEXT_FACTORY.`
		Default value:
		`com.sap.engine.services.jndi.InitialContextFactoryImpl`
		User property:
		`userexit.initialContextFactory`

Name	Type	Description

`jarFile`	`File`	The path to a jar file that should be deployed.
		Default value:
		`${project.build.directory}/${project.build.finalName}.jar`
		User property:
		`userexit.jarFile`

`jndiLookupString`	`String`	The JNDI lookup string.
		See also:
		`javax.naming.InitialContext.lookup(String)`
		Default value:
		`ejb:/appName=sap.com/cdev~fbs_slc_java, jarName=sap.com~cdev~fbs_slc_exit_mgr_ejb.jar, beanName=UserExitBean, interfaceName=com.sap.custdev.projects.fbs.slc.exit.api.UserExitBeanRemote`
		User property:
		`userexit.jndiLookupString`

`providerUrl`	`String`	The service provider URL.
		See also:
		`javax.naming.Context.PROVIDER_URL`
		Default value:
		`localhost:50004`
		User property:
		`userexit.providerUrl`

Name	Type	Description

`urlPkgPrefixes`	`String`	The URL package prefixes to use.
		See also:
		`javax.naming.Context.URL_PKG_PREFIXES`
		Default value:
		`com.sap.engine.services`
		User property:
		`userexit.urlPkgPrefixes`

`userExitUnitName`	`String`	The user-exit unit name.
		Default value:
		`${project.artifact.artifactId}`
		User property:
		`userexit.userExitUnitName`

Optional Parameters

Name	Type	Description

`password`	`String`	Password used for authentication in SAP Application Server Java. If not specified, it defaults to no password (empty string).
		See also:
		`javax.naming.Context.SECURITY_CREDENTIALS`
		User property:
		`userexit.password`

Name	Type	Description
`serverId`	`String`	The server ID in `settings.xml` is used when connecting to SAP Application Server Java. If specified, it overrides the values specified in the username and password properties. **User property:** `userexit.serverId`
`settings`	`Settings`	The Maven settings. **User property:** `settings`

`simulate`	`Boolean`	This flag decides whether the deployment should be simulated or not. If set to true, no server communication takes place.
		Default value:
		`false`
		User property:
		`userexit.simulate`

`username`	`String`	Username used for authentication with SAP Application Server Java.
		See also:
		`javax.naming.context.SECURITY_PRINCIPAL`
		Default value:
		`Administrator`
		User property:
		`userexit.username`

7 UI Composer

Naming Conventions

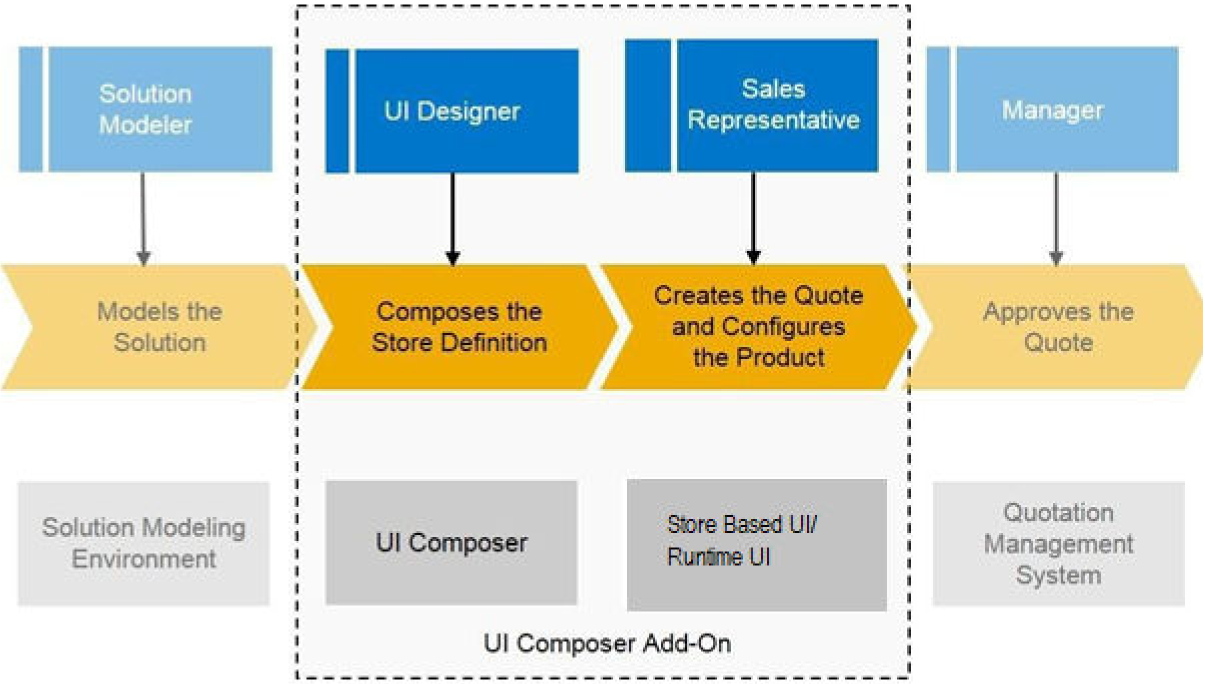
Throughout this document, the **SAP Solution Sales Configuration, UI Composer** is simply referred to as **UIC**. This shorter name is used purely to increase readability; both names are entirely synonymous.

Purpose and Scope

The following sections aim to provide all the information required for UI designers and administrators to use the features of the UI Composer to produce harmonized user interfaces across product families. After designing and activating the screens in the UI Composer, the sales representatives can use the newly designed interface. This document assumes that you have successfully performed the required configuration activities and are ready to use the tool productively.

For information about the configuration, kindly refer to the **Configuring SAP Solution Sales Configuration, UI Composer** section in the SAP SSC: Administration Guide.

Process Overview



Solution Modeler
UI Designer
Sales Representative
Manager
Models the Solution
Composes the Store Definition
Creates the Quote and Configures the Product
Approves the Quote
Solution Modeling Environment
UI Composer
Store Based UI/ Runtime UI
Quotation Management System
UI Composer Add-On

In the UI Composer, UI designers use knowledge bases (classical and advanced models) from the solution modeling environment of SAP Solution Sales Configuration as the basis for creating product configuration screens, known as store definitions.

To compose the store definitions, UI designers select and configure UI controls for the elements of a solution model and use page and store templates to ensure a consistent layout of the store definitions for a particular product family or product line. The designers can view and edit the layout of the store pages by simply dragging and dropping different UI elements on the screen, thus, requiring no programming skills to compose the store definitions. The UI Composer also provides an interactive preview so that the designers can test the behavior of the store pages and UI controls. If needed, you can easily extend the UI Composer to include custom store templates and custom UI controls.

When the user launches the configuration of a product, it is launched in either of the following UIs (provided product is set for SAP Solution Sales Configuration):

- **JSP UI**: If the product is set to be launched with the JSP UI in transaction `/SLCE/CFG_MAT`.
- **Store Based UI**: If the product is set to be launched with UIC in transaction `/SLCE/CFG_MAT`, and there is a published store for the knowledge base of the product.
- **Dynamic Runtime UI**: If the product is set to be launched with UIC but there is no publish store for the knowledge base of the product.

7.1 Glossary

Term	Definition
Knowledge base	A structured database that contains the master data of a solution model created in the solution modeling environment of SAP Solution Sales Configuration. A knowledge base contains information about the classes and materials used in a solution.
Master store definition	A collection of predefined pages that UI designers can use in their regular store definitions for a specific product category. Master store definitions are created by UI administrator users.
Predefined page	A predefined page layout used for content that is identical for all stores of a specific product category, such as maintenance configuration information, contact configuration information, etc. Predefined pages can be reused in all regular store definitions.

Term	Definition
Solution	A collection of interrelated components, such as hardware components, software components, and services. In addition to selling their own products, many businesses also sell solutions that include products, services, and parts from other manufacturers. For example, many technology companies sell solutions that include combinations of highly complex hardware, software, and services, and each of these can have options or features that the customer must specify during the ordering process.
Solution model	A hierarchical decomposition of a solution. It defines the products (configurable materials and the services) that can be contained within the solution. It also defines the relationships between the various elements of the solution, including any dependencies (constraints and rules).
Store	A set of pages rendered by the Product Configurator application. A store displays the selectable options and features of a solution. To prepare a quotation, sales representatives use the store to configure a solution.
Store definition	A configuration of a store used in the Product Configurator application to specify options and features of a solution. Users define the store definition in the UI Composer. To do so, they use store templates and predefined pages to create a uniform look for the stores of a product category. A store definition is based on a specific version of a knowledge base.
Store template	A store template contains the basic UI layout of the store pages. For example, a store template can specify the layout of the header and footer for all pages. The store template also specifies the UI elements that can be used in the store definitions. A default store template is delivered with the UI Composer. However, you can develop your own store templates. For example, you might develop a different store template for each product line.
Workspace	A screen in the UI Composer where users organize and manage their store definition versions.

7.2 UI Composer Design-time User Interface

UI Composer Design-time UI focuses on store designing which can be launched for product configuration later.

7.2.1 Workspace Screen

The workspace screen displays your workspaces and a list of the store definitions referenced in a workspace.

You can use several workspaces to organize your store definitions.

> i Note
>
> You can only access your own workspaces and cannot share workspaces with other users.
>
> The workspace screen appears once you start the application.

Features

- In the *Workspaces* screen area, you can edit and delete your workspaces, as well as create new ones. You can choose a workspace to display the store definitions in that workspace.
- In the *Store Definitions* screen area, you can see the store definitions in the selected workspace and sort or filter the list for specific store definitions. Regular store definitions in your workspace can have any of the following statuses:
  - Draft
  - Committed
  - Published

  In the *Store Definitions* list, you can also choose a store definition to edit the store pages on the *Composer* screen.

  > i Note
  >
  > While you are editing store pages, the store definition is locked temporarily and cannot be edited by other users.

  You can choose (Menu) to navigate to different screens where you can perform following tasks:
  - Create new store definitions
  - Create a new store definition based on an existing store definition
  - Add a reference for a store definition to the workspace

7.2.2 Create Store Screen

You can create new store definitions on the *Create Store* screen. To access the *Create Store* screen, choose ▶ *Menu* › *Create* ◀ from the workspace.

Features

The *Create Store* screen contains the following tabs:

- *Knowledge Base*
  Select the relevant knowledge base, its version and profile (if knowledgebase has multiple profiles).
- *Store Template*
  Select a store template.
- *Pages*
  Select redefined pages.
- *Header Info*
  Enter a suitable description for the store definition.

7.2.3 Copy Store Screen

On the *Copy Store* screen, you can edit a copy of an existing store definition to create a new store definition. To access the *Copy Store* screen, choose ▶ *Menu* › *Copyfrom* ◀ from the workspace.

Features

The *Copy Store* screen contains following tabs:

- *Store Definition*
  Select the store definition you want to copy.
- *Knowledge Base*
  Select the knowledge base and its relevant version.
- *Header Info*
  Complete the description of the store definition.

7.2.4 Composer Screen

You can configure the layout of a store on the *Composer* screen.

You can access the *Composer* screen by performing one of the following actions:

- Creating a new store on the *Create Store* screen
- Creating a new store definition based on an existing one, in the *Copy Store* screen
- Choosing a store definition in a workspace or on the *Manage Store Definitions* screen

On the Composer screen you can choose Menu to perform following actions:

- Commit or publish the store definition

- Delete the draft version

- View and edit the name and description of the store definition in the Header Info

- View previous versions

- Roll back to previous versions

Features

The *Composer* screen contains following screen areas:

- *Knowledge Base Elements*
  A tree view of the classes and materials of the knowledge base.
- *Characteristics*
  Characteristics of class or material selected in the *Knowledge Base Elements* screen area.
- *Control Palette*
  The UI elements that you can use for the classes and materials of the knowledge base.
- *Canvas*
  The central view of the *Composer* screen. On the canvas, you arrange the UI elements and map them to classes or materials of the knowledge base.
- *Properties*
  You configure the attributes of the UI elements on the canvas.

On the *Composer* screen, you can choose *Menu* to perform following actions:

- Commit or publish the store definition
- Delete the draft version
- View and edit the name and description of the store definition in the *Header Info*
- View previous versions
- Roll back to previous versions

7.2.5 Interactive Preview

From the *Composer* screen, you can display an interactive preview of a store in an additional browser tab.

The interactive preview simulates the store pages that you defined in the store definition. You can preview the design of your store and interact with the UI elements to test the features of the store pages.

7.2.6 Manage Store Definitions Screen

On the *Manage Store Definitions* screen, you can find all store definitions in the system and perform multiple administrative tasks.

To display the *Manage Store Definitions* screen, you need to have the UI administrator role. You can access the *Manage Store* screen by choosing (Manage Stores) in the header of the workspace.

Features

On the *Manage Store Definitions* screen, you see a list of all store definitions in the system. You can find store definitions by sorting the list according to categories or by searching for the name, the underlying knowledge base, or other details of the store definitions.

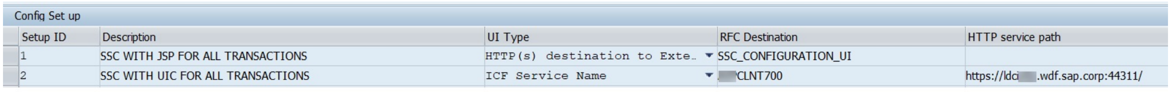
On the *Manage Store Definitions* screen, you can perform the following tasks:

- Choose the store definitions you want to edit in the composer
- Unlock store definitions that are locked by other users
- Delete store definitions

7.3 UI Composer Runtime User Interface

To launch the JSP UI or UIC Runtime UI for product configuration, the following setup is required:

1. Execute the transaction `/SLCE/V_CONFIG_SETUP`.
2. Create setup IDs for the JSP UI and UIC. The following screenshot shows an example of the setup IDs:



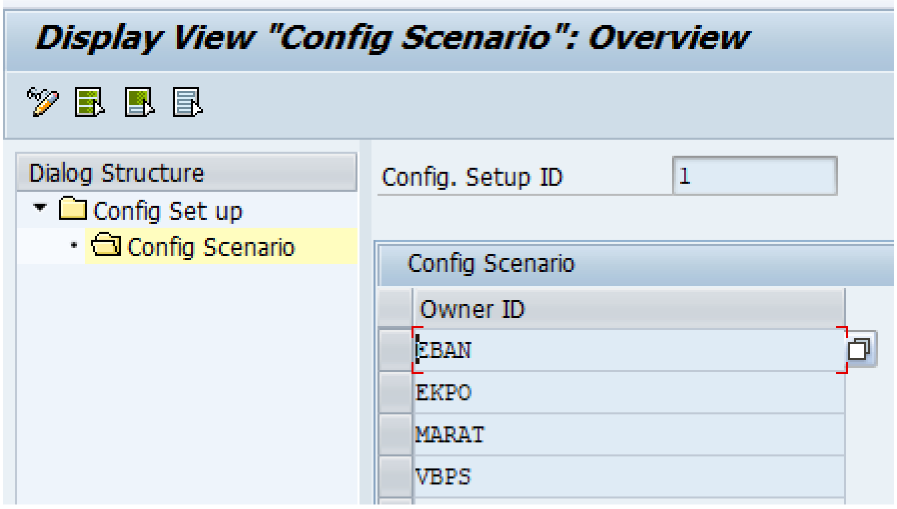
Config Set up

Setup ID	Description	UI Type	RFC Destination	HTTP service path
1	SSC WITH JSP FOR ALL TRANSACTIONS	HTTP(s) destination to Exte… ▾	SSC_CONFIGURATION_UI	
2	SSC WITH UIC FOR ALL TRANSACTIONS	ICF Service Name ▾	[redacted]CLNT700	https://ldci[redacted].wdf.sap.corp:44311/

> **i Note**
>
> You need to mention the SSC JAVA destination in the *RFC Destination* field for the JSP UI setup ID.

> For the UIC setup id, please maintain entries as follows:
>
> UI Type: ICF Service name
>
> RFC destination: ABAP connection to gateway system
>
> HTTP service path: Initial URL to launch UIC service in gateway system

3. You must provide the configuration scenarios where each of these UIs should appear. To do this, select the setup ID and double-click *Config Scenario* in the *Dialog Structure*. This provides you control over configuration UI. For example, if you do not want SAP Solution Sales Configuration UI to come up in purchase requisition, you may choose not to provide *EBAN* as a configuration scenario.



Display View "Config Scenario": Overview

Dialog Structure
- Config Set up
  - Config Scenario

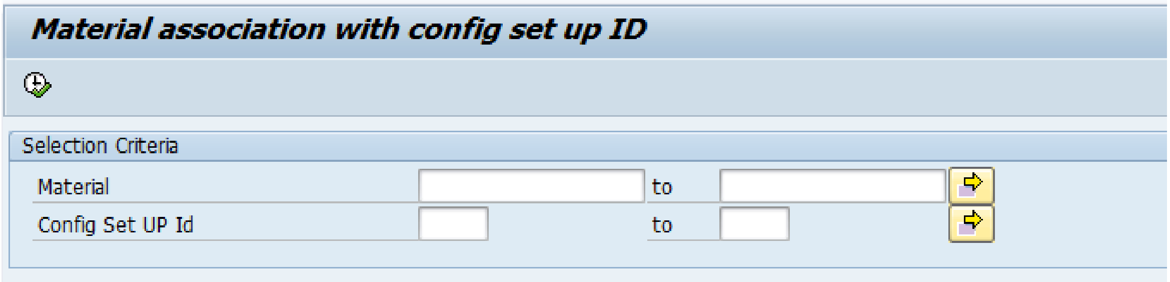
Config. Setup ID: 1

Config Scenario

Owner ID
EBAN
EKPO
MARAT
VBPS

You can provide the same scenarios for both JSP UI and UIC. You may find possible scenarios in table *T371F*.

4. Execute the transaction `/SLCE/CFG_MAT` to setup UI for a material.

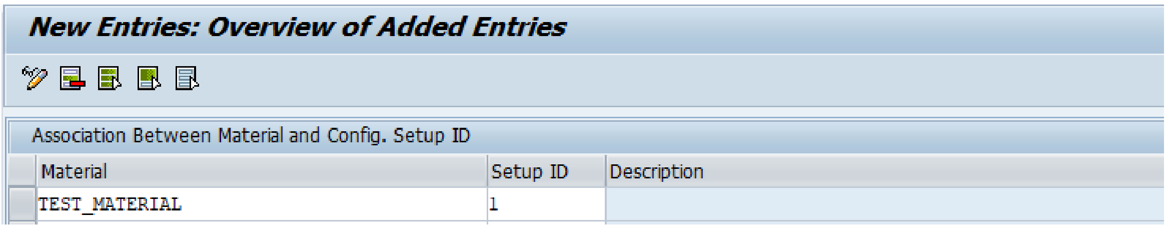


Material association with config set up ID

Selection Criteria

Material		to	
Config Set UP Id		to	

5. You can execute and add a new entry for material.



New Entries: Overview of Added Entries

Association Between Material and Config. Setup ID

Material	Setup ID	Description
TEST_MATERIAL	1	

In the previous image, since we have specified the *Setup ID* as 1 for JSP UI, whenever *TEST_MATERIAL* is configured, the JSP UI will be launched.
If the *Setup ID* is set as 2, then either the store-based UI (if a published store exists) or a dynamic runtime UI will be launched.

7.4 Roles and Authorizations in UI Composer

Prerequisites

Before you can use the UI Composer, you must meet the following prerequisites:

- The system is configured as per the SAP Solution Sales Configuration for SAP S/4HANA: Administration Guide.
- For recommended browser versions, refer to SAP Note 1716423 (SAPUI5 Browser Support)

Logon and Site Authorizations

To access the UI Composer application, use your browser to navigate to the following URL:

```
<https,http>://<host>:<port>/sap/bc/ui5_ui5/slcui/uic_web/index.html?sap-
client=<client>&Endpoint=ECC}
```

Authorizations

You need to have users in the following systems:

- SAP S/4HANA
- SAP NetWeaver Gateway Hub
- SAP NetWeaver Application Server for Java of SAP Solution Sales Configuration

In addition, you need one of the user roles for the UI Composer. For more information about this, refer to **User Roles for UI Composer**.

User Roles for UI Composer

UI Designer

The UI designer creates regular store definitions and configures them by composing store pages. The designer can also save the store definitions as a draft, commit store definitions, or create new draft versions, however, cannot cannot publish or delete store definitions that are created by other users.

UI Administrator

The UI administrator reviews the store definitions that the UI designers have committed and can publish these committed store definitions so that they are available in the Product Configurator application. The administrator also configures predefined pages by creating master store definitions.

The UI administrator can perform all the tasks of a UI designer but can also delete store definitions, unlock store definitions locked by other users, and revert store definitions to previous versions

UI Developer

The UI developer develops store templates and custom UI controls.

For more information on UI developer user tasks, kindly refer to the **Configuring SAP Solution Sales Configuration, UI Composer Add-on** section in the SAP Solution Sales Configuration for SAP S/4HANA: Administration Guide.

7.5 UI Designer Tasks

The following tasks can be performed by users with the UI designer role. However, the UI administrator role also contains authorizations for all these tasks, as well as for some additional administration tasks:

- Creating workspaces
- Adding store definitions to a workspace
- Creating new store definitions
- Copying store definitions
- Composing stores
- Committing store definitions
- Creating new versions of store definitions

Related Information

Creating Workspaces [page 215]
Adding Store Definitions to a Workspace [page 215]
Creating New Store Definitions [page 216]
Copying Store Definitions [page 217]
Composing Stores [page 218]
Committing Store Definitions [page 220]
Creating New Versions of Store Definitions [page 220]

7.5.1 Creating Workspaces

You can create several workspaces to organize your store definitions.

Procedure

1. Start the UI Composer application
2. In the *Workspaces* screen area, choose *Add*
3. Enter a name for the workspace and save it
4. You can also change the name of or remove existing workspaces

> i Note
>
> Workspaces are specific to each individual user. You cannot share workspaces with other users.

7.5.2 Adding Store Definitions to a Workspace

In a workspace, you can add references to store definitions.

Procedure

1. In the *Workspaces* screen area, select the relevant workspace
2. In the *Store Definitions* screen area, navigate to ▶ *Menu* › *Add Reference* ◀
3. Select the store definition you want to add

   You can find the store definition by sorting the list according to available categories or by searching for the name, the underlying knowledge base, or other details of the store definition. Only store definitions that were created for a knowledge base in the selected source system are displayed. If a store definition is already referenced by the selected workspace, this definition will not appear in the list.
4. Choose *Done*

> i Note
>
> Store definitions are not saved in workspaces; the workspace only contains references to store definitions. If you remove the reference to a store definition in your workspace, the store definition can still be used in another workspace by another user.

7.5.3 Creating New Store Definitions

Prerequisites

To create a new store definition, you need to fulfill the following prerequisites:

- A solution or product model has been created in the solution modeling environment of SAP Solution Sales Configuration and the knowledge base runtme version is deployed in the backend system (SAP S/4HANA).
- The configurable product has been created in the backend system (SAP S/4HANA).

Procedure

1. On the *Workspace* screen, select a relevant workspace
2. On the *Store Definitions* screen area, navigate to *Menu* > *Create*

   The *Create Store* screen opens.
3. On the *Knowledge Base* tab, select the knowledge base version of the solution for which you want to create the store definition

   > i Note
   >
   > The list contains only the knowledge base versions that are available in the selected source system and for which no store definitions have been created yet. This ensures that only one store definition exists for each knowledge base version.
4. On the *Store Template* tab, select the store template that you want to use for the store
5. On the *Pages* tab, select the predefined that you want to use in the store

   > i Note
   >
   > On the *Pages* tab, you can search and filter for all the predefined pages that are available in the system. You must ensure to select pages that are specific to the product category of the solution.

   A blank page is a page that is not predefined for specific content.
6. On the *Header Info* tab, enter a name and a description for the store definition. You can also specify tags for product categories.

   The information specified on the *Header Info* tab can be used to search and filter for the store definition, on various screens of the application.
7. Choose *Done*

   The store definition is added to the workspace and the *Composer* screen opens.

   > i Note
   >
   > For more information about designing and configuring the store pages, see **Composing Stores**.

Related Information

Composing Stores [page 218]

7.5.4 Copying Store Definitions

You can create new store definitions that use the same store templates, predefined pages, and UI controls as an existing store definition.

Context

When a new version of a knowledge base is available, you can copy a store definition that is based on a previous version of the knowledge base, to create a store definition for the new version. This way, you only need to configure UI controls for knowledge base elements that are different in the new version when you compose the store pages.

Procedure

1. On the *Workspace* screen, select a relevant workspace
2. In the *Store Definitions* screen area, navigate to ▶ *Menu* ▶ *Copy Store* ▶

   The *Copy Store* screen opens up.
3. on the *Store Definitions* tab, select the store definition you want to copy.

   You can find the store definition by sorting the list according to categories or by searching for the name, the underlying knowledge base, or other details of the store definition.
4. On the *Knowledge Base* tab, select the knowledge base of the solution

   > **i Note**
   >
   > There can be several versions of a knowledge base. You must ensure you select the correct version.
   >
   > The list only shows the knowledge base versions available in the selected source system.
5. On the *Header Info* tab, enter a name and a description for the store definition. You can also specify tags for product categories.

   The information specified on the *Header Info* tab can be used to search and filter for the store definition on various screens of the application.

   > **i Note**
   >
   > If there are discrepancies between the original store definition and the knowledge base that you have selected for your copy of the store definition, an error message appears. UI controls that are used in

> the original store definition and are incompatible with the knowledge base of your new store definition are automatically removed from the store design. You can download a summary of the incompatible knowledge base elements and the removed UI controls.

6. Choose *Done*

   The store definition is added to the workspace and the *Composer* screen opens up.

   > i Note
   >
   > For more information about designing and configuring the store pages, see **Composing Stores**.

Related Information

Composing Stores [page 218]

7.5.5 Composing Stores

Context

You can access the *Composer* screen by performing one of the following actions:

- Creating a new store in the *Create Store* screen
- Creating a new store definition based on an existing one in the *Copy Store* screen
- Choosing a store definition in a workspace

Procedure

1. Open the *Composer* screen.
2. Above the canvas, there is a tab for each store page. Select the store page that you want to configure.
3. In the *Knowledge Base Elements* area, select a class, material, or characteristic group that you want to configure on the selected store page. Drag and drop the material or class onto the canvas.

   In the *Characteristics* screen area, you see the characteristics of the class or material that you selected.

   > i Note
   >
   > Classes, materials, or characteristics that are already used in the store are indicated by a check.
4. In the *Knowledge Base* tab of the *Control Palette*, select a UI control or another UI element that you want to use for the class or material. Drag and drop the UI element onto the class or material on the canvas.

If the class or material has characteristics that need separate UI controls, these characteristics appear on the canvas. You can drag and drop the UI controls onto the characteristics. You can also copy UI elements on the canvas and paste them to other knowledge base elements or other areas of the store.

The UI elements you can select depend on the class, material, or characteristic that you want to configure for the store page.

> i Note
>
> - On the *General* tab of the *Control Palette*, you can find basic UI elements, such as headers, vertical sections, or horizontal sections. You can use these UI elements as static UI controls; they will not be bound to any knowledge base elements. Therefore, this category of UI controls is always visible at runtime.
> - On the *Custom* tab of the *Control Palette*, you can find custom UI Controls which were created by your UI5 developers as extensions of the solution. You can use them the same way as the controls on the *Knowledge Base* tab.
> - When binding the UI Controls to knowledge base elements (Characteristic), the tool validates the metadata of the characteristic to verify the use of the control to configure the characteristic. For example, you cannot use Checkbox to configure single-valued characteristic.
> - You can nest UI elements within superordinate UI elements. For example, you can use a UI element for a whole class or material and then use additional UI controls for the characteristics within the class or material.
>   You can use the *Store Navigation* UI control to create a navigation to a nested store definition. You use the *Store Navigation* if the knowledge base contains a material that is modelled as a solution and has its own knowledge base for which you have already composed a store definition. The *Store Navigation* UI control creates a hierarchical structure for the store pages.

5. Configure the UI controls and other UI elements by choosing each UI element on the canvas and configuring it in the Properties screen area.

   > i Note
   >
   > Which options you have for configuring a UI element depends on the type of UI element. For example, you can enter text for a header, or specify width and height.

6. Complete the page by selecting other knowledge base elements and matching them with UI elements.
7. If required, perform the following optional steps:
   - To compose other pages in the store, choose the tab for the page.
   - To add an additional page, choose (Add) above the canvas and select a predefined page.
   - To change the order of the store pages, drag the tabs and move them into the right order.
8. Choose *Preview*.

   The interactive preview opens in a separate browser tab. You can interact with the UI elements to test the functionality of the store. Error messages contain information about any configuration errors.

   > i Note
   >
   > The UI controls for saving, importing, and exporting a product configuration are disabled in the preview.

9. To save the store definition as a draft, choose *Save*.

10. Choose the home button at the top of the screen to unlock the store definition and return to the workspace. If you close the browser without first navigating to the workspace, the store definition will remain locked and can only be unlocked by a UI administrator.

7.5.6 Committing Store Definitions

Context

You commit a store definition to signal to other users that you have finished composing the store definition and that it is now ready to be reviewed by a UI administrator.

Procedure

1. Choose the store definition to open the *Composer* screen.
2. On the *Composer* screen, navigate to ▶ *Menu* › *Commit* ◀.

7.5.7 Creating New Versions of Store Definitions

Context

You can create new versions of committed or published store definitions.

> **i Note**
>
> You can only create new versions of regular store definitions, not of master store definitions. Master store definitions can be edited, updated, and saved but their version number does not change.

Procedure

1. Choose a committed or published store definition and edit the store definition on the *Composer* screen
2. On the *Composer* screen, navigate to ▶ *Menu* › *Create New Version* ◀
3. A new version of store definition is created in draft mode. A new version number is automatically assigned depending on the original status of the store definition

Example

If you edit a **committed** store definition with the version 1.0 and then save the edited store definition as a draft, the version number of the new draft is **1.1**.

If you edit a **published** store definition with the version 1.0 and then save the edited store definition as a draft, the version number of the new draft is **2.0**.

7.6 UI Administrator Tasks

The following tasks can only be performed by users with the UI administrator role:

- Creating Master Store Definitions and Predefined Pages
- Publishing Store Definitions
- Rolling Back Store Definition Versions
- Migrating Store Definitions to the Product Configurator
- Migrating Store Definitions to the Product Configurator

Related Information

Creating Master Store Definitions and Predefined Pages [page 221]
Publishing Store Definitions [page 223]
Rolling Back Store Definition Versions [page 223]
Migrating Store Definitions to the Product Configurator [page 224]

7.6.1 Creating Master Store Definitions and Predefined Pages

You create master store definitions to configure predefined pages.

Prerequisites

You have UI administrator authorization.

Procedure

1. On the *Create Store* screen, select a relevant workspace

2. In the *Store Definitions* list, navigate to *Menu* > *Create*
3. On the *Knowledge Base* tab, select a knowledge base

   Recommendation

   As this product will not be configured in the Product Configurator, we recommend selecting the knowledge base of a product for which you do not need to create a store definition. If a knowledge base serves as the basis for a master store definition, you cannot create a regular store definition for the same knowledge base.

   > i Note
   >
   > The list contains only the knowledge bases for which no store definitions have been created yet. This ensures that only one store definition exists for each knowledge base.
   >
   > Only knowledge base versions that are available in the selected source system are displayed.

4. On the *Store Template* tab, select a store template

   > i Note
   >
   > You can select any store template in this step.

5. On the *Pages* tab, select the predefined pages that you want to use as the basis for the new predefined pages that you want to configure.

   > i Note
   >
   > A blank page is a page that is not predefined for a specific content.

6. On the *Header Info* tab, enter a name and a description for the store definition. You can also specify tags for product categories

   The information specified on the *Header Info* tab can be used to search and filter for the store definition on various screens of the application.
7. To save the store definition as a master store definition in the *Type* field, select *Master* and click on *Done*
8. On the *Composer* screen, compose the predefined pages using the UI elements in the *Control Palette*
9. Choose the tab of each predefined page and edit the name and description in the *Properties* screen area

   Recommendation

   When UI designers create new store definitions, they need to be able to find the predefined pages they should use for their product or solution. Therefore, it is recommended that you specify this in the name and description of the predefined pages.
10. To save the master store definition and predefined pages, choose *Save*

    You cannot commit or publish master store definitions. They can only be edited, updated, and saved. The version number of the master store definitions does not change in any of these cases.

7.6.2 Publishing Store Definitions

You need to publish store definitions before migrating them to the Product Configurator.

Prerequisites

- You have UI administrator authorization
- The store definition is in *Commit* mode

Procedure

1. Choose the store definition and open the *Composer* screen
2. On the *Composer* screen, navigate to ▶ *Menu* ▶ *Publish* ▶

7.6.3 Rolling Back Store Definition Versions

You can roll back a store definition to remove any changes and restore the definition to a previous version.

Prerequisites

- You have UI administrator user authorization
- The store definitions is in *Draft* mode

Procedure

1. Choose the store definition to open the *Composer* screen
2. On the *Composer* screen, navigate to ▶ *Menu* ▶ *History* ▶

   A list displays the current and previous versions of the store definition.
3. 3\. Select the version you want to restore and choose *Roll Back*

   The restored version is saved as a draft version.

> i Note
>
> You can roll back a published store definition to the state of a previous draft version or committed version. However, the published version remains active in the Product Configurator until you publish the restored version.

Example

If your current version is published with the version number 1.0, you can roll back the store definition to the state of a draft or committed version. In this case, the restored version has the version number 2.0 and is a draft version. The published version 1.0 remains active in the Product Configurator until the new version 2.0 is published

7.6.4 Migrating Store Definitions to the Product Configurator

You use an ABAP report to migrate the published store definitions from the modeling system to the productive or test UI Composer system.

Prerequisites

You need to fulfill the following prerequisites before migrating the store definitions to the product configurator:

- The regular store definitions that you want to migrate are published
- The knowledge bases of your store definitions are created in the S/4HANA system.
- The transport request is created in the S/4HANA system.

Procedure

1. In the S/4HANA system, execute the report `/SLCE/TRANSPORT_STORE_LIST`.
2. Select the store definitions that you want to migrate
3. Choose *Execute*
4. Create or select a customizing transport request
5. Release the transport request

   The store definitions are now available in the productive or test system.

7.7 UI Composer Extension Project

7.7.1 Project Description

The UI Composer extension project is an eclipse-based Web application project. It is used to host the following custom components used by the UI Composer:

- Custom Components
- Custom Templates
- Custom Widgets
- Custom metadata files
- Custom CSS

Important areas of the Web application are described in detail below

Apache Maven Files

Files	Description
`assembly.xml`	Assembly descriptor file used by the Maven build.
`pom.xml`	Maven build file. In order to avoid caching on the browser, before uploading the files to the server, it is important to increment the version number in the pom.xml file. The build will then update the `manifest.properties` file which the UI COMPOSER references.

Web Content Folder

Files	Description
`Ui5RepositoryTextFiles`	File contains the list of file extensions that the ABAP upload program (explained later in this document) needs to interpret as text files.
`index.html`	Dummy `index.html` file; currently not used.

Files	Description
`manifest.properties`	File containing the project version number. This version number is used as part of the deployment path of the extension project when uploading it to the SAP Gateway Hub. This file is referenced by the UI Composer to dynamically fetch the project version number in order to properly locate the extension project deployed on the server.

Resources Folder

Files	Description
`components.scss`	SASS file containing all the styles of the custom components, templates and widgets. This file needs to be compiled by the Compass tool in order to generate the corresponding components.css file which is referenced by the UI Composer.
`custom.scss`	SASS file containing all style overrides of existing standard components, templates and widgets. This file needs to be compiled by the Compass tool in order to generate the corresponding custom.css file which is referenced by the UI Composer.
`compile.command config.rb`	Compass tool-specific files, you do not need to change these unless you are familiar with how the Compass tool works.

WEB-INF Folder

Files	Description

`web.xml`	File containing the web application parameters for local deployment in Apache Tomcat (or any Java web server).
	The important section to maintain is:
	Sample Code
	`<context-param>`
	`<param-name>com.sap.ui5.proxy.REMOTE_LOCATION</param-name>`
	`<param-value><protocol>:<Gateway host>:<Gateway port></param-value>`
	`</context-param>`
	If the SAP Gateway Hub server name or port ever changes, you will need to update the `param` value field.

Javascript File Package Structure

Files	Description
`sap/UI Composer/extension/component`	
`CustomWeatherButton.js`	Example of a custom component that can be added to the UI Composer.
`CustomWeatherButtonRen derer.js`	Renderer class for the CustomWeatherButton component. The standard is always to use the component name in the renderer name and just append “Renderer”.
`sap/UI Composer/extension/metadata`	
`CustomControl.json`	Custom JSON metadata file for the custom components and widgets
`CustomStoreTemplate.json`	Custom JSON metadata file for the custom store templates
`sap/UI Composer/extension/storetemplate`	
`FooterViewExtension.js`	Example of a custom `FooterView` used in a custom store template
`FooterViewExtensionRenderer.js`	Renderer class for the `FooterViewExtension`

Files	Description
`HeaderViewExtension.js`	Example of a custom `HeaderView` used in a custom store template
`HeaderViewExtensionRenderer.js`	Renderer class for the `HeaderViewExtension`
`StoreTemplateExtension.js`	Example of a custom store template
`StoreTemplateExtensionRenderer`	Renderer class for the `StoreTemplateExtension`
`sap/UI Composer/extension/widget`	
Empty placeholder folder for custom widgets	

7.7.2 Packaging the Extension Project for Deployment

Context

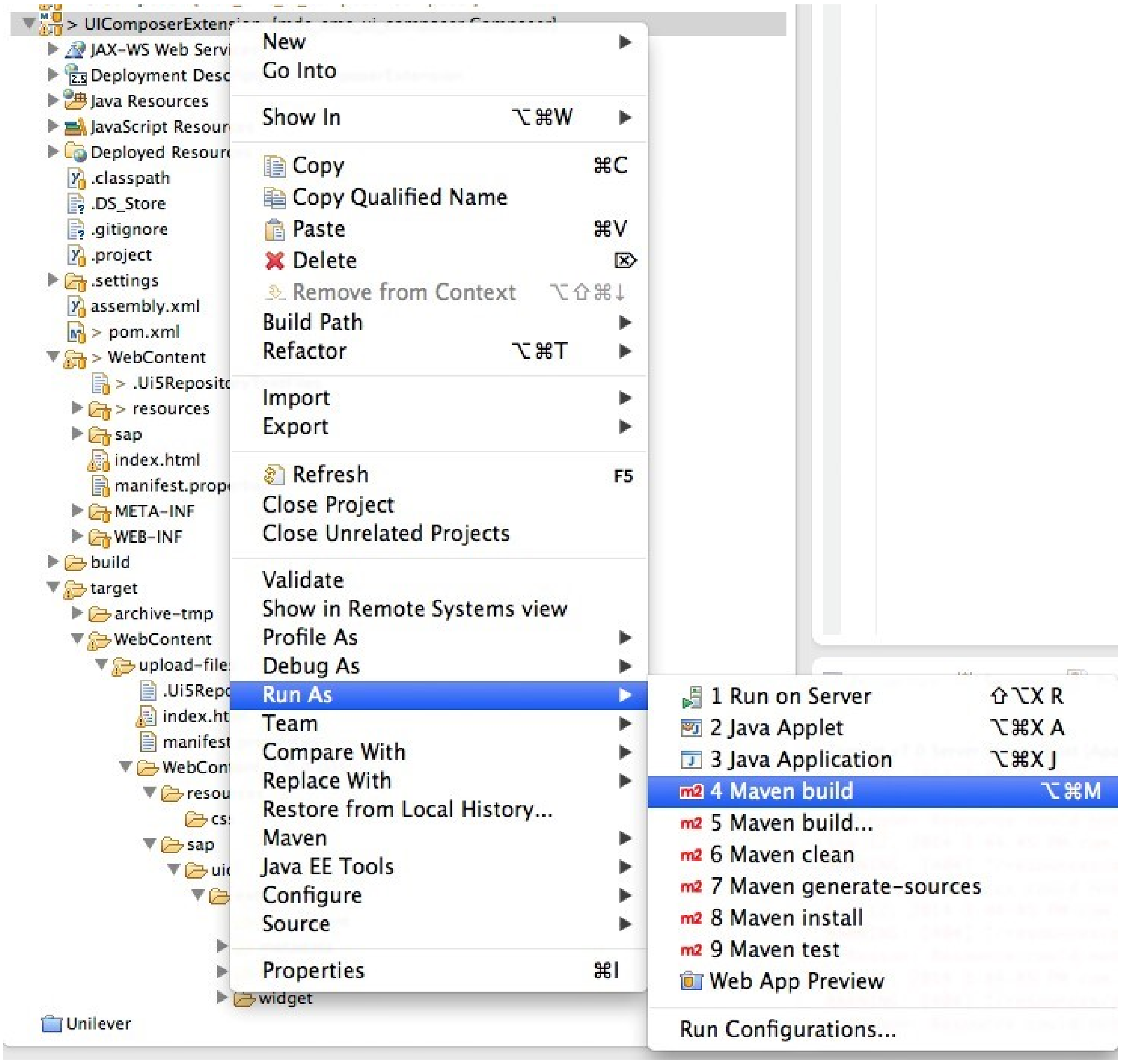
In order to upload the files to the SAP Gateway Hub server, the extension project needs to be packaged in a certain way.

The `manifest.properties` file contains the version number of the project. This version number is used as means to reference the project from the UI Composer and to avoid caching issues after each deployment.

The tool used to package the extension project is Apache Maven (Maven). Maven is built-in to Eclipse, so there are no additional plug-ins to install.

Procedure

1. To run Maven, right-click on the project and choose *Run As* › *Maven build*

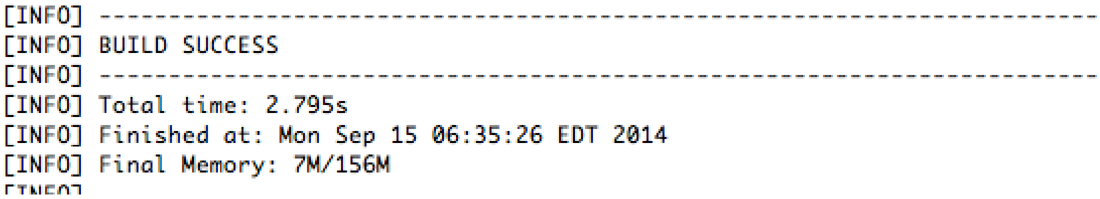


Running the Maven Build

2. Enter `clean install` in the *Goals* field of the resultant dialog box
3. Choose *Run*

   When the build completes, you should see a success message.

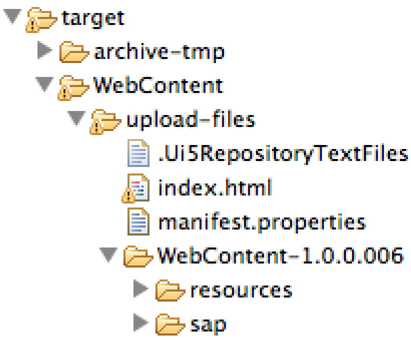
   For example,



```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.795s
[INFO] Finished at: Mon Sep 15 06:35:26 EDT 2014
[INFO] Final Memory: 7M/156M
[INFO]
```

Maven Success Message

The build process creates a folder in the `target` directory, as shown below:



```
target
  archive-tmp
  WebContent
    upload-files
      .Ui5RepositoryTextFiles
      index.html
      manifest.properties
    WebContent-1.0.0.006
      resources
      sap
```

Maven Build Target Folder

When using the upload program (described in **Deploying the Extension Project**) to deploy the app on the SAP Gateway server, you need to point to a folder. For this purpose, use the `upload-files` folder shown in the figure above. This folder contains all the necessary files that the UI Composer needs to integrate the custom components, JSON files, and CSS files.

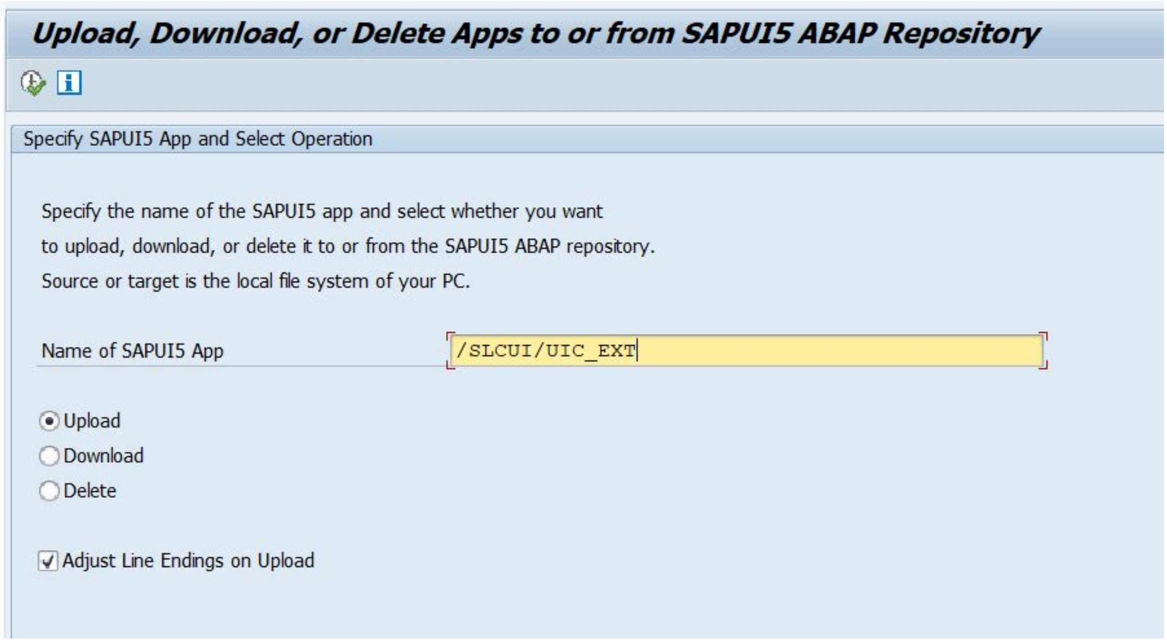
Related Information

Deploying the Extension Project [page 230]

7.7.3 Deploying the Extension Project

Uploading the Files to the SAP Gateway Hub Back-End

1. In the SAP Gateway Hub system, go to transaction `SE38` and run the program `/UI5/UI5_REPOSITORY_LOAD`.
2. Enter the name of the extension BSP application: `/SLCUI/UIC_EXT Composer_ext`



Upload, Download, or Delete Apps to or from SAPUI5 ABAP Repository

Specify SAPUI5 App and Select Operation

Specify the name of the SAPUI5 app and select whether you want to upload, download, or delete it to or from the SAPUI5 ABAP repository. Source or target is the local file system of your PC.

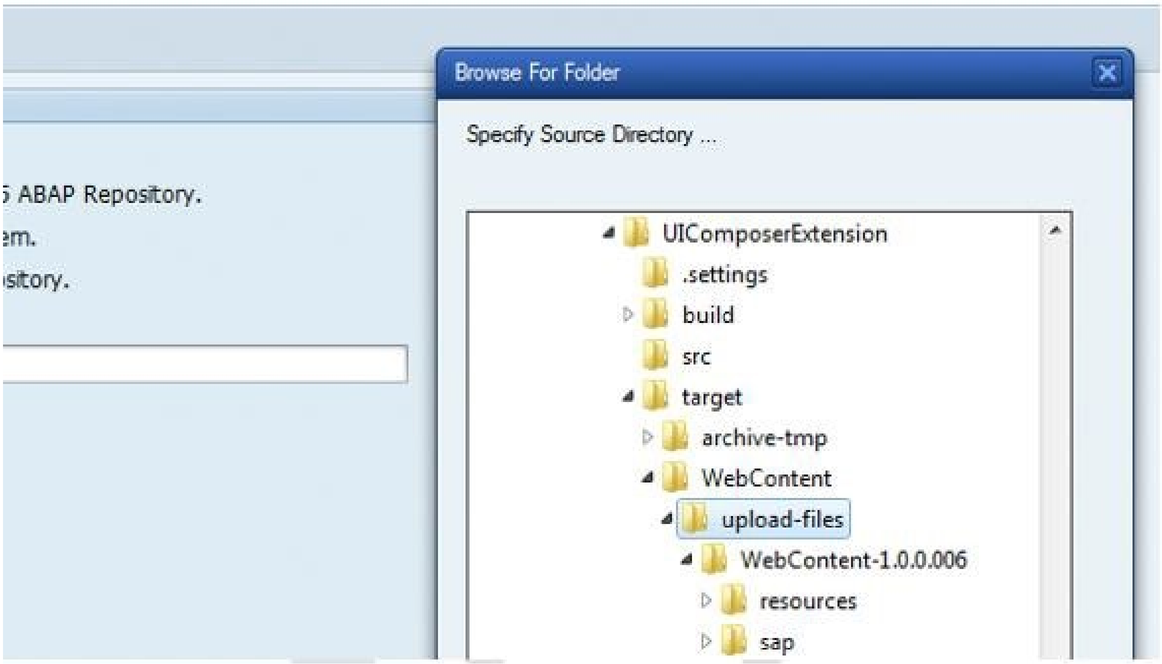
Name of SAPUI5 App: /SLCUI/UIC_EXT

● Upload
○ Download
○ Delete

☑ Adjust Line Endings on Upload

SAPUI5 Upload Program

3. Select the `upload-files` folder that you have generated with the Maven build of the UI Composer extension project, and choose *OK*



Folder Selection

4. The confirmation screen appears.
   You will be informed if certain files cannot be uploaded. For example, if the file type isn't recognized. If so, you will need to add the file type in the `.Ui5RepositoryTextFiles` file.
5. If there are no errors, scroll to the bottom of the screen and choose *Click here to Upload*.
6. Enter the transport request number and the codepage parameter `UTF-8`
7. When the upload is complete, you will be returned to the initial screen of the program
8. Test your extensions in the UI Composer and Store Preview using the URLs:

- UI Composer: `http://[gateway:port]/sap/bc/ui5_ui5/slcui/UI Composer_web`
- Store Preview: `http://[gateway:port]/sap/bc/ui5_ui5/slcui/stpr_web`

Local Deployment

The UI Composer extension web application can be deployed on your local Apache Tomcat (Tomcat) and it will load the UI Composer application deployed on the SAP Gateway Hub, taking your local extensions into account. This way, you do not need to upload your extension code to SAP Gateway Hub to test it in the UI Composer.

Once deployed through eclipse to your local Tomcat installation, you can access the app via this URL (assuming your Tomcat http port is 8080):

`http://localhost:8080/UIComposerExtension`

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.
About the icons:

- Links with the icon ➦ : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon ➦: You are leaving the documentation for that particular SAP product or service and are entering an SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.
The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Bias-Free Language

SAP supports a culture of diversity and inclusion. Whenever possible, we use unbiased language in our documentation to refer to people of all cultures, ethnicities, genders, and abilities.

www.sap.com/contactsap

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

THE BEST RUN SAP