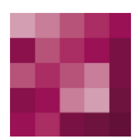


FirstSpirit™
Your Content Integration Platform

FirstSpirit CorporateContent

FirstSpirit Versions 4.0 and 4.1

Version	**1.09**
State	**RELEASED**
Date	**2012-06-06**
Department	Techn. Documentation
Author/ Authors	B. Ehle
Copyright	2012 e-Spirit AG
File name	PACK40EN_FIRSTspirit_PackagePool

e-Spirit AG
Barcelonaweg 14
44269 Dortmund | Germany

T +49 231 . 477 77-0
F +49 231 . 477 77-499

info@e-spirit.com
www.e-spirit.com

Table of Contents

1 Introduction 5

1.1 Topics covered in this document 5

2 Terms and Concepts 7

2.1 Package 7

2.1.1 Package types 7

2.1.2 Package dependencies 7

2.1.3 Package definition and package version 10

2.2 Publication groups 11

2.3 Subscription 12

2.3.1 Updating packages in the subscription 13

2.3.2 Subscribe to metadata and project settings 14

2.3.3 Release 14

2.4 Integrating workflows and scripts 15

3 Configuration 16

3.1 Check license file 16

3.2 Start PackageManagerService 17

4 Package menu item (Master project) 19

4.1 Create new packages 19

4.1.1 Select package type 19

4.1.2 Edit package properties 20

4.1.3 Define permissions for a package 25

4.1.4 Changing package types and defining package dependencies 27

4.1.5 Configuring events for a package 28
4.1.6 Deactivate namespace extension (V4.1 and higher) 34
4.1.7 Changing conflict resolution on importing (V4.1 and higher) 39
4.2 Edit packages 41
4.2.1 Package list 41
4.2.2 Edit package properties 43
4.2.3 Edit package version 43
4.2.4 Generate package version 44
4.2.5 Edit package availability 47
4.2.6 Activate specific events 48
4.2.7 Edit package content 49
4.2.8 Integrate structure variables 53
4.3 Publish packages 56

5 Subscription menu item (target project) 59
5.1 Create new subscriptions 59
5.1.1 Choose package 60
5.1.2 Create subscription for a package 61
5.1.3 Limit package content in the subscription 63
5.1.4 Configure events for a subscription 65
5.1.5 Configure structure variables 66
5.1.6 Create subscription 66
5.2 Edit subscription 67
5.3 Update subscription 69
5.4 Combine package and target project content 72
5.4.1 General information 72
5.4.2 Combine sections 72
5.4.3 Order for importing objects into the target projects 73

6 Overview menu item ... 78
6.1 Detail information ... 81
6.1.1 Detail information on subscriptions ... 81
6.1.2 Detail information on packages ... 83
6.1.3 Display logs ... 84

7 Publication Groups menu item ... 86
7.1 Edit publication groups ... 87
7.2 Add publication group ... 89
7.3 Delete publication group ... 90

8 Package Pool context menu ... 91
8.1 Add to package (master project) ... 91
8.2 Remove from package (master project) ... 93
8.3 Undo package relation (target project) ... 95
8.4 Change state (target project) ... 97
8.5 Rebind original (target project) ... 99

9 Transfer existing projects into package master projects 102
9.1.1 Using the reference graph ... 102
9.1.2 Structuring the package content ... 104
9.1.3 Limiting the picture selection in templates ... 105
9.1.4 Limiting the template selection ... 107
9.1.5 Avoiding language-dependent structures in templates ... 108
9.1.6 Automatic conversion in the Page Store ... 108
9.1.7 Manual conversion of templates ... 109
9.1.8 Manual conversion in the Content Store ... 111
9.1.9 Checking the functionality in a test project ... 112

9.2 For the same types of projects ... 112
9.3 Export / Import ... 113
9.3.1 Master package projects ... 113
9.3.2 Subscribing projects ... 113

10 PackagePool for developers ... 115
10.1 Individualizing the package content in the target projects ... 115
10.1.1 Layout changes via structure variables ... 115
10.1.2 Layout changes via templates ... 115
10.2 Multilingualism support ... 116
10.2.1 Page content ... 116
10.2.2 Language-dependent media and files ... 119
10.2.3 Menu structures ... 119
10.2.4 Templates ... 120
10.3 Using workflows and events ... 124
10.3.1 Determining the affected nodes ... 124
10.3.2 Exemplary workflow for the release ... 125

11 Joint database access ... 129
11.1 Configuring the target projects (read DB access) ... 131
11.2 With existing databases ... 132
11.3 New databases ... 133
11.4 “contentSelect" function ... 133
11.5 Language-dependent content ... 135
11.5.1 Implicit modeling of the language dependency ... 135
11.5.2 Explicit modeling of the language dependency ... 136
11.6 Different database layer in the master and target project ... 137

1 Introduction

This "FirstSpirit PackagePool" document describes the licensed FirstSpirit "PackagePool" function. It enables objects from the FirstSpirit Client, for example, pages including all links, to be collated in so-called packages and made available for importing into different target projects. The advantage of this function: The objects can be managed in a central place (in the master project). The import into the target projects then takes place automatically or manually by subscription. This significantly simplifies working with uniform data throughout the entire company. The objects, such as company logos or section templates, are simply provided in a package and can be imported into all target projects. If the package objects, for example a company logo, change, a new package version is created and is automatically imported into all target projects, which have taken out a valid subscription for this package. This ensures that the objects are always kept up to date.

Provided a valid license exists, the new function is located in the FirstSpirit Client under the "PackagePool" menu item (see Chapter 3.1 page 16).

Project Tasks Search Package pool View Extras Help

Figure 1-1: Main menu item – PackagePool

1.1 Topics covered in this document

Chapter 2 explains the most important terms and concepts for working with packages and subscriptions. The chapter gives a general overview of the how the PackagePool works and facilities entry *for first-time users* (from page 7).

Chapter 3 describes the *configuration settings* on the server. This chapter is only relevant for *administrators* (page 16 ff.).

Chapter 4 deals with all menu items and dialog boxes for creating, editing and publishing *packages*. The chapter is only relevant for advanced users, who may create their own packages (*project administrators*) or have permissions to edit a package (*qualified persons*) (see page 19 ff).

Chapter 5 explains how to handle subscriptions. It describes how to create and edit subscriptions and also introduces various options for importing packages. This chapter is directed at *all users*, who want to import packages into products (from

page 59 ff.).

Chapter 6 presents a *package management overview*, which shows all the relationships between the different packages and projects and is a useful function for *all users* of the PackagePool (page 78 ff.).

Chapter 7 deals with the concept of *publication groups*, which facilitate the publishing and importing of packages for users in complex work environments. This chapter is also only relevant for *advanced users* (page 86 ff.).

Chapter 8 explains how to work with the *context menu* and the functions available in it, for *all users* of the PackagePool (from page 91).

Chapter 9 describes in depth how to *transfer existing projects* into a master project package. This chapter is relevant for *Package Developers* (page 102 ff.).

Chapter 10 points out several important *aspects for package developers* and, in particular, describes the *multilingualism* support provided by the PackagePool and *adjusting the content* in the target projects (page 115 ff.).

Chapter 11 introduces the option of *joint database access* linked with the PackagePool. It can be implemented not only for existing databases, but also for a new, jointly used database. This chapter is only relevant for *Administrators Developers* (page 129 ff.).

2 Terms and Concepts

2.1 Package

Packages are created and edited in the master project. The master project is the term used to name the project, which provides the package for importing into other projects. In a package, different objects are grouped together and stored as a compressed zip file. The objects are chosen from the project tree of the master project. At the same time, a so-called start node is defined. All lower level objects, including complete folders, are copied into the package from this node. If all the required objects have been collated in the package, a new package version is generated, which is then available for importing by all target projects with a valid subscription.

2.1.1 Package types

FirstSpirit differentiates between two types of packages:

- **Content packages:** Content packages contain objects from the Page Store, the Site Store and the Media Store. They do not contain *any templates* or objects from the Content Store.

- **Template packages:** Objects from the Template Store are integrated in template packages. In addition, a template package may contain objects from the Content Store and the Media Store. However, the integration of objects from the Media Store into a template package should be limited to media, which is referenced directly in the templates and is used, for example, for the layout (cascading style sheets, spacer.gif, logos, etc.). Other media objects continue to belong in a content package.

Each object can only be integrated in one package!

2.1.2 Package dependencies

As already explained in Chapter 2.1, different objects are grouped together in packages. Most objects, with the exception of Media Store objects, can reference other objects. A page from the Page Store references, for example, an image from

the Media Store and a template from the Template Store. In order to successfully import objects into different projects, the dependencies between the objects must be resolved. This means, it is necessary to ensure that all objects referenced within a package are also contained in the package. This is the reason for the strict separation into content and template packages (see Chapter 2.1 page 5).

A differentiation is made between two dependencies:

1. Content dependencies:
   On the one hand there are content dependencies. These dependencies within a content package are **automatically** resolved, with the help of the so-called reference graph (see Chapter 9.1.1 page 102). Here, for example, each page to be copied into a package is checked to find out which objects it references. The referenced objects are then copied into the package too. If an object that is to be copied into a content package is, e.g. a page reference or a folder from the Site Store, all corresponding pages from the Page Store are also copied into the package.
   If referenced objects are already integrated in a content package, **they cannot be copied into another content package, as each** object may only be contained in a single package. In this case, the system establishes a dependency with this independent content package. This is displayed on generating a package version (see Chapter 4.2.4 page 44) or in the version list of a package (see Figure 4-18) and in the detail information of the package (see Figure 6-6). The dependent content packages can then be manually subscribed to (see Chapter 5.3 page 69). A content package can have several dependent content packages.

2. Dependencies on templates
   On the other hand there are dependencies on templates. The dependency of a content package on a template **cannot** be resolved **automatically**. The relationship between a content and a template package must be given in the properties of the content package (see Chapter 4.1.4 page 27). If a dependency exists between a content package and a template package, a specific order must be kept to on generating a version in the master project (see Chapter and for publishing from the master project (see Chapter 4.3 page 56).
   Templates can also be dependent on other templates. These dependencies **cannot always** be resolved **automatically**, as some effects would be very far-reaching. On making a template package, the package developer should therefore think in advance about the dependencies and the most effective possible package structure. The order in which items are added to a package must also be considered. For example, if a template is dependent on a

content source, the corresponding database schema (incl. table templates and queries) must be added to the package first (see example in Chapter 11 page 129).

For a subscription, this means that: One package can depend on another package (content or template package). To subscribe to a package, all dependent template packages must be subscribed to and all dependent content packages can also be subscribed to. The import order is not random:

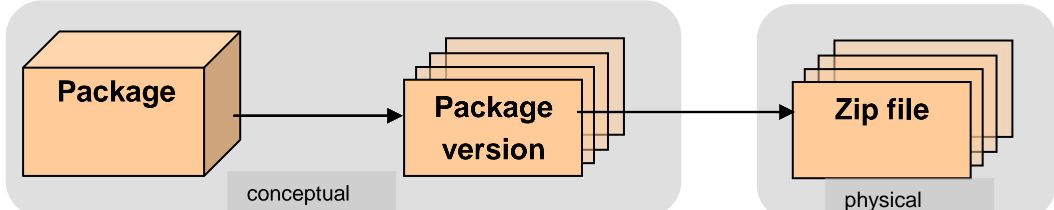
*Whenever a content package that is **dependent on a template package** is imported, the template package must be imported first and only then the corresponding content package. If this order is not followed, an error message appears ("The dependent template package (Templates) is not up to date. Please import the template package first.", see Chapter 5.3 page 69) and the user can restart the import.*

*Whenever a content package that is **dependent on another content package** is imported, the dependent content package must be imported first and only then the content package, which contains the references to the dependent objects. If this order is not followed on importing or the dependent content packages are not imported, this can cause errors in the target project (see Chapter 5.3 page 69).*

A specific order must also be followed for publishing dependent content packages (see Chapter 4.3 page 56).

2.1.3 Package definition and package version

Regardless of its specific type (content or template package), a package is made up of one or several package versions. Each **package version** consists of precisely one Zip file, which is used for importing into the target projects.



The zip file contains all data required for the package version as well as a Meta description of the package content. This Meta description is called the **package definition**. The package is defined hierarchically on the basis of a list of start nodes. These start nodes determine which objects are part of the package. When the package is generated, all objects located below the start node are copied into the package. Precisely which objects these are depends on the structure and content of the master project.

Apart from the package definition, the dependencies between individual objects must also be taken into account (see Chapter 2.1.2. page 7). If dependencies exist between an object contained in the package and another object, which is not part of the package, the dependency is automatically identified with the help of the reference graph and the referenced object is added to the package, although it is not explicitly part of the package definition. This means a package cannot be solely defined by the selected package content. Therefore, differentiation between the package definition and package version is of central importance.

Package definition:
Describes the content of a package with the help of the start nodes from the master project integrated in the package. This node list does not contain any referenced objects and they are therefore also not part of the package definition. The complete content does not result until a *package version* is generated with the help of the *package definition*.

Package version:
Contains all objects determined with the help of the package definition and all manually referenced objects. A package version therefore provides a complete description of the package content. Unlike the package definition, which always reflects the most up-to-date content, a package version is only as current as the date on which it was last created.

The package content changes if:

- a new start node is explicitly added to the package, i.e. if there is a *change in the package definition*.
- an object is added implicitly, because it was created as a new object in the master project, below a start node (*no change to the package definition*).

In these two cases the package should be updated by generating a new package version (see Chapter 4.2.4 page 44). A package version can be released for one or several publication groups.

> *If a new object (e.g. a picture) is created in an existing package, below an already integrated start node, this object is automatically added to the package and is included in the next package version.*

> *When creating packages, there must be no overlapping between the package content. This means that each project node can only belong to precisely one package. Project nodes and objects already used in a package are recognized by the name extension "ObjectName@PackageName" (in the reference name; provided the name extension is not disabled, see Chapter 4.1.6 page 34) and a corresponding symbol in the project tree. In the "Classic" Look & Feel this is a blue dot, in the "LightGray" Look & Feel it is a package symbol. This method increases clarity, as otherwise, if a single object is changed, several new package versions would have to be created and published.*

2.2 Publication groups

Creating and publishing packages is a complex task. Incorrect action by the user can cause problems and conflicts in target projects. Before packages are used in a productive environment, they should therefore be fully tested. To this end, the concept of the "publication group" was introduced (see also Chapter 7 page 86). A publication group is a kind of "marker", which can be assigned to one or several package versions. On the master project side, packages can be "released" for specific publication groups and on the target projects side, in the subscription to a package, it is defined for which publication group the package is required. Publication groups are defined server-wide and can therefore be used not only in the

master projects but also in the target projects.

For example, the following publication groups could be defined:

- Development: for the development of packages.
- Test: for projects, which are for testing packages.
- Productive use: for projects which use a package in the productive ("live") environment.

The exemplary procedure is then as follows:

Package version	Release for Pub. Group
Version 0.1	Development
Version 0.2	Development
Version 0.3	Development, test
Version 0.4	Development, test
Version 1.0	Productive use
Version 1.1	Development

The "development" group begins with the development of a package. The first package versions 0.1 and 0.2 are only released for this group. The development continues until, at a certain point in time, package version 0.3 is created, which is released not only for the "development" publication group, but also for the "test" group. As a result, this version of the project is available for all projects with a subscription to the "development" and "test" publication groups. Depending on the project configuration, the target project is now updated either automatically or manually. If the development of the package is completed, a new package version 1.0 can be released for the "productive use" group.

As shown in the example, several versions can be released for a publication group. In this case, the package version with the highest package number, i.e. the most recent version, is always used. The package number is unique and is generated when a new package version is created.

2.3 Subscription

Subscriptions are created and edited in the target projects. Target projects is the term used to describe projects, which can import the packages from a master project (see Chapter 2.1 page 7). Only packages, which are defined in the master project as

"available" can be subscribed (see 4.1.2.1 page 22).

A differentiation is made between two subscription states:

1. **Initialization:** Initially, with a subscription, all package content (e.g. all media files of the master project) is copied into the target project and, depending on the package or subscription configuration, can then be further edited by the target project editors.

2. **Updating:** As soon as, in the master project, something changes in the objects integrated in the package or new objects are to be made available, e.g. a new picture, a new package version must be created. Each new package version not only contains the changes to the preceding version, but also all objects of the preceding version that have not yet been changed. But when the update with a new package version takes place in the target project, only the new objects added and changed objects are exchanged.

2.3.1 Updating packages in the subscription

A package update can be performed with the help of two different methods:

1. **Automatic updating:** In the case of automatic updating, the decision whether to update a package is made by the **master project administrator**. They initiate the updating of all target projects with a valid subscription to this package centrally, by publishing the package (see Chapter 4.3 page 56). In this case, this is also called a "push" process. It is not necessary for the person responsible for the target project to intervene manually.

2. **Manual updating:** In the case of manual updating, the decision whether to update a package is made by the **target project administrator**. They are provided with a new package (visible e.g. in the Package Overview, see Chapter 6 page 78, or in the Subscription List, see Chapter 5.1.6 page 66) and can use the new package to update their project if necessary (see Chapter 5.3 page 69). In this case, this is also called a "pull" process. With a manual update, the administrative work lies in the target projects.

Three possible states are feasible for an update:

- An object from the master project is created as a new object in the target project.

- An existing object in the target project is updated with content from the master project.
- A conflict situation occurs (see also Chapter 8.4 page 97).

To simplify the update and avoid errors in the productive projects, publishing groups (see Chapter 2.2 page 11) have been defined.

2.3.2 Subscribe to metadata and project settings

In most projects, in addition to the default page templates, there is also a template page for the global project settings and for so-called metadata, if they are used in the project. These templates can be part of a template package, and therefore can be imported into all target projects with a valid subscription. By integrating a project settings template, for example, it is possible to define uniform, project-wide layout requirements for headings or continuous texts. By integrating metadata, for example, it is possible to work with personalized pages. If these templates are imported into the target projects, they can be easily extended there and adjusted to the project-specific circumstances.

The imported metadata templates must be set in the server and project configuration in the project settings, under the "Options" item, in the "Metadata Template" field.

In both cases, these templates can be imported into the target projects, but they do not necessarily have to be used. This can lead to problems, if other packages are based on these project settings or metadata.

2.3.3 Release

The project-specific concept of the release rule can also be used for working with packages. When a package is subscribed to, the subscriber decides whether or not the subscribed to content is to be released automatically. If **automatic release** is chosen, all new or changed objects are automatically released directly after they have been imported, without any action on the target project side (release via workflows, see Chapter 2.4 page 15). In this case, the editors of the target project

cannot see which objects have been changed.

In contrast to this procedure, an **explicit release** can be defined. In this case, the changed or new objects are displayed in red in the project tree of the target project and have to be explicitly released by a responsible user or editor. Advantage: The changes are visible at a glance. This solution provides more transparency, but would not be very convenient especially where extensive package content is involved. For this reason, the explicit release can be given with the help of a simple workflow. If a package update occurs, a list of the objects to be released is created at the same time and is announced within the subscription. All objects from this list can then be released with only one workflow (see also Chapter 10.3.1 page 124).

2.4 Integrating workflows and scripts

The updating and importing of packages mostly takes place in complex project environments. To make the packages as convenient as possible to work with and to prevent errors, the integration of workflows in the target projects is of central importance. In this case, certain events are announced in each package. Each of these events can then be assigned a workflow or a script, which is started after the package is imported. Examples of such events are:

- Automatic release: Directly after the import a workflow is started, which automatically releases all new or changed objects, without any action on the target project side (see Chapter 2.3.3 page 14).

- Resolve conflict: If a package conflict occurs when a package is imported, a workflow is started when this event occurs, which is intended to remove the conflict.

- Report function: The report function is especially interesting for large projects; when a package is imported; it creates a log file and informs certain responsible groups of people about the updates.

In a subscription, the assignments created in the package are adopted by default for events, but can be reconfigured in the target project. It is also possible to define project-specific events in the target projects and to assign separate scripts to the events for editing.

Apart from the execution in the target projects, workflows can also be used in the master project. A package update can also be initiated by a workflow or a script.

3 Configuration

The Package Pool is a licensed function; this means that the "Package Pool" menu item is only displayed in the menu bar of the FirstSpirit editing environment if a valid license exists for this function.

Two steps are necessary to activate the function.

- Check license file and if necessary replace (see Chapter 3.1 page 16)
- Activate PackageManagerService (see Chapter 3.2 page 17)

3.1 Check license file

The "FirstSpirit – Configuration – License" menu of FirstSpirit Server Monitoring can be used to display the valid FirstSpirit functions of the license file `fs-license.conf`. The `license.PACKAGEPOOL` parameter must be set to value `1` for use of the PackagePool (see Figure 3-1).

If not, a new valid license can be requested from the manufacturer and added in the blue part of the window. The new license file can be saved by clicking the Save button.

Manipulating the `fs_license.conf` results in an invalid license. If changes become necessary, please contact the manufacturer.

When adding a new `fs_license.conf` configuration file, it is not necessary to restart the server. The file is automatically updated on the server.

License

```
license.ID=365
#FIRSTspirit license
#Mon Jan 02 08:37:22 CET 2012
license.USER=e-spirit
license.EXPDATE=15.07.2012
license.MAXPROJECTS=0
license.MAXSESSIONS=0
license.MAXUSER=0
license.SOCKET_PORT=0
license.VERSION=4
license.MODULES=personalisation,search,integration,newsletter,portal,form_edit,enterprise_search
license.WEBEDIT=1
license.WORKFLOW=1
license.REMOTEPROJECT=1
license.PACKAGEPOOL=1
license.DOCUMENTGROUP=1
license.ACCESS_API=1
license.APPTAB_SLOTS=100
license.ARCHIVE=0
license.CLUSTERING=1
license.ENTERPRISE_BACKUP=1
license.HIGHAVAILIBILITY=1
license.OFFICE_IMPORT=1
license.OFFICE_INTEGRATION=1
license.SCOPE=CORPORATE
license.TYPE=PRODUCTION
```

Please insert the unchanged content of the licence file here

Figure 3-1: Display of the license file parameters (server monitoring)

3.2 Start PackageManagerService

In the next step the PackageManagerService must be started on the server. The service can be activated via FirstSpirit Server Monitoring in the "FirstSpirit – Control – Services" area (or using the server and project configuration application).

Services

Name	Comment	Type	Start/Stop	Reboot
PackageManager	PackagePool Service	System	start	reboot
PermissionService	Permission Service	System	stop	reboot

Figure 3-2: Activating the PackageManagerService (server monitoring)

Click the "start" entry to start the service. It is not necessary to restart the server.

The configuration for automatic starting of the service with each server restart can be defined in the "FirstSpirit – Configuration – Services" area.

For information on configuring FirstSpirit Server Monitoring, please also refer to the *FirstSpirit Manual for Administrators, Chapter 8*.

4 Package menu item (Master project)

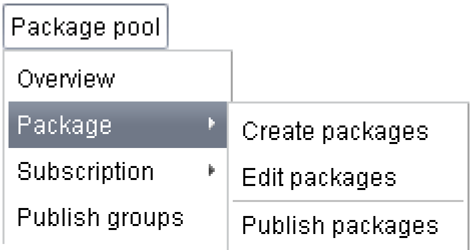


Figure 4-1: Package menu item

The Package menu item is only relevant for the master project, which provides the packages for importing into the target projects. All settings concerning a package are made in the Package menu item. New packages can be created (Chapter 4.1 page 19); existing packages can be adjusted and made available within a package version for importing into the target projects (Chapter 4.2 page 41). It is also possible, from the master project, to initiate automatic package updating for all target projects (Chapter 4.3 page 56).

Packages can be deleted using the Edit Packages menu item (Chapter 4.2.1 page 41).

4.1 Create new packages

To create a new package, the submenu item "Create packages" is opened. Creating a new package involves several steps, which are explained in the following. The initial creation of a package can only be carried out by the administrator of the master project.

4.1.1 Select package type

The "Create Packages" menu item first opens the "Select package type" dialog box (see Figure 4-3).

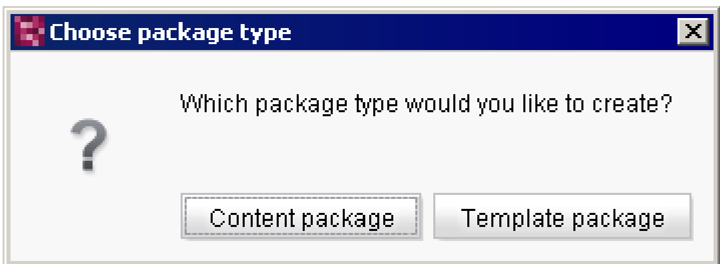


Figure 4-2: Dialog box – Select package type

The package type (see Chapter 2.1 page 7) for the new package can be assigned here. The package type is also displayed in the "Edit package properties" dialog box, but cannot be changed there.

Content package: Click this button to select a content package as the package type. A content package may only contain objects from the Page, Media and Site Store and only these Stores are displayed in the choice available for selecting the package content.

Template package: Click this button to select a template package as the package type. A template package may only contain objects from the Template, Content and Media Store and only these Stores are displayed in the choice available for selecting the package content. The media integrated here should be limited to only *media directly referenced in the templates*. Other media objects should be integrated in a content package.

If objects from a database schema are to be integrated in the package, the database configuration must be adjusted in the project properties of the target project (see Chapter 11 page 129). Otherwise a corresponding error message will be output when the package is subsequently imported into the target project (see Chapter 5.3 page 69).

4.1.2 Edit package properties

Regardless of the type selected, the "Create package" dialog box then opens (corresponds to the "Edit package properties" dialog box). All initial settings for the package are made there by the administrator of the master project. The settings made here can be changed later using the "Edit package properties" dialog box (see Figure 4-3: Dialog box – Edit package properties). Only the package name and the package type can no longer be changed (see Chapter 4.1.2.1 page 22).

In FirstSpirit Version 4.1 and higher advanced configuration options are also

available for packages. The namespace extension, which to date was assigned for all project content, can now be disabled by the template developer for all or for only certain object types (see Chapter 4.1.6 page 34). At the same time, the conflict handling on importing the content into a target project can also be adjusted (see Chapter 4.1.7 page 39).

For details of advanced settings, see Chapter 4.1.2.2 page 24.

4.1.2.1 Settings

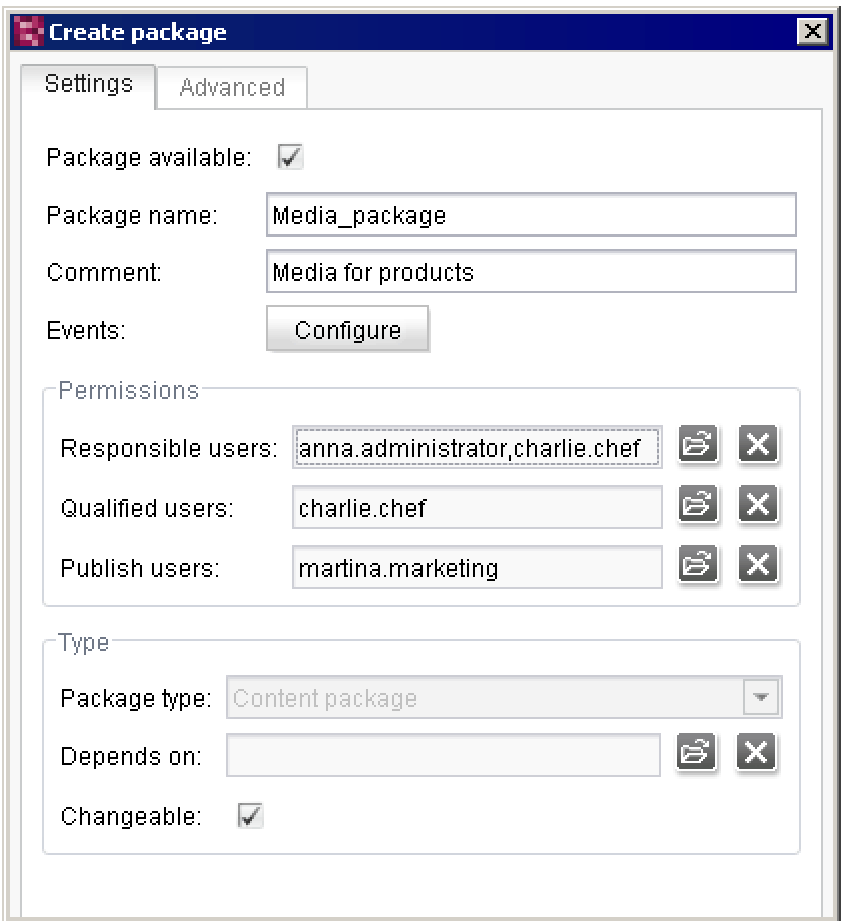


Figure 4-3: Dialog box – Edit package properties (new Look&Feel)

Package available – if this checkbox **is enabled**, the new package is made available to all target projects. If the checkbox is **disabled**, the package is not made available and cannot be selected for a subscription within the target projects.

Package name – unique name of the package, is assigned initially when the package is created and can no longer be changed later.

Comment – optional comment on the package.

Configure – Click this button to open the "Configure events" dialog box (see Chapter 4.1.5 page 28).

Permissions – the persons authorized for a package are given here (see Chapter 4.1.3 page 25).

Type – the package type can be changed here or a package dependency can be

defined (see Chapter 4.1.4 page 27).

Cancel – Click this button to interrupt the process, the new package is not created.

OK – Click the button to open the next "Edit package" dialog box. Here the package properties can be revised, content added to the package and new package versions can be created. The procedure is described under the "Edit packages" menu item (see Chapter 4.2 page 41).

For details of advanced settings, see Chapter 4.1.2.2 page 24.

4.1.2.2 Advanced (in 4.1 only)

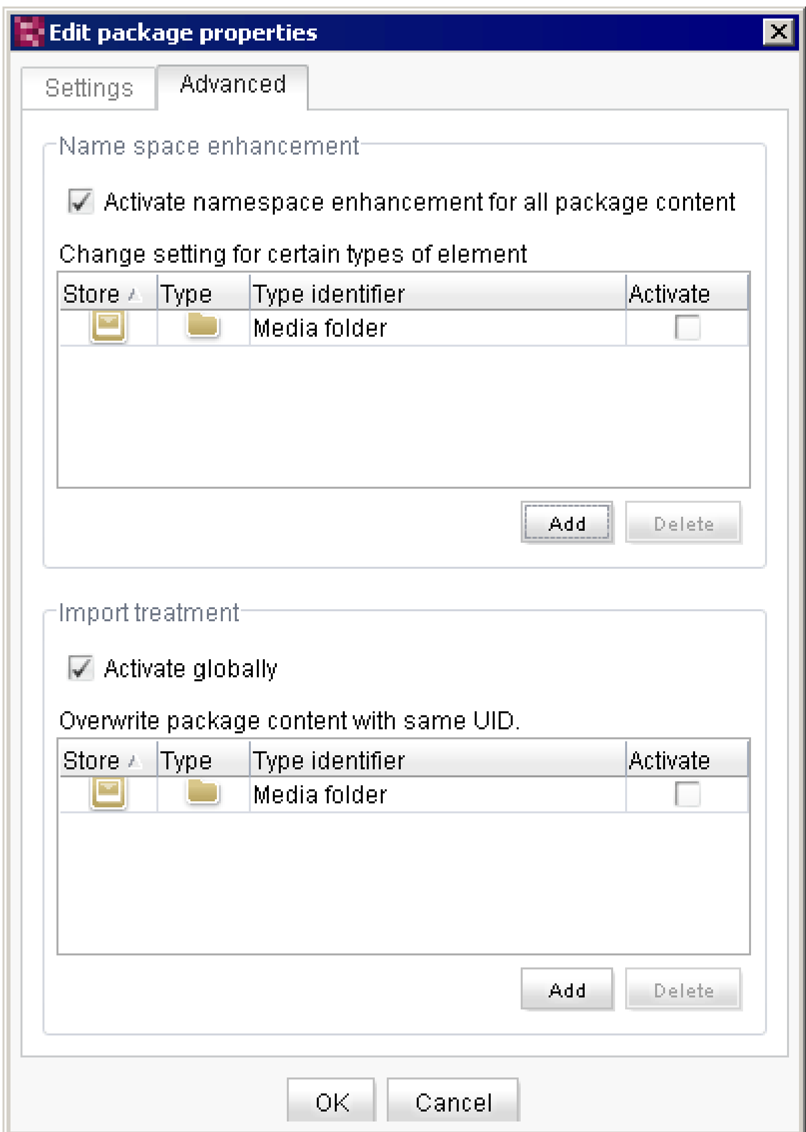


Figure 4-4: Dialog box – Editing advanced package properties

Name space enhancement: Enable or disable the name space enhancement or extension for package content. The extension can either be enabled or disabled globally, or for individual element types only. For a description of the configuration, see Chapter 4.1.6 page 34.

Import treatment: Enable or disable conflict handling on importing the content into a target project. This setting is only useful if the name extension has been disabled for all or for specific element types. For a description of the configuration, see Chapter 4.1.7 page 39.

4.1.3 Define permissions for a package

The editing permissions for the package are set in the permissions pane. A new package is first created by the administrator of the master project, who also assigns the permissions for the package. As soon as the permissions have been defined here, the package properties can be edited by all *qualified* persons ("Qualified users").

Responsible – these are persons responsible for the package in the master project. The persons responsible are notified by e-mail if a new package version is available or a new package version has been imported

Qualified – these persons may edit the package properties (permissions, dependencies, etc.) and make content changes to the packages, e.g. add events or delete start nodes.

Publishers – may publish packages and therefore make them available for importing into the target projects.

This icon opens the "Select user" dialog box for adding users. A person can be chosen from the list of possible people. The entry is selected and the user is added to the required group (Responsible, Qualified, Publishers) by clicking the "OK" button. If no users are explicitly assigned to the groups, only the administrators of the master project are authorized to edit and publish the package.

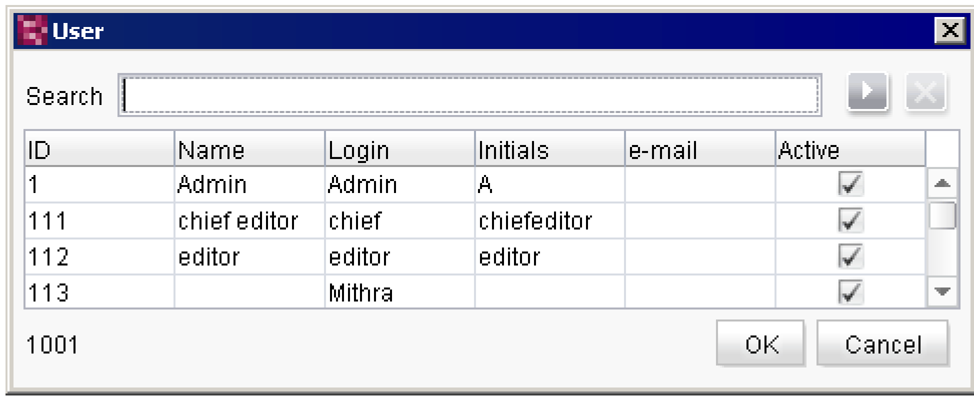


Figure 4-5: Dialog box – User

For example, if you click the icon behind the "Qualified" field and then select a user from the "Users" dialog, the new user is entered as an authorized person for the package.

This icon opens the "User" dialog box. A person can be selected from the list of

already assigned persons. The entry is selected and the user is removed from the required group (Responsible, Authorized, Publishers) by clicking the "OK" button ("Ctrl" or "Shift" can be used for multiple selection).

4.1.4 Changing package types and defining package dependencies

The package type and package dependencies are set in the "Type" pane.

Package type – indicates the package type, selected in the "Select type" dialog box on creating the package (content or template package, see Chapter 4.1.1 page 19).

Depends on – is only active for content packages. Manual dependencies on template packages are defined here. If the content package is subscribed to, the customer must at the same time also subscribe to the corresponding template package given here. Template packages do not have any dependencies. Therefore, this field is disabled for template package type.

As only a template package can be selected here, it is absolutely necessary for all templates (page, section, link templates, etc.), on which pages and sections in the content package are based, are included in this template package. See also Chapter 9.1.2 page 104.

A content package can also be dependent on other content packages (see Chapter 4.2.4 page 44). These content dependencies are not shown here! But they are visible in the version list of a package (see Figure 4-18) and in the detailed information of the package (see Figure 6-6).

This icon can be used to define a dependency of the package content on an existing template package. The icon opens the "Choose Package" dialog box. The list displays all packages, which either exist in the same project (master project) or were subscribed from another project. The required package is marked in the list.

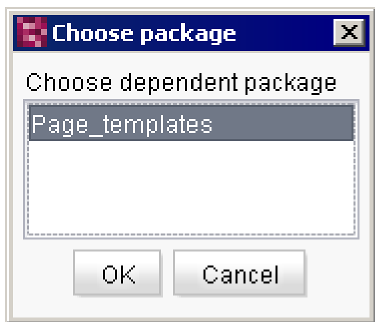


Figure 4-6: Dialog box – Choose dependent package

OK – Assigns the selected template package to the content package and closes the "Choose dependent package" dialog box.

Cancel – Closes the "Choose dependent package" dialog box. Already selected assignments are not transferred.

Removes the dependency once more.

Changeable – if this checkbox is **enabled**, a write permission for the target projects is issued for the imported objects. If the checkbox is **disabled**, the imported objects can be seen and used in the target projects, but not changed.

4.1.5 Configuring events for a package

Configure: The "Configure" button, which appears on initially creating in the "Create package" dialog box or during subsequent editing of a package in the "Edit package properties" dialog box, opens the "Configure events" dialog box.

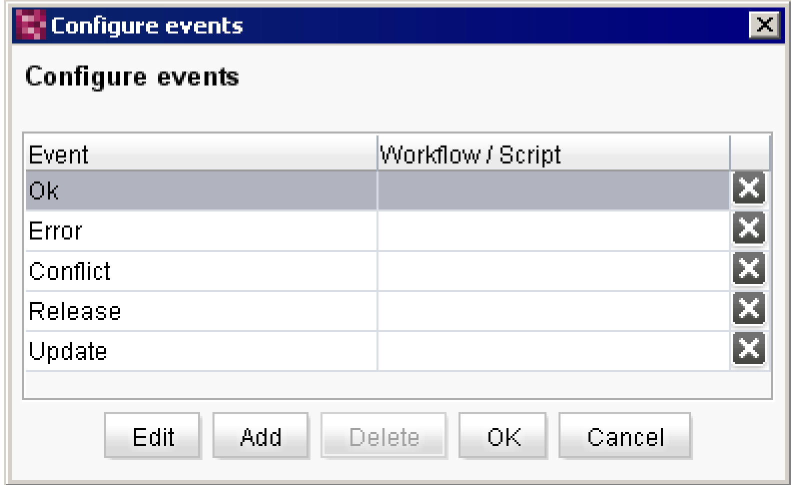


Figure 4-7: Dialog box – Configuring events in the master project

All events defined for the package are listed in the table and the workflows or scripts assigned to the event are displayed. There are two types of events: Standard events (see Figure 4-7) and so-called package-specific events. The **standard events** are predetermined by the system and handle the most common sequences on importing packages. Standard events are:

- **OK**: The assigned workflow is implemented following successful importing of the package version.
- **Error**: The assigned workflow is run if the importing of the package version is faulty.
- **Conflict**: The assigned workflow is started in case of a conflict situation following the import of the package version.
- **Release**: The assigned workflow is run following successful importing of the package version, provided automatic release is not set in the subscription (see Chapter 5.1.2 page 61). For example, all objects contained in the package can be released automatically.
- **Update**: The assigned workflow is implemented following successful importing of the package version. The selected workflow is initiated for all nodes, which has not been newly imported into the project, but has only been changed.

Package-specific events are defined in the master project and automate special package-relevant sequences. One use case is the updating of pages in the Page Store after changing templates. For example, if a page or section template in the project is changed by importing a new template package, *all* pages from the Page Store created to date on this template must be updated to the new state by blocking and unblocking. Simple updating (standard "Update" event) is not sufficient in this case. Double-click a package-specific event to assign a script to the event (see Chapter 4.1.5.3 page 34).

Edit: Click this button (or double-click the table) to assign a standard event selected in the table to a workflow (see Chapter 4.1.5.1 page 31).

Add: Adds a new, package-specific event to the package (see Chapter 4.1.5.2 page 32).

Delete: Deletes a package-specific event from the list. If the list only contains standard events (see Figure 4-7), the button is not active. Standard events cannot be deleted.

OK: Saves the changes and closes the "Configure events" dialog box.

Cancel: Closes the "Configure events" data box. Already made changes are not accepted.

All events configured in the package pool are copied into the target projects with a subscription. In the subscription management, however, it is possible to change the event configuration again for a package. The workflows and scripts, which have been defined for the package within the master project, can be changed again in the target projects using the "Configure" button (see Chapter 5.1.4 page 65). These changes are not visible in the master project and are also not accepted in other target projects.

> *The workflows from the master project can be assigned to a package. However, the first time they are imported the workflows are not known in the target project. In this case the required workflows must be imported into the target project first with the help of a template package. Only then can the events be configured and used in other packages. Scripts are defined on the server-side in the script service and are therefore known in all projects server-wide.*

4.1.5.1 Assign workflows

Existing *standard events* can be assigned to workflows in the "Configure Events" box. For example, the standard "Release" event can be assigned to the "Request release" workflow.

Edit: If a standard event is already selected in the box, the "Choose workflow" box is opened by clicking this button (or by double-clicking the table).

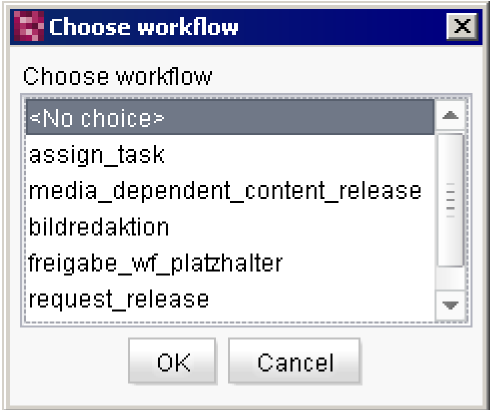


Figure 4-8: Dialog box – Choose workflow

All known workflows from the Template Store of the master project appear in the list. The required workflow is chosen from this list.

OK: Click this button or double-click the workflow to assign the selected workflow to the event.

Cancel: The dialog is cancelled, no (new) workflow is assigned to the event.

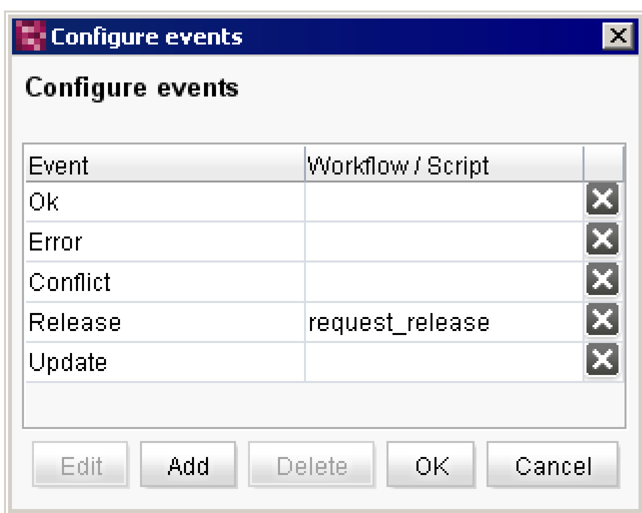


Figure 4-9: Dialog box – Assign workflow (standard event)

If the "Release" event (see Chapter 4.1.5 page 28) occurs now, after the package has been imported into the target project, the assigned "Request release" workflow is started automatically.

To delete an existing assignment, the button directly behind the workflow to be deleted is clicked. The "Delete" button only removes project-specific events, not workflows.

4.1.5.2 Add new event

Apart from the existing standard events, additional *package-specific events* can also be added to a package. Scripts are assigned to these package-specific events. If the event assigned to the package is also activated on generating a package version (see Chapter 4.2.6 page 48), the script is automatically run for all target projects with a valid subscription on importing. Such a project-specific event can be used, for example, to update a form field in all corresponding pages of the target project on importing. If a form field, for example, a combobox, is to be imported into a target project with a new entry and a preset value, a package-specific event is used to start a script immediately after importing the form field, which loads and saves all corresponding pages once. In this way, all occurrences of the form in the target project are automatically updated.

Add: In the "Configure events" dialog box, the "Add package-specific event" dialog box is opened by clicking the button.

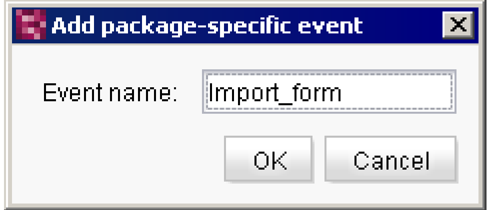


Figure 4-10: Dialog box – Add package-specific event

Event name – the name of the new event must be unique in the entire package. If the field is left empty or a name is chosen that already exists in the list, the new event cannot be created, the lettering remains red and the button is disabled.

OK: Click this button to create the new event.

Cancel: Click this button to end the dialog; a new event is not created.

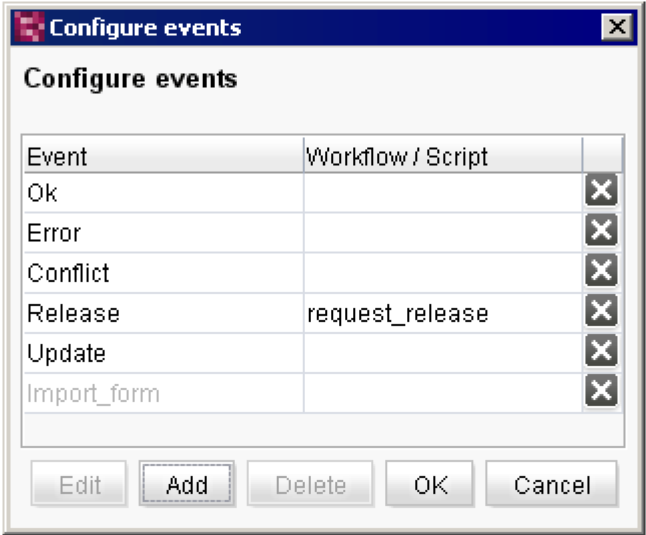


Figure 4-11: Dialog box – Configure package-specific event

If a valid event name is entered and the input is confirmed with the **OK** button, the new package-specific event "Import_form" appears in the "Configure events" box. The event is only available in the package in which it was generated, while the standard events are available in all packages. To usefully use an event in the target projects, a script must be assigned to the event (see Chapter 4.1.5.3. page 34).

4.1.5.3 Assign scripts

> *The selection of scripts as user-specific events is currently not supported in FirstSpirit Version 4.0.*

4.1.6 Deactivate namespace extension (V4.1 and higher)

When creating packages, there must be no overlapping between package content, i.e. each project node may belong to precisely one package only. The so-called "name space enhancement" was introduced for package objects so that the affiliation of an object to a package is unique and is as transparent as possible for the package developer. This is done by appending an "@" and the package name (see Chapter 4.1.2.1 page 22) to the reference names of the objects of a package ("ObjectName@PackageName") (see Chapter 4.2.7 page 49).

> *The reference names with name space enhancement can be shown in the tree structure using the "Display reference names in tree" option in the "Extras" menu / "Preferred display language".*

After adding a package, all objects are assigned this namespace extension. All objects in the project that use the "old" reference names must then be changed, this means that the old reference name must be replaced everywhere with the new reference name (with "@PackageName"). Some of these changes have to be made manually (see Chapter 9.1.6 page 108 to Chapter 9.1.8 page 111).

The namespace extension is problematic for package content with identical reference names in the master and target project. This primarily concerns default format templates ("Bold", "Italic", etc.), which exist in each FirstSpirit project and are grouped together in one folder in the Template Store under the "Format templates" node. They are used for text formatting and, e.g. are used in the DOM Editor and DOM Table input components in the Page Store (*see also FirstSpirit Manual for Developers (Basics))*. Due to the name extension, the assignment to the corresponding buttons (e.g. "Bold") is lost within these input components. Here the namespace extension can result in errors, not only in the master project but also in the target project (see Chapter 9.1.7).

In FirstSpirit Version 4.1 and higher the template developer can disable the namespace extension for the default format templates as well as for other objects

that have the same reference names in the master and in the target project.

To do this, the package properties have to be opened first and the settings for the package must be adjusted within the "Advanced" tab (see Chapter 4.2.2 page 43):

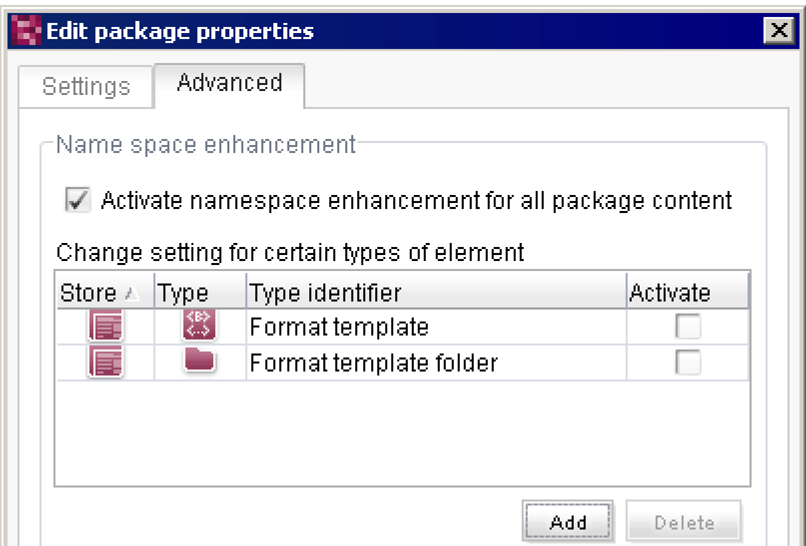


Figure 4-12: Enable / disable namespace extension

Activate namespace enhancement for all package content: If this checkbox is **enabled**, the namespace extension is switched on for all package content. This is the recommended default setting for reasons of uniqueness of reference names in the project and increased transparency. This means: If an object is added to the package, the extension "@PackageName" is assigned to the reference name (see Figure 5-5: Namespace extension for package content). Potential references then have to be adjusted to the added object within the project (see Chapter 9.1.7 ff. page 109).
If this checkbox is **disabled**, the namespace extension is switched off for all package content. If an object is added to the package, the reference name remains unchanged (i.e. the reference name is not assigned an extension by "@PackageName"). The conflict handling for importing the package content into the target project (see Chapter 4.1.7 page 39) can be used in this case to define whether package content of the master project overwrites existing objects in the target project or is to be created in the target project under another name.

Change setting for certain types of element: As explained above using the example of default format templates, in most cases enabling and disabling the namespace extension is only required for specific element types. The global setting for the package content can therefore be limited to specific element types. In this case, the "Add" button can be used to add the required element types to the "Change setting for certain types of element" table. The default setting for copying

into the table is always the opposite of the global settings, which were defined using the "Activate namespace enhancement for all package content" checkbox (see Chapter 4.1.6.1 page 36).

Store: Display of the Stores as an icon (analogous to the tree display in the FirstSpirit JavaClient). The column can be sorted.

Type: Display of the element type as an icon (analogous to the tree display in the FirstSpirit JavaClient). The column can be sorted.

Type identifier: Name of the element type. The column can be sorted.

Activate: The namespace extension for the respective element types can be switched on or off by enabling or disabling the checkbox. If the checkbox is **enabled**, the namespace extension is switched on for the selected element type. If an object of the selected type (e.g. a format template) is added to the package, the extension "@PackageName" is assigned to the reference name (see Figure 5-5: Namespace extension for package content). Potential references then have to be adjusted to the added object within the project (see Chapter 9.1.7 ff. page 109).
If the checkbox is **disabled**, the namespace extension is switched off for the selected element type. If an object of the selected type (e.g. a format template) is added to the package, the reference name remains unchanged (i.e. the reference name is not assigned an extension by "@PackageName"). The conflict resolution for importing the package content into the target project (see Chapter 4.1.7 page 39) can be used in this case to define whether or not package content of the master project is to overwrite existing objects in the target project.

Add: Click this button to add the required element types to the "Change setting for certain types of element" table (see Chapter 4.1.6.1 page 36).

Delete: Click the button to remove a selected element type from the list again (see Figure 4-12). After this element type has been removed, the global settings for the namespace extension apply to it once again.

4.1.6.1 Add new element types (V4.1 and higher)

Add: Click the button to open the "Element selection" dialog:

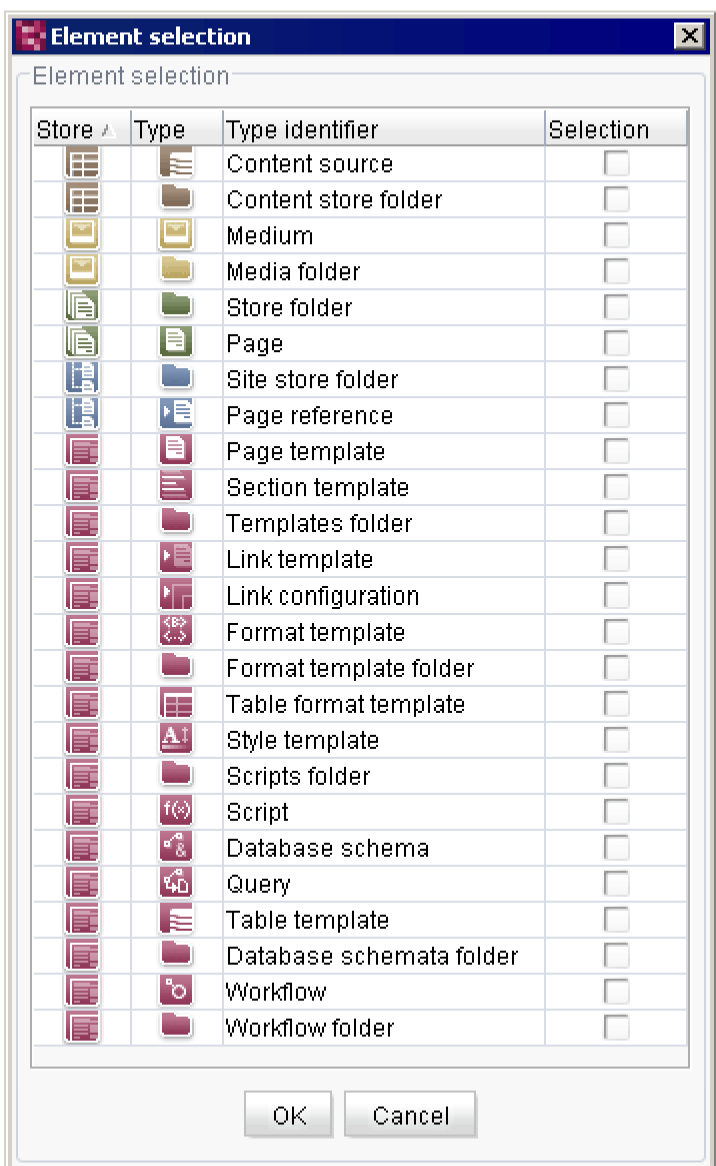


Figure 4-13: Element selection for the namespace extension

For a description of the Store, Type and Type identifier columns, see Chapter 4.1.6 page 34.

Selection: By activating the checkbox, the selected elements are copied into the "Change setting for certain types of element" table. Naturally, only the element types that are to be included in the subsequent package have to be selected. Therefore, in content packages, e.g. it is not necessary to select any element types from the Template or Content Store (pink colored or brown icons).

The default setting for copying into the table is always the opposite of the global settings, which were defined using the "Activate namespace enhancement for all

package content" checkbox.

Therefore, if the namespace extension for the package content is globally disabled, the namespace extension is enabled directly on copying the selected element types.

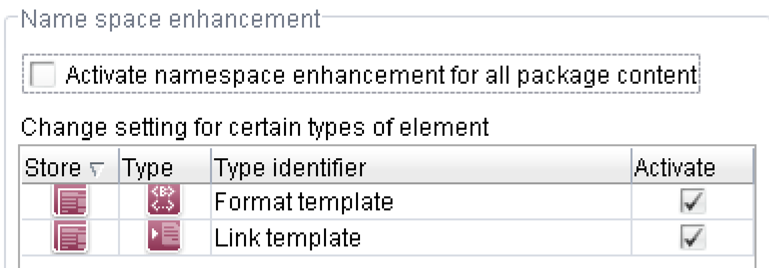


Figure 4-14: Default setting for globally disabled namespace extension

If, on the other hand, the namespace extension is globally enabled, the namespace extension is disabled directly on copying the selected element types.

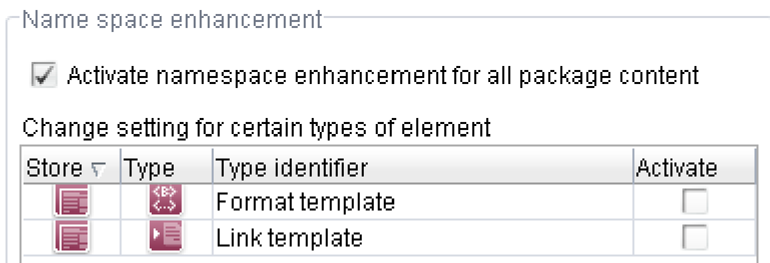


Figure 4-15: Default setting for globally enabled namespace extension

4.1.7 Changing conflict resolution on importing (V4.1 and higher)

If name extension is disabled (see Chapter 4.1.6 page 34), when the package content is imported, the target project may already contain identical reference names (e.g. the default format template "bold" with the reference name "b").

In this case the new imported objects (e.g. the format template) would be assigned the postfix "_1" in the target project, i.e. for example "b_1". To use the template format in the target project, either all references to this format template would have to be adjusted manually in the target project, or the format template originally in the target project would have to be removed or renamed and the postfix "_1" in the new imported format template would have to be deleted again.

The PackagePool has been enhanced to include a further function, "Import treatment", to prevent this behavior. Here the template developer can enable the overwriting for all objects with the same name in the target project or for a certain type of objects with the same name (e.g. format templates) (see Figure 4-16). This means that manual adjustment in the target project is no longer necessary.

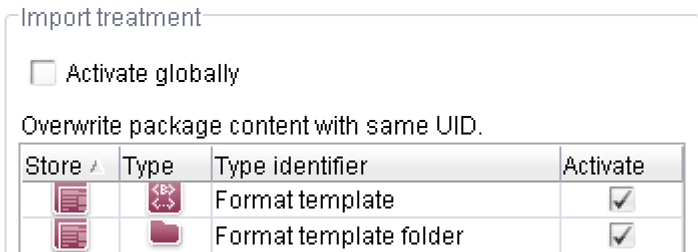


Figure 4-16: Import treatment – Overwriting package content

Activate globally: If the checkbox is **disabled**, overwriting the content in the target project is prevented by package content with the same name (default setting). In this case the conventional conflict resolution takes effect, which is also used on creating objects of the same name within a package: If a reference name (Uid) is entered which has already been assigned within a namespace, FirstSpirit automatically replaces the name with a unique name, mostly by appending a number. In this case the package content is created in the target project under another name.
If the checkbox is **enabled**, the content in the target project with the same name is overwritten by the package content from the master project on importing. Therefore, for example, if the package contains a format template with the unique name "b", on importing, a format template with the same name in the target project is overwritten by the format template with the same name from the master project.

Overwrite package content with same UID: In most cases, the overwriting of

content with the same name in the target project is only required for specific element types. The global setting for the package content can therefore be changed for specific element types. In this case, the "Add" button can be used to add the required element types to the "Overwrite package content with same UID" table. The default setting for copying into the table is always the opposite of the global settings, were using the "Enable globally" checkbox (analogous to the namespace extension, see Chapter 4.1.6.1 page 36).

For a description of the Store, Type and Type identifier columns, see Chapter 4.1.6 page 34.

Activate: The default settings for handling imports can be changed by enabling or disabling the checkbox. If the checkbox is **enabled**, overwriting objects with the same name in the target object is switched on for the selected element type. In this case, existing content in the target project can be overwritten.
If the checkbox is **disabled**, overwriting objects with the same name in the target object is prevented for the selected element type. If an object of the selected type (e.g. a format template) and with the same name already exists in the target project, the object in the target project is retained and the new package content is imported into the target project under a different name. In this case it may be necessary to make adjustments in the target project.

4.2 Edit packages

To edit an existing package, the submenu item "Edit packages" is opened. Only persons defined as "Qualified" for the package by the project administrators of the master project are authorized to edit a package (see Chapter 4.1.3 page 25).

4.2.1 Package list

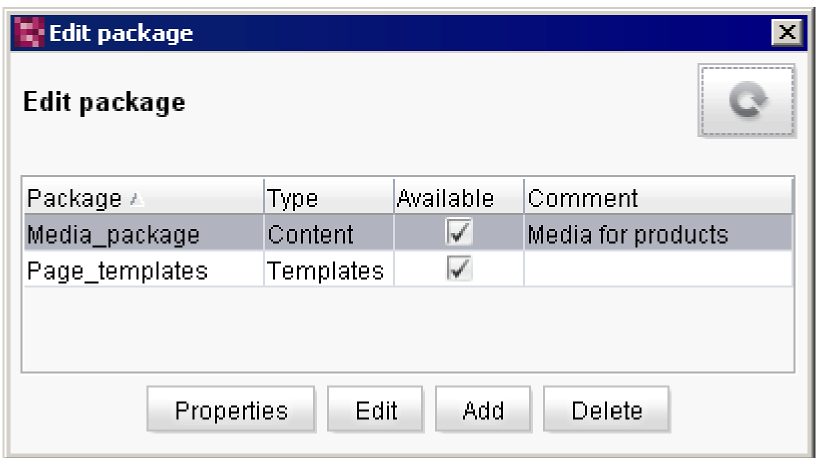


Figure 4-17: Dialog box – Edit package

The "Edit Package" menu item opens the "Edit Package" dialog box. All packages available in the master project are displayed in this dialog box. The table provides the following information for each package:

Package – unique package name.

Type – package type, indicates whether the package is a content package or a template package.

Available – if this checkbox is **enabled**, the package is available for the target projects and be can subscribed to. The subscription can be crated, even if no package version exists yet for a package. If the checkbox is **disabled**, the package is not available for subscription in the target projects.

Comment – optional comment on the package.

If a table entry is selected in the list, and therefore a package selected for editing, various editing options are provided by the buttons displayed in the bottom area of the window.

Properties: Click this button to open the "Edit project properties" dialog box. The box corresponds to the "Create package" dialog box (Chapter 4.1.2 page 20), which is opened on creating a package, with the difference that there the properties are already filled.

Edit: Click this button (or double-click the table row) to open the "Edit package 'xyz'“ dialog box. In this dialog box, new package versions can be created, the availability of the package for the target projects can be edited and package content can be added or removed (see Chapter 4.2.3 page 43 ff.).

Add: Click this button to create a new package and add it to the list. First, the familiar "Choose package type" dialog box opens, the remaining procedure is analogous to the "Create packages" menu item (see Chapter 4.1 page 19).

Delete: this button can be used to delete packages from the table. A confirmation prompt is displayed before finally deleting, to ensure that a package is not deleted accidentally.

> *If a package is deleted, all other versions of the package are also removed! It is therefore not possible to delete packages directly, for which subscriptions have already been taken out. In this case the following confirmation prompt is displayed first:*

"Could not delete package. There are subscriptions existing which subscribes this package.
[Subscription (Package: 675376 - Project: 38478)]
Delete subscriptions too?"

OK: If this button is clicked, all existing subscriptions to the package are deleted first and then the package is deleted.

If a dependency to a template package still exists (Chapter 4.1.4 page 27), the following message is displayed beforehand:

"Could not delete package.
There is a dependency to the template package 'Vorlagen (163247)' existing."

In this case the link to the template package (here "Vorlagen (163247)") has to be removed first in the properties of the content package. Only then can the content package be deleted.

If the namespace extension is enabled (Chapter 4.1.2.2 page 24), the extended reference names ("ObjectName@PackageName") continue to exist after a package has been deleted and are not reset to the original reference names.

Cancel: Click the button to cancel the action. Neither the package nor the corresponding subscription is deleted.

4.2.2 Edit package properties

Properties: The button opens the "Edit package properties" dialog box, in which all properties of the package can be edited. Among other things, the package availability can be changed in the package properties and the events are modified when imported. In addition, the permissions for the package are defined there, the package type is specified and dependencies on existing template packages are defined. The "Edit package properties" dialog box was described in Chapter 4.1.2 page 20.

4.2.3 Edit package version

Edit: This button opens the "Edit 'Package Name' package" dialog box.

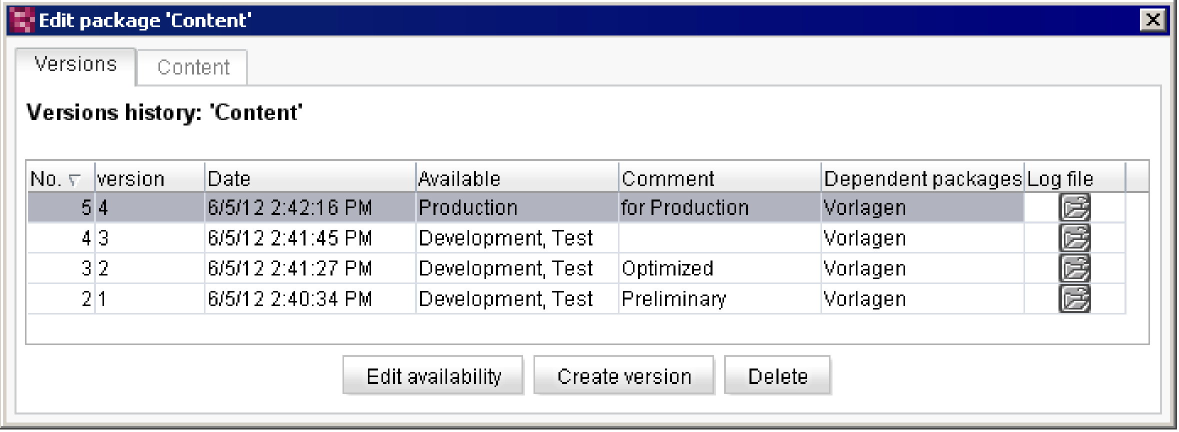


Figure 4-18: Dialog box – Version history 'PackageName' (Edit package)

The dialog box is divided into the "Versions" tab and the "Content" tab (see Chapter 4.2.7 page 49).

The version history is shown in the "Versions" tab, that is to say, all previous package versions of the selected package are listed here with the following

information:

No. – unique version number automatically assigned when a new package version is created.

version – the manually assigned version name assigned by the creator of the package.

Date – date and time when the package version was created.

Available – shows the publication groups, for which the package version is available.

Comment – optional comment on the package version.

Dependent packages – shows the dependent packages (template and content packages, see also Chapter 2.1.2 page 7) of the respective package version.

If a table entry in the list is marked and therefore a package version selected, the package version's availability to the individual publication groups can be changed.

Edit availability: This button opens the Edit Package Version dialog box (see Figure 4-19).

In addition, it is also possible to generate a new package version here (see Chapter 4.2.4 page 44).

4.2.4 Generate package version

> *If the package content has been changed, a new package version should always be generated. The changes are then available in the subscribing target projects (see also Chapter 2.1.3 page 10).*

Create version: This button opens the "Create package version" dialog box.

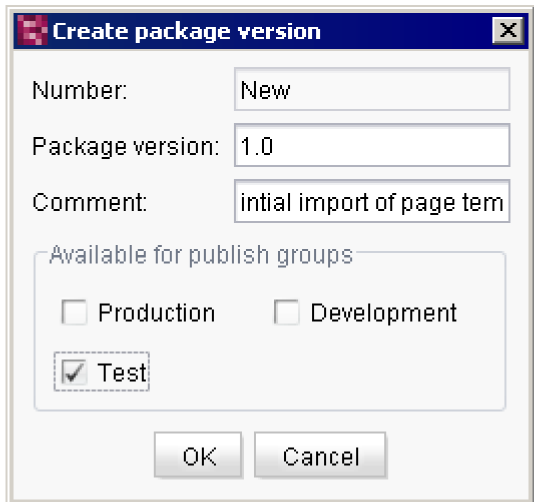


Figure 4-19: Dialog box – Create package version

Number – instead of a unique version number, the entry "New" is displayed here. The version number is assigned automatically by the system when a new package version is created (field is inactive). As there are not yet any new package versions at this time, a number cannot be displayed yet.

Package version – in addition to the version number assigned by the system, a "descriptive" (more informative) version number can be assigned here.

Comment – optional comment on the new package version.

Available for publish groups – all available publication groups (see Chapter 7 page 86) are displayed here as a checkbox. The availability of the package version to the respective publication group is changed by enabling or disabling a checkbox. If the checkbox is **enabled**, the package version is available for importing to this publication group. If the checkbox is **disabled**, the package version is not available for this publication group. A package version can be available for several publication groups, subscriptions on the other hand are always taken out for precisely one publication group (see Chapter 5.1.2 page 61). For example, if a package version is available for the "Test" and "Production" publication groups, not only a subscription for the "Test" publication group but also a subscriptions for the "Production" publication group can access the package version.

Activate custom events – this window pane is only display if package-specific events have been configured for the package. For further explanations, see Chapter 4.2.6 page 48.

OK: After clicking this button, the message "Creating version on the server." is displayed, after confirming this dialog box a new package version is created.

Depending on the package size, this can take quite a while.

Cancel: Click the button to cancel the action. No new package version is created.

If the new package version was successfully generated, the following information appears:

"Version successful created.
Dependent packages: 'Vorlagen'"

If dependencies on other packages exist, they are pointed out to the user (here dependency on the "Vorlagen" package). Not only dependencies on content packages are displayed but also dependencies on template packages.

Dependencies on content packages are created if objects are referenced within a package, e.g. a media file, which is already part of an existing package (e.g. the 'Vorlagen' content package). As objects may only be contained in one package, in this example the medium cannot be added to the package. For this reason, the package's content dependency on another content package is displayed.

*However, a content dependency is only established, if it referenced objects are involved (e.g. media). If objects are to be copied into the content package, of which child elements already exist in an existing package, a corresponding error message is displayed when the package content is selected ("Could not add the element to package 'Inhalte'.
Found existing child 'p_11@Content ID(275338)' which belongs to package 'Content'.", see Chapter 4.2.7 page 49).*

This content package can also be subscribed to. Unlike the dependent template package, it is not absolutely necessary to subscribe to a dependent content package.

When importing a dependent content package into the target project, the import order must be noted later: Firstly, the dependent content package is imported and then the package, which contains the reference to the dependent package. If the dependent content package is not subscribed to or is, but in the wrong order, it can cause errors in the target project (for a strategy for debugging in the master project see Chapter 4.3 page 56 and for debugging in the target project see Chapter 5.3 page 69).

> *When creating a package version, there must be no difference between the current and the released state of the content integrated in the package. If the package contains objects, which are not yet released at this time, the following error message appears when an attempt is made to generate a new package version.*

"Could not create version zip file.
Found store elements which are not released:
'mpg_datei@Content' (ID=275347)"

Only when all unreleased objects have been released a new package version can be generated!

4.2.5 Edit package availability

A package can be made available for different publication groups with different package versions for importing into the target projects. This project availability can also be subsequently changed, for example, following development and testing, by activating a package version for the "Production” publication group too.

Edit availability: Click this button or double-click the required package version to open the "Edit package version" dialog box (see also Figure 4-19).

The following information on the selected package version is displayed:

Number – unique version number. The field is inactive and cannot be edited.

Package version – manually assigned version name. The field is inactive and cannot be edited.

Comment – optional comment. Here an existing comment can be changed or a new comment can be added.

Available for publish groups – all available publication groups are displayed here as checkboxes. The availability of the package version for the edited publication group is changed by enabling or disabling a checkbox. If the checkbox is enabled, the package version is available for import. If the checkbox is disabled, the package version is not available for this publication group.

OK: this button is used to accept the changes for the existing package version.

Cancel: with this button, the process is cancelled and any changes already entered are not accepted.

4.2.6 Activate specific events

Configure – this button can be used to assign events to a package, which run a script when imported into the target projects (see Chapter 4.1.5.3 page 34). Unlike the workflows of the standard events, configuring alone is not sufficient for package-specific events. On creating a new package version the event must be explicitly activated; only then is the script run on importing into the target projects. Under this precondition it is possible to activate different events/scripts for different package versions.

The selection of scripts as user-specific events is currently not supported in FirstSpirit Version 4.0.

If a package-specific event has been configured in a package, the familiar "Create package version" dialog box (see Figure 4-19) is supplemented with the "Activate custom events" pane.

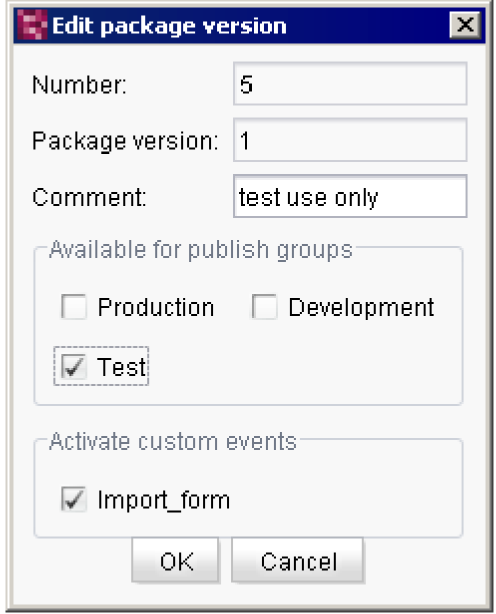


Figure 4-20: Dialog box – activate custom events

In this section the respective checkbox is used to activate the required event.

When the package version for subscriptions with publication group "Test" is imported, the script assigned to the "Import_form" event is started.

4.2.7 Edit package content

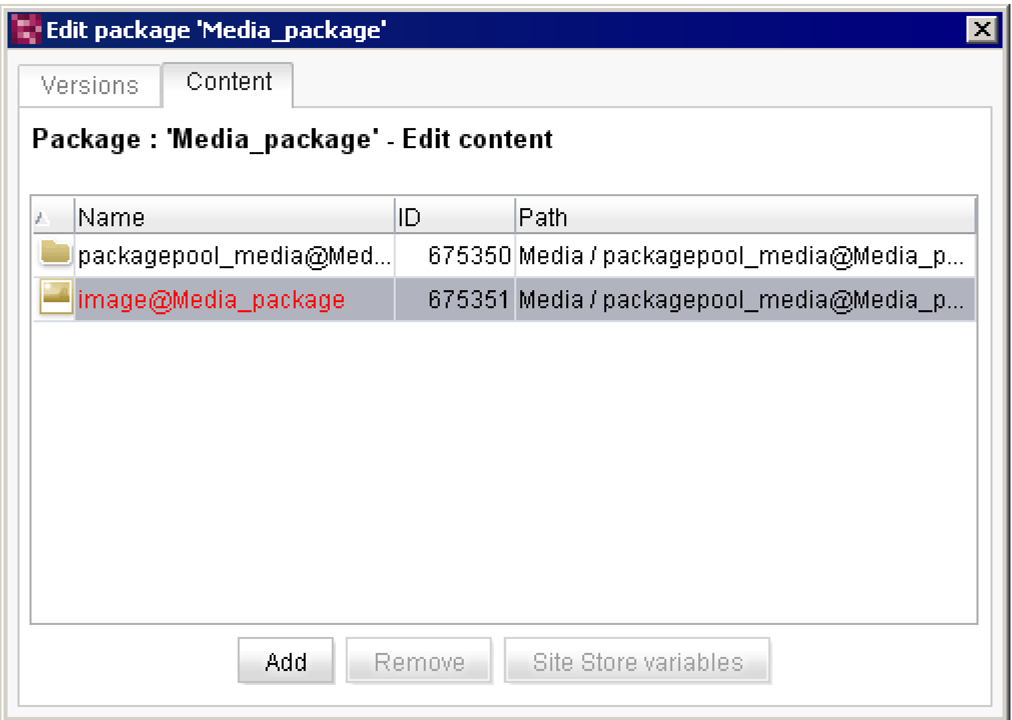


Figure 4-21: Dialog box – Edit package content (Edit package)

The "Content" tab displays the global package content, not the content of a package version. Regardless of whether an older or the latest package version is selected, the content currently contained in the package is always displayed, not the content in the package version.

Icon – the left-hand columns shows the Store from which the package content was added. In the example, Figure 4-21, a "People" folder (incl. media object) was copied from the Media Store into the package (yellow icon color).

Name – unique name of the object in the package. Red marking indicates that the objects are not released.

These have to be released first to generate a version, otherwise an error message like "Could not create version zip file.
Found store elements which are not released:
'mpg_datei@Content' (ID=275347)" (see Chapter 4.2.7 page 49) is displayed.

ID – ID of the object from the master project

Path – path to the object in the project tree of the master project

Delete – this button removes the selected object from the package. If no package content is selected, the button is not active.

Site Store variables – this button is only active if the package contains objects from the Site Store and these objects contained defined or inherited structure variables (see Chapter 4.2.8 page 53).

Add – this button opens the "Add content node" or "Add template node" dialog box, depending on whether the package is a content or a template package.

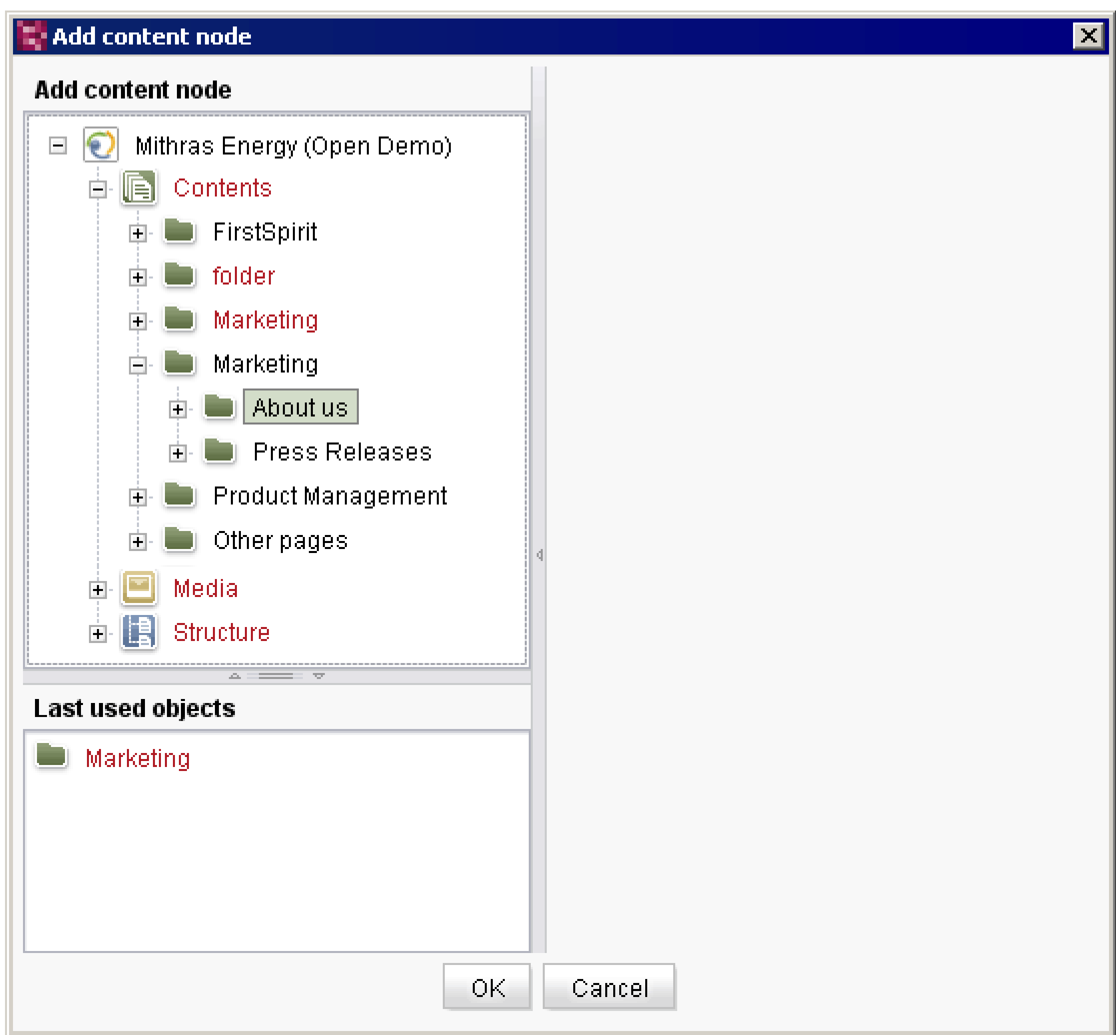


Figure 4-22: Dialog box – Add content node

The project tree of the master project is shown in the "Add content node" box. Only the Stores allowed for this package type are displayed. Therefore, for a content package, only the Page, Media and Site Store are displayed (see Figure 4-22), for a Template Store, only the Content, Media and Template Stores are displayed. The required start nodes or individual objects can now be selected in the view. Multiple selection is possible with the "Ctrl" key pressed.

If the "Show symbols (Metadata, Packages, Permissions)" is enabled in the "Extras" menu item, a corresponding symbol is displayed in the project tree, behind the objects integrated in a package. In the "Classic" Look & Feel this is a blue dot, in the "LightGray" Look & Feel it is a package symbol. In this way it is possible to quickly see which objects are already integrated in a package. These objects and child elements can no longer be added to another package. Otherwise the following error message is output on confirming with "OK":

"Could not add the element to package 'Inhalte'.
Found existing child 'p_11@Content ID(275338)' which belongs to package 'Content'."

In the bottom left-hand area of the dialog, below the "Last used objects", the most recently added objects are available for renewed selection. If a preview graphic has been stored for a selected object in the project, this is displayed in the right-hand area of the dialog. Thumbnails are displayed for Media Store objects.

OK: Click to add the selected objects to the package.

More than the explicitly sought content can be copied into the package. Dependencies between content are automatically recognized by FirstSpirit and are added to the package (see Chapter 2.1.2 page 7). Dependencies between the Stores are resolved in the following order:
*1. firstly, dependent objects of explicitly added objects from the **Site Store** are added to the package,*
*2. then dependent objects of explicitly added objects from the **Page Store** are added to the package and*
*3. finally, dependent objects from the **Media Store** are added to the package.*

For example, if a page reference from the Site Store is added, the corresponding page from the Page Store is also added and any referenced media are copied into the package- On the other hand, if only one page is explicitly added to the package, only referenced media are also copied, however, not a page reference from the Site Store.
Important: Folder structures are also not automatically copied.

*Folder structures are only imported into the target projects if the required folders from the target project are also added to the package content, before referenced objects are automatically added to the package content. If the folder structures from the master project are to be retained in the target project, a specific **order** must be kept **when adding** content:*
1. Add objects from the Media Store,
2. Add objects from the Page Store,
3. Add objects from the Site Store.

If the “Show symbols (Metadata, Packages, Permissions)" is enabled in the "Extras" menu item, a corresponding symbol is now displayed in the project tree, behind the objects integrated in a package.

If the namespace extension is activated (see Chapter 4.1.6 page 34 and Chapter 4.1.7 page 39), when an object is added to a package the reference name of the object is also changed. The old reference name is replaced by "ObjectName@PackageName" as part of the **namespace extension** (see Figure 5-5: Namespace extension for package content). It is therefore visible at first glance, which content has already been integrated in a package and in which package it is located. All pages, which have a reference to the changed object then have to be adjusted. This process can take some time to complete.

The extensions are not visible until the view is updated!

The namespace extension only applies to the reference name of an object.

Cancel: the process is cancelled and the "Add content node" box is closed.

4.2.8 Integrate structure variables

Structure variables can be used to configure the properties of the Site Store. One possible use for structure variables is the color coding for menu levels, with which each menu level is displayed with a different background color. The values for the background colors are saved in structure variables and have to be referenced and evaluated within a template in the project, in order to bring about an effect.

The Package Pool supports working with structure variables. Structure variables can be integrated in a package by editing the package content (see Chapter 4.2.7 page 49). However, the button in the "Edit package – Edit content" dialog box is only active for content nodes from the Site Store.

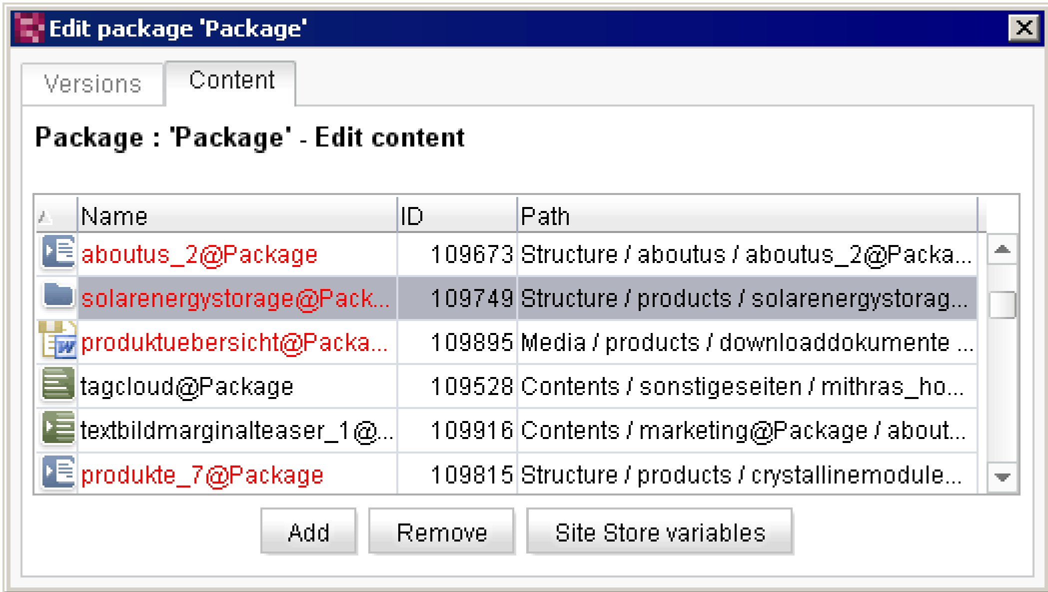


Figure 4-23: Dialog box – Edit structure variables (Edit package)

A list of the structure variables to be copied can be defined for each content node from the Site Store that is in the package:

Site Store variables – click the button or double-click a structure node to open the "Selection of Site Store variables" dialog box.

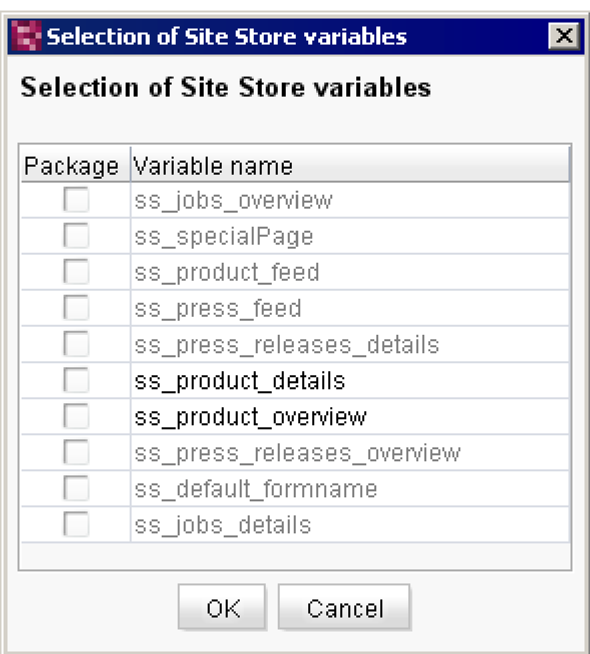


Figure 4-24: Dialog box – Selection of Site Store variables

All structure variables defined on this node or inherited structure variables are displayed in the dialog box. The required structure variables can be copied into the package by enabling the corresponding checkbox in the "Package" column. The inheritance hierarchy applies within the package content. The structure variables selected in a higher-level folder are also copied for all folders below it. The structure variables therefore do not have to be selected for each individual structure node.

OK: the selected structure variables are copied into the package where they are therefore available for importing into the target projects.

Cancel: the process is cancelled and the box is closed.

4.3 Publish packages

This function is used to update package content in target projects using the so-called "push" method (see also Chapter 2.3.1 page 13). The "Publish packages" menu item opens the "Publish package" dialog box, which lists all available packages in a table.

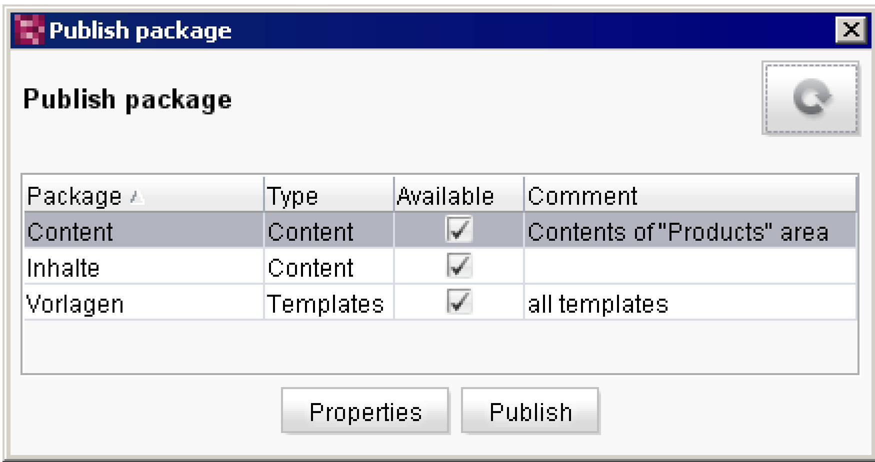


Figure 4-25: Dialog box – Publish package

Users are familiar with this box from the "Edit package" menu item. However, only two buttons are displayed in the bottom part of the window, with which the package selected in the table can be edited.

Properties: Click the button to open the "Package Properties" dialog box. The box corresponds to the "Edit package properties" dialog box (see Chapter 4.1.2 page 20), with the difference that the predefined package properties cannot be changed here; they merely have an informative character. If changes are to be made to the package properties before publishing a package, this is done in the "Edit package" menu item.

Publish – this button opens the "Publish 'package name' package" dialog box (also possible by double-clicking the table row).

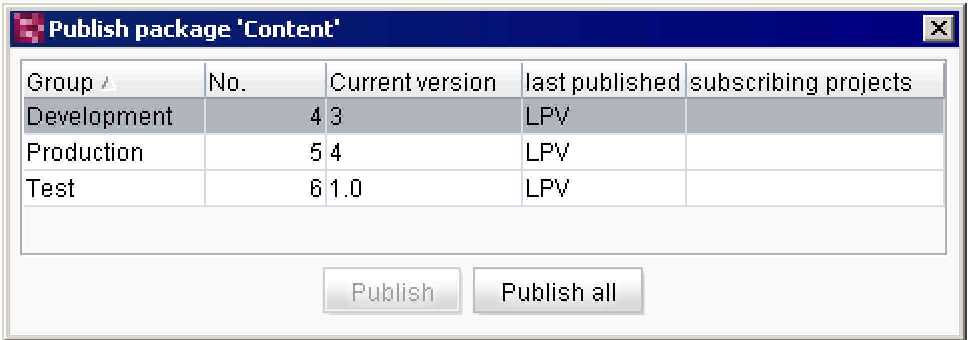


Figure 4-26: Dialog box – Publish package

There only the respective most up-to-date package versions for the known publication groups are listed in a tabular overview.

Group – publication group, for which the package version was marked as being "available".

No. – unique package version number, automatically assigned by the system.

Current version – manually assigned version name.

Last published – shows the last published version.

Subscribing projects – shows all projects, which have subscribed to a valid, active subscription for this package version and this publication group.

The subsequent publishing of a package version is done using the buttons in the bottom part of the dialog box. Packages can only be published if:

- the editor has publication permissions for the package.
- an active subscription exists for the package version and the publication group.

Publish: if the required package version is selected in the table, it can be published by clicking the button. In all target projects, which have taken out a valid, active subscription, with automatic updating, to this package version and the given publication group, importing of the content from the master project starts straight away.

Message: "Update process of underlying subscriptions started on the server.

If one of the conditions named above is not fulfilled, the button is inactive and publishing is not possible.

Publish all: optionally, *all* package versions displayed in the window can be published together. The button is always active, but only package versions that fulfill all the conditions named above are published.

> *Before publishing, the package dependencies should always be determined first (see Chapter 2.1.2 page 7). Dependencies on template packages are defined in the package properties (see Figure 4-6: Dialog box – Choose dependent package). These dependencies are automatically checked. If the dependent template packages are not published, or not in the right order, the publishing is cancelled and an error message is displayed.*

Optional dependencies on other content packages are displayed in the "Detail Info" dialog box (see Figure 6-6: Dialog box – Detail information on a package), which can be opened via the package overview. These dependencies are not checked automatically on publishing. If the dependent content packages are not published, or not in the right order (1. Import dependent content package, 2. Import package containing the references to the dependent package), they can cause errors in the target project: for example, on publishing page references, if the referenced page and the page reference are located in different packages. If, in this example, the package with the page reference is published first and then the package with the referenced page, an error is caused in the target project. To correct the error, the page reference in the master project must be locked to prevent editing and immediately unlocked again (Edit mode on, edit mode off). A new package version (of the package with the page reference) is then generated and re-published, this time in the right order.

5 Subscription menu item (target project)

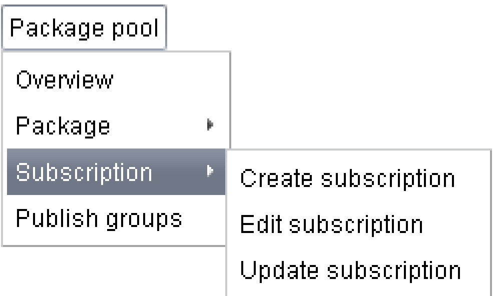


Figure 5-1: Subscription menu item

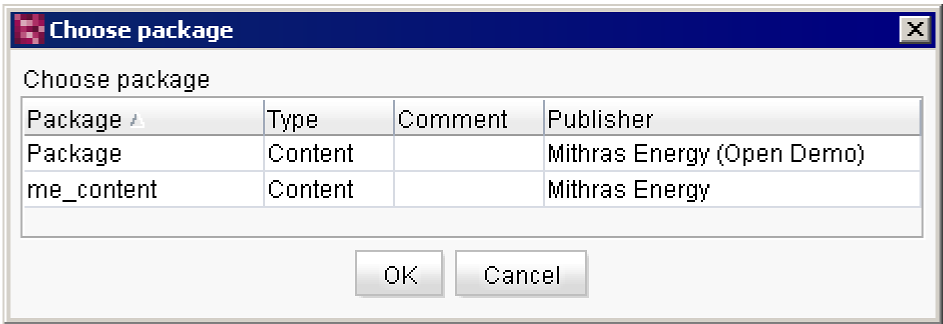
The Subscription menu item is only relevant for the target projects into which available packages can be imported. This is where all settings for importing packages are defined. New subscriptions can be created and existing subscriptions can be edited. In addition, it is possible to trigger a manual package update from the target project. The Subscription Management menu item contains three submenu items:

- Create subscription (see Chapter 5.1 page 59).
- Edit subscription (see Chapter 5.1.6 page 66).
- Update subscription (see Chapter 5.3 page 69).

5.1 Create new subscriptions

To create a new subscription, the "Create Subscription" submenu item is opened. Creating a new subscription involves several steps, which are explained in the following. The initial creation of a subscription can only be carried out by the administrator of the target project.

5.1.1 Choose package



Package	Type	Comment	Publisher
Package	Content		Mithras Energy (Open Demo)
me_content	Content		Mithras Energy

Figure 5-2: Dialog box – Choose package

The "Create subscription" menu item opens the "Choose package" dialog box. All packages available on the server are displayed in this dialog box. Only one package can ever be selected. The table provides the following information for each package:

Package – unique package name.

Type – package type, indicates whether the package is a content package or a template package.

Comment – optional comment on the package.

Publisher – shows the name of the master project, in which the package was crated.

Cancel: Click this button to close the box. The dialog box opens Figure 5-8.

OK: click this button to open the next dialog box: "Create subscription for the ''PackageName' package" (Chapter 5.1.2 page 61).

> **!** *If no packages are available to subscribe to, no new subscription can be created. A dialog box with an error message appears:*

"There are no packages which could be subscribed."

5.1.2 Create subscription for a package

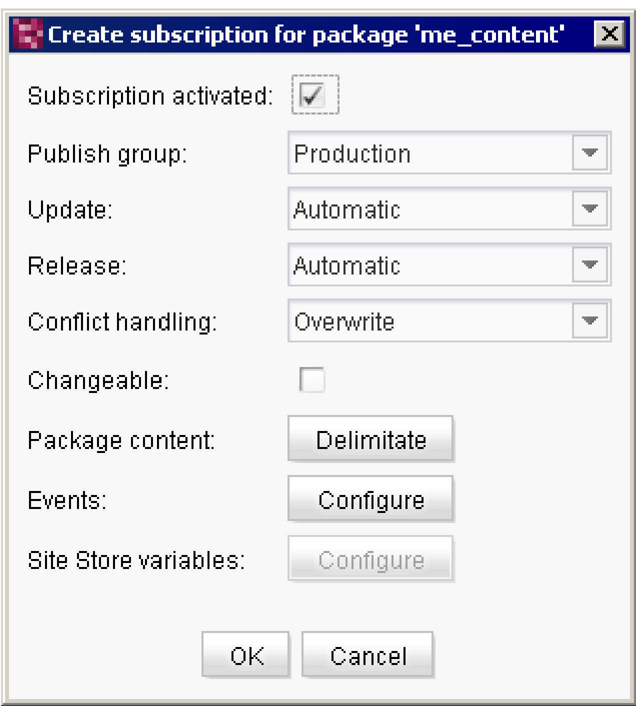


Figure 5-3: Dialog box – Create subscription for package 'xyz'

The administrator of the target project defines all the settings for the subscription in the "Create subscription for package 'xyz'" dialog box:

Subscription activated – if this checkbox is **enabled**, an update is planned with each new package version, which can be initiated manually or automatically. If the checkbox is **disabled**, there is no automatic updating of the package in the target project. If manual updating of the subscription is planned, the administrator of the target project can update the subscription, even if it is not "active" (see Chapter 5.3 page 69).

A subscription can only be deleted from the master project (see Chapter 4.2.1 page 41, "Delete" function. To "cancel" a subscription, the "Subscription activated" option should therefore be disabled here. In this case the subscription can only be updated manually; updating from the master project is therefore prevented.

Publish group – a publication group for (see Chapter 2.2 page 11) for the subscription can be selected from the drop-down list. All available publication groups are displayed. If a publication group is selected here, for which no package version

has been made "available", a subscription can be created, but an update (see Chapter 5.3 page 69) does not take place until a package version exists for this publication group:

"No versions existing for package xyz."

Update – the type of update for the package in the target project can be selected from the drop-down list. If **automatic update** is set, the import from the master project is initiated and takes place automatically in the target project. If, on the other hand, a **manual update** is set, the import from the target project is initiated with the help of the "Update subscription" menu item (see Chapter 5.3 page 69). The manual update can also be performed if the subscription is not "active".

Release – the release rule for the package can be adjusted using the drop-down list. The release can be given **automatically**, i.e. after the package has been imported, all the objects in it are automatically released in the target project. The release can also be set via a **workflow**. Both settings are only relevant if the target project works with releases (see Chapter 2.3.3 page 14). If this is not the case, the entries are simply ignored.

> *If release is used in the target project, different release states can occur, if a package is imported again after the "Release" workflow has been started. At this point the new imported object no longer corresponds to the first released state.*

Conflict handling – the drop-down list controls the procedure in case of a conflict on importing the package. These conflicts can only occur if the "Changeable" checkbox (see below) is active. This means, the package content can be changed locally in the target project. These local changes can cause a conflict situation to occur with the next update. A conflict is only caused if the change state of an object is set to "changed" or "blocked" (see Chapter 8.4 page 97). The change state is set manually using the context menu of the respective objects.

Depending on the change state set and the conflict resolution set, changes to objects are overwritten, copied or the updating of the entire subscription is prevented.

- **Overwrite** – the local changes are overwritten by the new package content
- **Cancel** – the import is canceled.
- **Copy** – a copy is created of the node on which the conflict occurred. An exception is the nodes in the Site Store: here no copies of nodes are created,

instead they are overwritten.
The precise results of the conflict handling, depending on the change state set, are described in Chapter 8.4 on page 97.

Changeable – if this checkbox is **enabled**, write permission is assigned to the target project for the imported objects. If the checkbox is **disabled**, the imported objects can be seen and used in the target project, but cannot be changed. An error message appears if an attempt is made to block the object in the target project. This setting also affects the order in which objects are imported into the target projects (see Chapter 5.4.3 page 73).

Error message: "This object belongs to package xyz and couldn't be modified."

This setting option is also available in the Package Pool of the master project (see Chapter 4.1.4 page 27). However, the entries cannot contradict each other. If "changeable" is selected in the package, this can be switched off again in the subscription. However, if "changeable" is disabled in the package, i.e. the package is write-protected, the option is also disabled in the subscription.

Package content: Delimitate this button opens the "Choose node list" dialog box, to limit the package content for importing (see Chapter 5.1.3 page 63).

Events: Configure the button opens the "Configure events" dialog box, in order to edit or delete events that already exist in the package (see Chapter 5.1.4 page 65).

Structure variables: Configure the button opens the "Overwrite the structure variables" dialog box, in order to overwrite the structure variable values in the target project with the values of the package structure variables (see Chapter 5.1.5 page 66).

OK: click this button to create a new subscription.

Cancel: Click the button to cancel the action. No new subscription is created.

5.1.3 Limit package content in the subscription

Delimitate: Click the button to open the "Choose node list" dialog box.

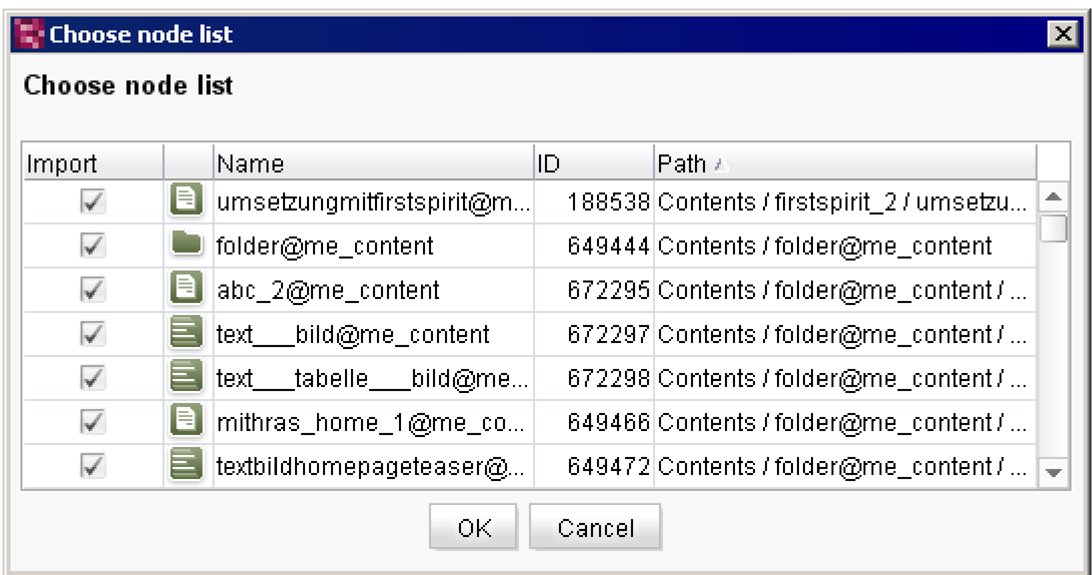


Figure 5-4: Dialog box – Choose node list

All objects contained in a package version are listed in the dialog box.

"Import" checkbox – is enabled for each object by default. If certain objects are not to be imported into the target project, the corresponding checkbox must be disabled. Pages from the Page Store can always only be disabled together with the child elements (sections).

> *Caution: If package content is limited here manually, the dependencies between package content must always be taken into account (see Chapter 2.1.2 page 7). If nodes are deleted manually, which must be contained in the package, a faulty import will occur!*

Name – shows the name of the object from the master project. Objects, which are integrated in a package are assigned a namespace extension.

Figure 5-5: Namespace extension for package content

In FirstSpirit Version 4.1 and higher it is possible to disable the namespace extension (see Chapter 4.1.6 page 34). In this case the objects are displayed without appended "@PackageName".

ID – shows the object ID from the master project.

Path – path to the object in the project tree of the master project

5.1.4 Configure events for a subscription

Configure: Click the button to open the "Configure events" dialog box.

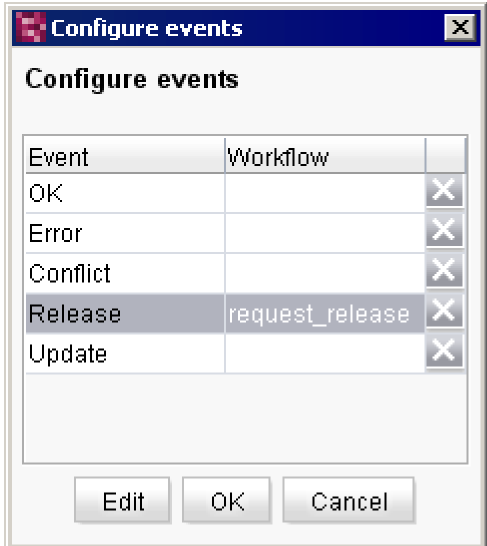


Figure 5-6: Dialog box – Configure events in the target project

Here events, e.g. errors or release, which were defined on creating a new package version (see Chapter 4.1.5 page 28) in the master project, can be assigned new workflows. This dialog can be used to delete the workflows for the target project or replace them with other workflows. New events cannot be created in the target project.

Deletes an existing workflow from the events table.

Edit: click the button or double-click the selected event to open the dialog box for selecting a workflow (see Chapter 4.1.5.1 page 31).

OK: click to accept the changes and close the box.

Cancel: the process is cancelled and the box is closed.

5.1.5 Configure structure variables

Configure: this button opens the "Overwrite the Site store variables" dialog box:

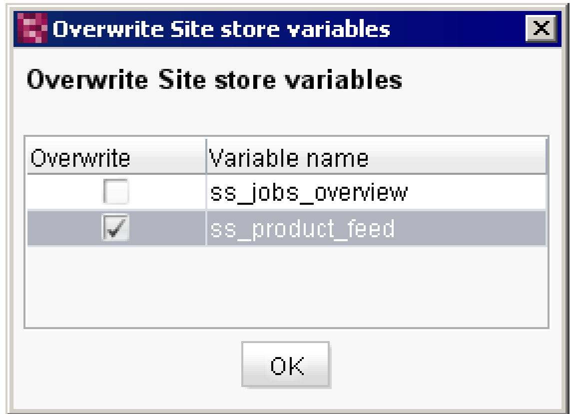


Figure 5-7: Dialog box – Overwriting the Site store variables

Structure variables can be defined for each node from the Site Store contained in the package. These structure variables can be integrated into the package content and can therefore be imported into the target projects (see Chapter 4.2.8 page 53).

When subscribing to a package, the "Overwrite the Site store variables" dialog box is used to configure how the structure variables of a package are dealt with.

- **"Overwrite" checkbox** – if this checkbox is **enabled**, the values of the structure variables are copied from the package into the target project. In the example, the value of the "ss_product_feed" variable from the master project is assigned to the variable of the same name in the target project. If the checkbox is **disabled**, the values of the structure variables are retained in the target project. In the example, the values of the structure variables "ss-Category_2" and "ss_loginRequired" from the master project therefore have no effect on variables with the same name in the target project.

If the structure variables contained in the package were not yet available in the target project, they are created when imported into the target project – regardless of whether the "Overwrite" checkbox is enabled or not.

5.1.6 Create subscription

If the configurations were made to date (as explained in Chapter 5.1.1 to 5.1.5), the subscription is displayed first in the overview (see Figure 5-8, displayed with orange

colored marking (for details of the color coding of subscriptions, see Chapter 5.3 page 69) and is initially created using the Update button.

5.2 Edit subscription

The "Edit subscription" menu item opens the "Edit subscriptions" dialog box. A list of all packages subscribed to for the project is shown in this box.

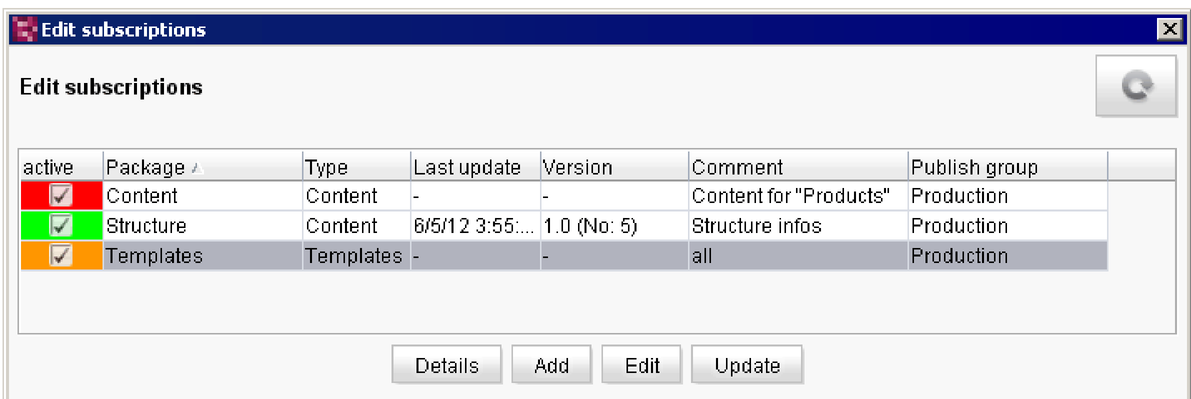


Figure 5-8: Dialog box – Edit subscriptions

active – this checkbox is identical to the "Subscription active" checkbox in the dialog in Figure 5-3. If it is **enabled**, the subscription is active for the corresponding package and can be updated if a new package version is available (orange colored coding). If the checkbox is **disabled**, the subscription can no longer be updated automatically (from the master project, see Chapter 4.3 page 56). In this view the state in the "active" column can be changed by clicking the checkbox. For details of the color coding of subscriptions see Chapter 5.3 page 69 and Chapter 6 page 78.

Package – unique package name.

Type – indicates the type of package (content or template package – see Chapter 2.1.1)

Last update – date and time of the last update of the subscription by a new package version in the target project. If there is no entry there, there has not yet been an import into the target project.

Version – unique version number of the package (assigned by the system). If there is not yet an entry there, there is not yet a package version for this package in the required publication group.

Comment – optional comment on the package version.

Publish group – each subscription is taken out for precisely one publication group. A package version can only be imported if the subscription is active *and* the package has been marked as being "available" for the given publication group (see Chapter 4.2.5 page 47).

Details Click this button to open the "Detail Info: Project / Package" dialog box. The box corresponds to the subscription detail information from the "Package Pool Overview" (see Chapter 6.1.1 page 81). The box here is purely for information purposes, the displayed values cannot be changed.

Add: click this button to open the "Choose package" dialog box; the button adds a new subscription to those already in the list. The sequence is analogous to that of the "Create subscription" menu item (see Chapter 5.1 page 59).

Edit: click this button to open the "Edit subscription for 'PackageName' package" dialog box (or double-click the relevant row). All the settings for the subscription are defined in this dialog box (see Chapter 5.1.2 page 61).

Update: click this button to update a subscription directly from the "Edit subscription" menu item. The precise procedure for updating a subscription is described in Chapter 5.3 page 69.

Initially, subscriptions are also created using this button.

5.3 Update subscription

The "Update subscription" menu item is only required for *manual updating* (see Chapter 5.1.2 page 61) of the subscription in the target project. However, in this way, all subscriptions can be updated, regardless of whether manual or automatic updating was set in the subscription, or whether a subscription is marked as being active or inactive. This function is therefore used to update using the so-called "pull" method (see also Chapter 2.3.1 page 13).

If a subscription is *active* and set to *automatic update*, the "Update subscriptions" button is usually not needed. With automatic updating, the import from the master project is initiated by publishing a new package version (see Chapter 4.3 page 56). However, if an error occurs in the target project during the automatic update, the update can be easily repeated by a manual update in the target project.

If a subscription is set to *automatic update*, but was in *inactive* state at the time the new package was published, the update is not performed automatically. In this case the subscription is marked "not up-to-date" and has to be updated manually.

If a subscription is set to *manual update*, the update always has to be carried out manually in the target project. The *active* or *inactive* status is irrelevant for manual updating.

The "Update subscription" menu item opens the "Edit subscriptions" dialog box (see Figure 5-8). Users are familiar with this box from the "Edit subscriptions" menu item (see Chapter 5.1.6 page 66). Here only the "Update" button is relevant for manual updating of a subscription.

> **!** *Before updating, the package dependencies should be checked (see Chapter 2.1.2 page 7). Dependencies on **template packages** are defined in the package properties (see Figure 6-6). These dependencies are automatically checked. If the dependent template packages are not updated, or not in the right order, the updating is cancelled and an error message is displayed:*

"The dependent template package (Templates) is not up to date. Please import the template package first."

Optional dependencies on other content packages are displayed in the "Detail Info" dialog box (see Figure 6-6: Dialog box – Detail information on a package), which can be opened via the package overview. These dependencies are not checked

automatically on updating. If the dependent content packages are not updated, or not in the right order (1. Import dependent content package, 2. Import package containing the references to the dependent package), they can cause errors in the target project: for example, on updating page references, if the referenced page and the page reference are located in different packages. If, in this example, the package with the page reference is updated first and then the package with the referenced page is updated, an error is caused in the target project. To correct the error, the page reference in the target project must be deleted and the package must then be updated again with the page reference.

> *If the subscription is for a template package, which contains objects from a database schema, the database configuration must be adjusted in the project properties of the target project before the update is made (see Chapter 11 page 129). Otherwise a corresponding error message is output.*

Update: clicking the button initiates a manual update of a subscription from the target project (see Chapter 2.3.1 page 13). As the updating of a subscription is a sensitive step, a confirmation prompt appears before the update.

"Update subscription for package 'xyz'?"

No – the window is closed, the subscription is not updated.

Yes – The subscription update is now started. An update is only useful for subscriptions which do not have an up-to-date state. The update state is easily identified by the color coding for the subscriptions in the "Edit subscriptions" dialog box (see Figure 5-8).

The default case for an update is:

(orange) – The subscription is currently not up-to-date. A new package version is available for the subscribed to package and the defined publication group, which can be imported into the target project.

> *The orange colored code (or green for "up-to-date", see below) only relates to content, the properties of the package (see Chapter 4.1.2 page 20) or the subscription (see Chapter 5.1.2 page 61) can have changed compared to the last update, although a "green" state is displayed.*

If the update is started, the following message appears after a successful import:

"The subscription update is finished."

Close: Click this button to close the box.

Show protocol: click the button to display the log of the package import into the target project. Possible errors are listed here in detail.

The icon in the top right-hand edge of the box should then be clocked in the "Edit subscriptions" box. Only after the view has been updated is the new color coding for the subscription displayed. Following the update the subscription is now either in the "red" or "green" state.

From (orange) to (green) = subscription was updated successfully.

From (orange) to (red) = An error occurred during the update. The subscription is not up-to-date. In this case the log of the import should be displayed and evaluated.

A special case for an update exists if the color coding before the update was orange, but there is no package version available for importing. This error can occur if a subscription has already been created, but no package version is available yet, possibly because a package version doesn't exist yet, but possible also because there is not yet a package version available for the subscribed for publication group. In this case an error message is displayed:

"No versions existing for package 'xyz'"

If a new package version has been successfully imported into the target project, the editing environment still shows an old view of the project. Therefore, after each import the view should be updated by pressing the button or F5. Only then is all the content contained in the package displayed with a corresponding symbol in the project tree. In the "Classic" Look & Feel this is a blue dot, in the "LightGray" Look & Feel it is a package symbol. The symbol is only visible if the "Show symbols (metadata, packages, permissions)" option is active in the "Extras" menu item.

For further information on the color coding of subscriptions, see Chapter 6 page 78.

5.4 Combine package and target project content

5.4.1 General information

The Package Pool can be used to copy content from a master project into several target projects. To do this, objects from a package are imported into the respective target project. In the target project the package content mixes with content that already exists in the target project. For example, one page of the Page Store is maintained directly in the target project, however, another page is maintained in the master project and is only imported into the target project. For most content, this combination of package and target project content is possible without problems, provided certain rules (e.g. dependencies) are followed. The Package Pool can also be used to combine structures, which can normally not be created independently in the target project, for example, an individual section (see the following chapter).

5.4.2 Combine sections

In the target projects, content imported from a package can be supplemented with individual content, a page imported from a package, for example, can be extended by any number of sections. To do this, the "Changeable" checkbox must be enabled in the subscription (see Figure 5-3) and in the package settings (see Chapter 4.1.4). This setting is used to assign the target project with write permission for the imported objects.

For example, the Package Pool can be used to distribute company-wide uniform AGB (terms and conditions of business) pages to the individual subsidiaries. Within the subsidiaries, these pages and sections can then be supplemented with other company-specific sections, which are only contained in the target projects (subsidiaries), but are not contained in the package. The general part of the content, in this example the AGB pages, is therefore maintained and updated via the Package Pool, the specific sections are added in the target projects where they are also maintained. The sections added in the target projects are retained if the subscription is updated. If the order of the sections in the master project changes, this can also have effects on the order of the sections in the target project (see Chapter 5.4.3 page 73).

Package content can of course not only be extended, but can also be reduced in the target projects. To do this, the package content to be imported is simply limited in the subscription (see 5.1.3 page 63).

5.4.3 Order for importing objects into the target projects

On importing objects (for example, sections) into the target projects, the order in which the objects in the master project exist within an object chain is also taken into account. This must be kept as far as possible not only during the initial import into the target project but also when changed objects are imported; equally, in cases in which object chains are extended within the target project (see Chapter 5.4.2 page 72).

Changes to the imported objects in the target project can only be made if the "Changeable" checkbox has been enabled for subscriptions (see Chapter 5.1.2 page 61). This setting also affects the order in which objects are imported:

If a subscription is marked "**not changeable"**, the content from the master project cannot be changed in the target project (no write permission in the target projects). In this case the right to change lies in the master project. This setting also affects the import order of objects in the target project. If the order changes – for example, the order of the sections of a page – in the master project, when the subscription is updated this changed order is also copied into the target project. This not only applies to the first-time roll-out of the content in the target project but also to an update of existing content in the target project.

If a subscription is marked as being "**changeable"**, the content from the master project can be changed in the target project. The editors in the target project can add other objects to imported package content (for example, a new section to an imported page, and can also change the order of the objects in the target project. These changes should usually not be lost if the subscription is updated again. Following the initial import of the package content, the following therefore applies:

- New objects, which are added in the target project to already imported package content (for example, a new section to an imported page), are retained in the target project when the package content is updated.
- New objects, which are added to existing package content in the master project (for example, a new section to a page, which is already part of a package), are copied into the target project. The sorting in the existing package content is carried out according to specific rules:

  FirstSpirit Version 4.2R4 and higher: When objects from the master are sorted in the target project, the binding with the preceding object ("predecessor") has priority. I.e.,

  - if both a predecessor and a successor exist, the new object is added after the predecessor,

- if only a predecessor exists, the addition is made after the predecessor,
- if only a successor exists, the addition is made in front of the successor

Before, objects were more strongly bound to the following object ("successor"), provided a successor existed.

The order defined in the target project for the package content, (for example, by the initial roll-out or re-sorting of the package content in the target project), is retained, even if the package content in the master project is re-sorted and rolled-out again.

Example 1 – Initial import of the package content into the target projects:

The master project contains a page with 3 sections (Absatz_1, Absatz_2, Absatz_3) in the following order:

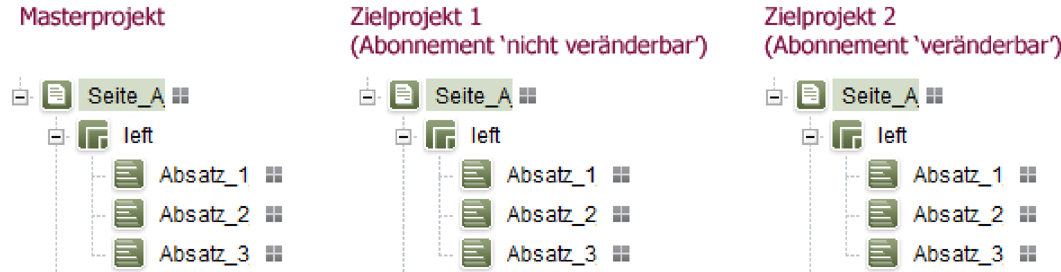


Figure 5-9: Package Pool – Page with three sections

If a package with these objects is rolled out in the target project, the order of the sections is retained. This applies to the initial import into the target projects (regardless of whether the subscription is changeable or not.

Example 2 – Updating the package content (without change in the target project):

Further sections (Absatz_a, Absatz_b, Absatz_c, Absatz_d) are now added to the page:

Figure 5-10: Package Pool – Add new sections

When the update is made the order from the master project (Absatz_a, Absatz_1, Absatz_b, Absatz_2, Absatz_c, Absatz_3, Absatz_d) is copied into both target projects. For target project 1 ("Zielprojekt 1"), this is the default behavior, as the order of the master project is always retained. For target project 2 ("Zielprojekt 2") this behavior only applies because no changes have been made yet to the package content.

Example 3 – Updating the package content (with change in the target project):

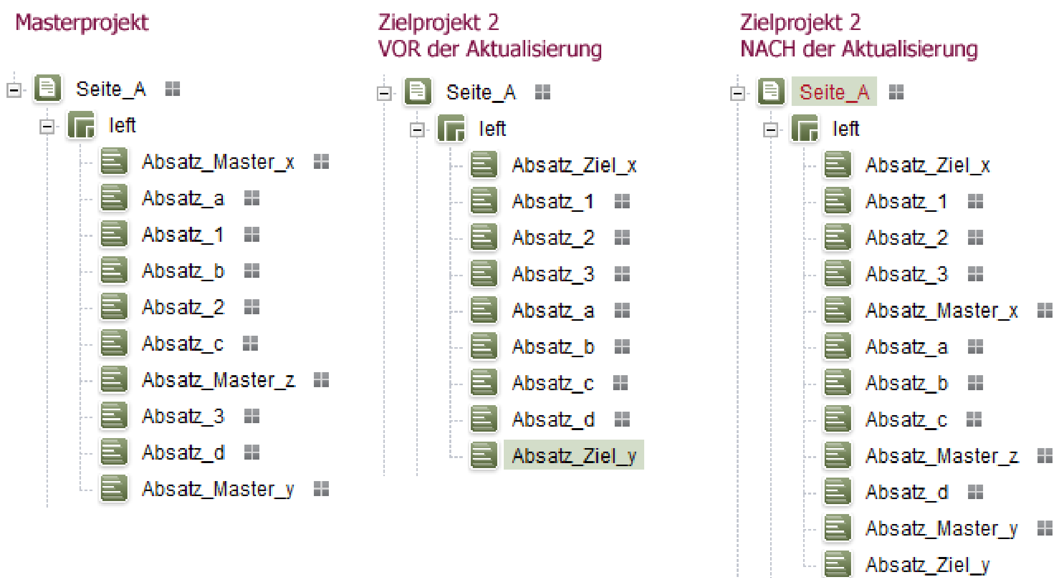


Figure 5-11: Package Pool – Combined sections in the target project

Three sections (Absatz_Master_x to Absatz_Master_z) are added to the page in the master project. Unlike the second example this time, before rolling out the updated package in the target project, changes are made to the page that comes from the master project (only possible if the package content as marked "changeable" in the subscription):

- the order of the imported sections in the target project is changed manually (Absatz_1, Absatz_2, Absatz_3, Absatz_a, Absatz_b, Absatz_c, Absatz_d)
- two new sections are added in the first and last position (Absatz_Ziel_x, Absatz_Ziel_y)

Following the renewed update of the package content, the difference from example 2 is clearly seen:

- The changed order of the sections (Absatz_1, Absatz_2, Absatz_3, Absatz_a, Absatz_b, Absatz_c, Absatz_d), which was defined in the target project for the package content, is retained, even if the sections are arranged differently in the master project (Absatz_a, Absatz_1, Absatz_b, Absatz_2, Absatz_c, Absatz_3, Absatz_d)
- The new sections from the master project are added to the existing content of the target project according to the following rules:
  - Absatz_Master_x only has one successor (Absatz_a) and is therefore added to the target project before the successor

- Absatz_Master_y only has a predecessor (Absatz_d) and is therefore added to the target project after the predecessor
- Absatz_Master_z has a predecessor (Absatz_c) and a successor (Absatz_3) and, as the predecessor is given priority, is added to the target project after the predecessor.

- The new sections from the target project are retained.

6 Overview menu item

Figure 6-1: Menu item – Overview

The Overview menu item opens the "Package Pool overview" dialog box. Here the same information is displayed in the master and target project(s).

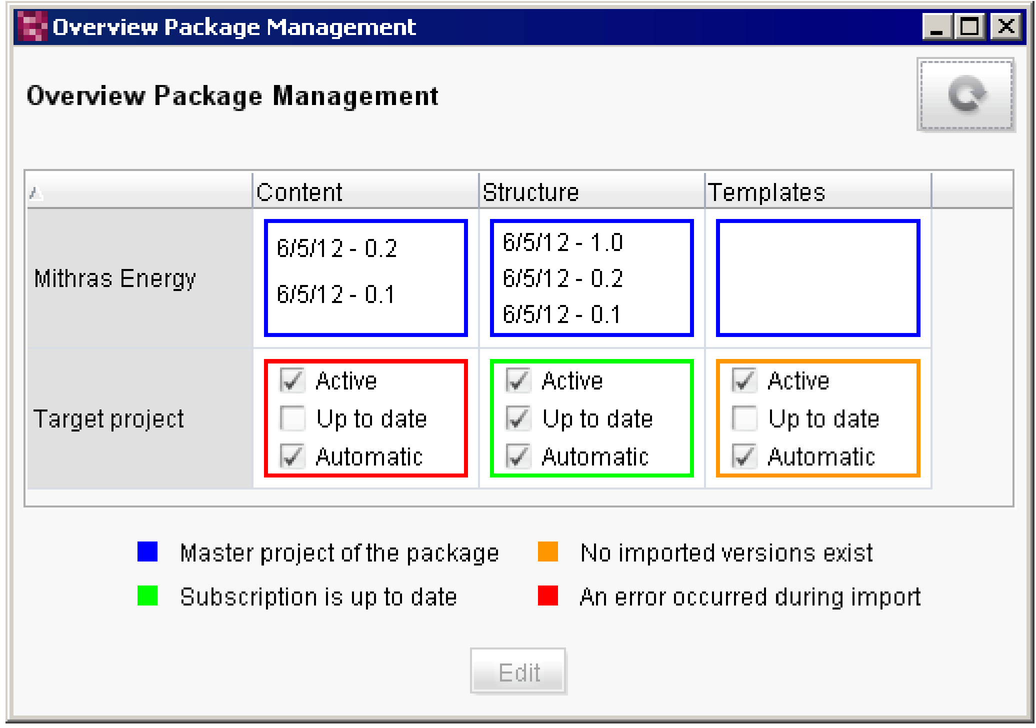


Figure 6-2: Dialog box – Package Pool overview

The box shows the most important information about packages, projects and the current state of the subscription in an overview window. All projects on the server are displayed on the vertical axis and all known server-wide packages are displayed on the horizontal axis. The intersection between a package and a project shows brief information about the status of the subscription for the package in the relevant project.

Subscriptions in the **target projects** are displayed with the following information:

Active – enabled checkbox indicates that the subscription to the package is active. The state can be changed directly in this overview by double-clicking the subscription or by using the "Edit" button (see Chapter 6.1.1 page 81) or via the "Edit subscriptions" dialog box (see Figure 5-8).

Up to date – enabled checkbox indicates that the currently up-to-date package version has already been imported into the target project. The checkbox is only enabled for subscriptions marked green; the checkbox is disabled for subscriptions marked orange or red (for details of the subscription color coding: see below).

Automatic – enabled checkbox indicates that the package content is automatically updated in the target project, as soon as a more up-to-date package version is provided (push method, see also Chapter 2.3.1 page 13). The status can be changed using the "Update" parameter in the dialog box in Figure 5-3.

Packages from the **master projects** are displayed with the following information:

23.11.07 - 9
16.11.07 - 8
16.11.0
23.11.07 - 9 (Test, Development)
16.11.07 - 8 (Test, Development, Production)
16.11.07 - 7 (Test)

Figure 6-3: Brief information on a package in the master project

Packages from the master projects display the last three package versions within the blue frame. In this way the user can see at first glance, how up-to-date the package version is in the respective project and which package version should be updated urgently in the target projects. If the user moves the mouse over the blue frame, a tooltip is shown with the corresponding publication groups as additional information.

If the intersections between the package and project are empty, there is not yet a subscription for the package (*information on creating a new subscription: see Chapter 5.1 from page 59*). If an empty blue frame only is displayed, a package version has not yet been created for the package in the master project (*information on generating package versions: see Chapter 4.2.4 page 44*).

In order to clearly show the status of a subscription, in addition to the short information, color codes have been introduced, which are displayed in the overview as a colored frame.

Master project of the package	No imported versions exist
Subscription is up to date	An error occurred during import

Figure 6-4: Color coding for a subscription's state

blue frame – identifies the master project for the respective package.

green frame – means that the currently most up-to-date package version has already been successfully imported into the target project.

red frame– identifies a faulty import into the target project. In this case the log file should be opened from the detail information (see Chapter 6.1.1 page 81) (see Chapter 6.1.3 page 84).

orange colored frame – means that a more up-to-date package version is available for importing, but the target project has not yet been updated (*for information on updating the subscription from the target project: see Chapter 5.3 page 69, Information updating the subscription from the master project: see Chapter 4.3 page 56*).

6.1 Detail information

Edit: Click this button or double-click the required package-project relationship to open the "Project/Package detail info" dialog box (see Figure 6-5). At each interface in the overview, additional information to the information from the overview can be viewed here (see Figure 6-2). A differentiation is made between *information on subscriptions* (green, orange colored or red frame, see Chapter 6.1.1 page 81) and *Information on packages* (blue frame, see Chapter 6.1.2 page 83).

6.1.1 Detail information on subscriptions

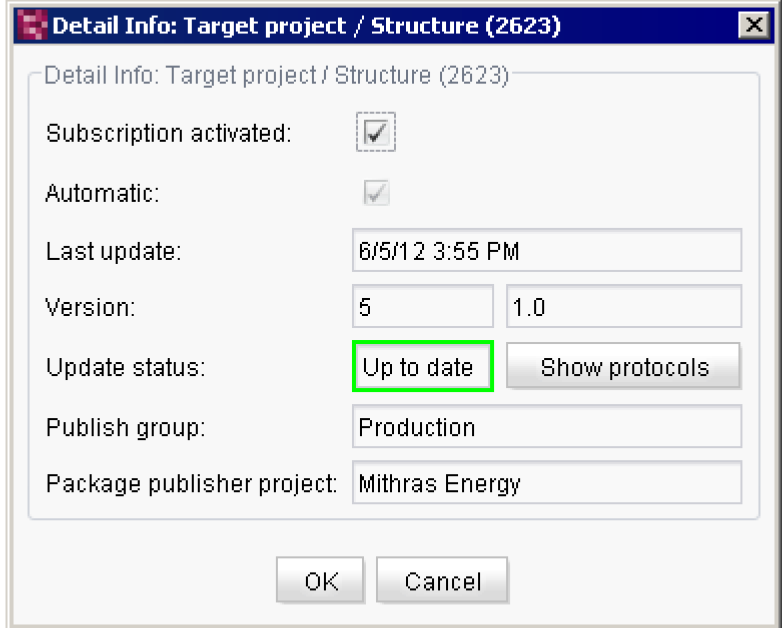


Figure 6-5: Dialog box – Detail information on a subscription

The name of the subscribing target project (here: "FIRSTools_20071010_Target") and the subscribed to package with ID (here: "Default page (1994222)") are displayed in the title bar of the window and as the heading in the content area.

Subscription activated – if the checkbox is enabled, the subscription for the package is active, this means all new package versions are made available for importing into this project (is also displayed in the short info). The checkbox is active and can be edited in this dialog (see also Chapter 5.1.2 page 61).

Automatic – if this checkbox is enabled, the update of a new package version is automatically performed in the target project. The checkbox is inactive and is used for information purposes only. The status can be changed in the subscription

properties (see Chapter 5.1.2 page 61).

Last update – sows the date and time of the last update of the package in the target project.

Version – the first field shows the unique version number for the package version assigned by the system. The second field shows an additional, version number assigned manually by the package developer of the master project.

Update status– shows the state of the update in the target projects. The information reflects the color coding in the overview window. The three known states of a subscription are given here:

- Up-to-date – the most up-to-date package version has been imported successfully.
- Out of date – a more up-to-date version is available for importing.
- Error – faulty import into the target project.

Show protocols: opens the "View log file" dialog box. The log file records the precise procedure during the import of the packages and is particularly interesting if a faulty import occurs, as it can be used to evaluate the error that has occurred. For further information see Chapter 6.1.3 page 84.

Publish group – shows the publication group(s), for which the subscription was taken out.

Package publisher project – shows the master project, i.e. the project in which the package was created.

OK: Any change to the "Active" checkbox is adopted for the subscription. The "Detail Info" dialog box is closed.

Cancel: the process is cancelled and the box is closed.

6.1.2 Detail information on packages

Detail Info: Mithras Energy / Content (2575)

Detail Info: Mithras Energy / Content (2575)

No.	versi...	Date	Available	Comment	Dependent packages	Log file
4	0.2	Jun 5, 2012	Production, Development	optimized	Templates	
3	0.1	Jun 5, 2012	Development, Test	Preliminary	Templates	

OK

Figure 6-6: Dialog box – Detail information on a package

Further information on the package can be opened within the blue frame in the "Package Pool overview" dialog box by double-clicking.

Before updating subscriptions in the target project (see Chapter 5.3 page 69) the detail information on the package should be checked via this dialog, in order to find out any dependent content packages ("Dependent packages" column), which have to be imported ***before*** *the content package, which contains the references to the dependent objects.*

The name of the subscribing master project (here: "Mithras Energy") and the subscribed to package with ID (here: "Content (2575)") are displayed in the title bar of the window and as the heading in the content area.

The table shows the package versions generated (see Chapter 4.2.4 page 44) for the different publication groups. By default the most up-to-date package version is displayed at the top.

No. – shows the unique version number assigned by the system.

Version – shows the manually assigned version name.

Date – shows the date on which the version was created.

Available – shows the publication group(s), for which this package version is "available".

Comment – optional comment on the package version.

Dependent packages – shows the dependent packages (template and content

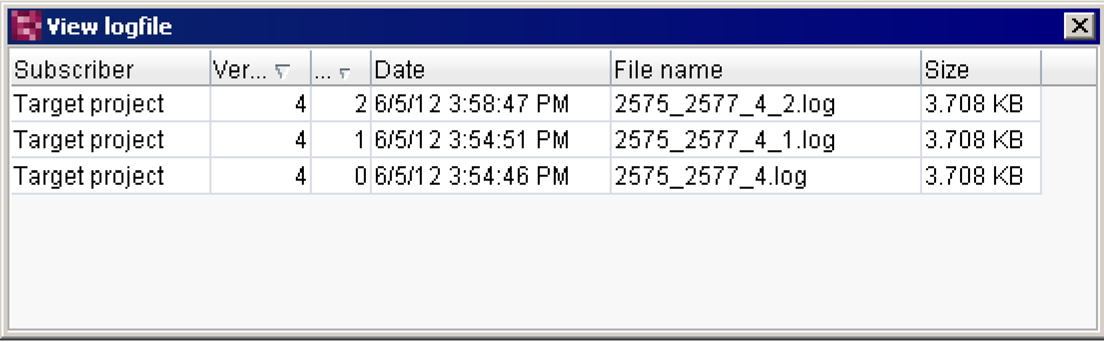
packages, see also Chapter 2.1.2 page 7) of the respective package version.

Import logs: opens the "Display log file" dialog box (see Chapter 6.1.3 page 84).

OK: The information box is closed.

6.1.3 Display logs

Show protocols: click the button to open the "View log file" dialog box:



Subscriber	Ver...	...	Date	File name	Size
Target project	4	2	6/5/12 3:58:47 PM	2575_2577_4_2.log	3.708 KB
Target project	4	1	6/5/12 3:54:51 PM	2575_2577_4_1.log	3.708 KB
Target project	4	0	6/5/12 3:54:46 PM	2575_2577_4.log	3.708 KB

Figure 6-7: Dialog box – View log file

With each import of a package version into a target project a log file is created, which logs all information during the execution and is important for possible bugfixing. The "View log file" dialog box can be used to select a log file for each subscription and each imported package version. The table can be sorted by clicking the respective column.

Subscriber – gives the target project, into which the package was imported.

Version – shows the version number assigned by the system.

No. – shows the number of attempts to import into a target project. If automatic import is set, the number is normally "0". However, if an error occurs while importing the package version, the import is initiated again and the number is increased by "1".

Date – shows the date and time of the import.

File name – shows the name of the log file. The name is made up of:

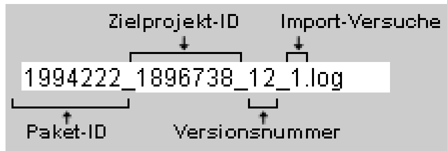


Figure 6-8: Log file name composition

Double-click an entry to open the selected log file:

```
Log - V: 5, No.: 0, Target project / Structure, (2623_2577_5.log)
INFO  05.06.2012 15:55:51.496 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): creation
INFO  05.06.2012 15:55:51.496 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): starting
INFO  05.06.2012 15:55:51.497 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): no templ
INFO  05.06.2012 15:55:51.498 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): found 2
INFO  05.06.2012 15:55:51.498 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): found 1
INFO  05.06.2012 15:55:51.526 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): found 1
INFO  05.06.2012 15:55:51.528 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): locking
INFO  05.06.2012 15:55:51.528 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): locking
INFO  05.06.2012 15:55:51.528 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): importin
INFO  05.06.2012 15:55:51.529 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): remove p
INFO  05.06.2012 15:55:51.530 (de.espirit.firstspirit.service.ppool.PPoolLoggerImpl): remove p
Close
```

Figure 6-9: Dialog box – Log file

Log entries with `ERROR` state are particularly interesting here. In case of a faulty import or update, it may be possible to identify here whether other referenced objects from the master project are required so that the import can then successfully take place

Close: click the button to close the box.

The log outputs can also be opened in an external editor. To do this, all outputs are first selected with the keyboard shortcut **Ctrl+A** and are then copied onto the clipboard with **Ctrl+C**. The external editor is now opened and the content of the clipboard is pasted into the editor with **Ctrl+V**. This method is particularly advantageous for the analysis of larger files.

7 Publication Groups menu item

Figure 7-1: Menu item – Publication Groups

The "Publication groups" menu item makes it easier for users to publish and import packages in complex work environments (see Chapter 2.2 page 11). For example, by differentiating into the three publication groups: development, production and test, packages can be published or imported into a test environment first and only then used in a productive environment, as a tested, stable package version.

The publication groups are defined server-wide and are therefore not only available in the master projects but also in the target projects. There are therefore two different areas of use for working with publication groups:

Publication groups in the master project: When it is created, each new package version is assigned the publication groups for which it is to be made available. (see Chapter 4.2.5 page 47). The package versions can then be published for all or only for individual available publication groups and are then ready for importing into the target projects.

Publication groups in the target project: Each subscription is taken out for precisely one publication group (see Chapter 5.1.2 page 61). Therefore, the most up-to-date package version available for this publication group is always imported into the target project. For example, if a subscription is taken out for the "Test" publication group, only the most up-to-date package version made available for the "Test" publication group is imported.

7.1 Edit publication groups

The "Publication Groups" menu item opens the "Edit Publication Groups" dialog box:

Figure 7-2: Dialog box – Edit publication groups

All publication groups available on the server are displayed here in a table with the following information:

Default – the enabled checkbox shows the default publication group (server-wide). This is preselected when a subscription is created (Chapter 5.1.2 page 61, "Publication group“). Precisely one publication group must always be defined as the default group. This publication group cannot be deleted, unless a new publication group has been selected as the default group beforehand.

Name – unique name of the publication group.

Description – optional description of the publication group.

Add: Click the button to open the "Create New Publication Group" dialog box. The further procedure is described under the "Add publication group" menu item (see Chapter 7.2 page 89).

Delete: Click the button to delete a publication group. The further procedure is described under the "Delete publication group" menu item (see Chapter 7.3 page 90).

Edit: Click the button to open the "Edit Publication Group" dialog box. The publication group selected in the table can be edited here.

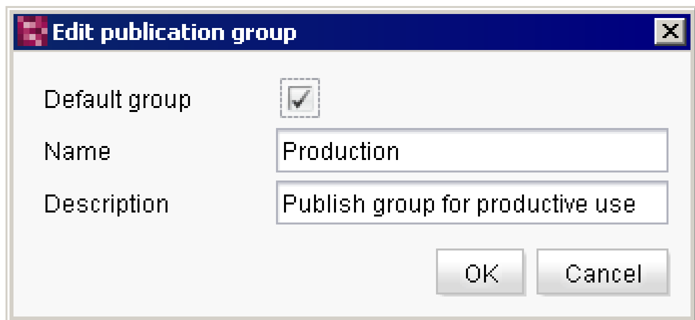


Figure 7-3: Dialog box – Edit publication group

The information from the "Edit publication groups" dialog box can be edited here for the selected publication group.

Default group – if the checkbox is enabled, the publication group is defined as the default group. Precisely one publication group must always be defined as the default group.

Name – a new name for the publication group can be given in this field. Subscriptions to date under the old name of the publication group are retained and are not taken out for the new publication group names. The publication group name therefore only has to be changed here if necessary; manual adjustment elsewhere is not necessary.

Description– a new optional description can be given in this field.

OK: click the button to confirm the changes and close the box.

Cancel: click the button to cancel the process.

7.2 Add publication group

Add: Click the button in the "Edit publication groups" dialog box to open the "Create new publication group" dialog box.

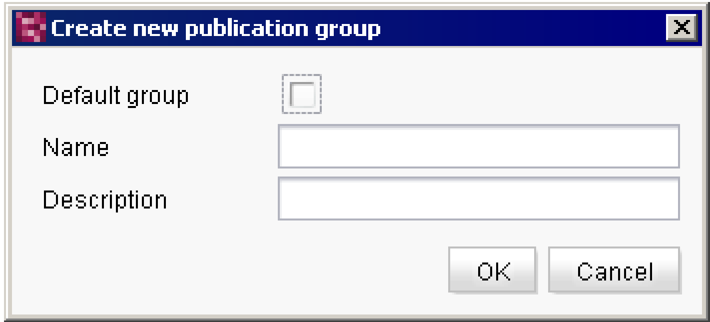


Figure 7-4: Dialog box – Create new publication group

Default group – if this checkbox is enabled, the new publication group is created as the default group. The default group to date then loses this state as exactly one publication group only can be defined as the default group.

Name – unique name of the new publication group. If the required name has already been assigned, the new group cannot be added. The "name" lettering is marked red here, in order to indicate the sources of the error and at the same time the **OK** button is not inactive. It is therefore not possible to enter two publication groups with the same name.

Description – optional description of the new publication group.

OK: click the button to create the new publication group. It then appears in the "Edit publication groups" dialog box.

Cancel: click the button to cancel the process. The new publication group is not created.

7.3 Delete publication group

Delete: Click the button in the "Edit publication groups" dialog box to delete the previously selected publication group. A confirmation prompt is displayed before finally deleting, to ensure that a publication group cannot be deleted accidentally.

Message: "Do you really want to delete publication group 'Development' ?"

Yes: click the button to delete the publication group.

No: click the button to cancel the dialog; the publication group is not deleted.

Publication groups can only be deleted if they are not used in a subscription.

A publication group, which has been defined as a default group, cannot be deleted. If a default group is to be deleted, a new publication group must be defined as the default group beforehand in the "Edit publication group" dialog box.

8 Package Pool context menu

The "Package Pool" context menu provides several functions for editing packages directly on the objects in the project tree. It is opened directly on an object or node of the project tree with a right-click. The "Package Pool" function is located under the corresponding menu item in the context menu. The "Package Pool" context menu is divided into five submenu items, which are described in the following chapters.

- Add to package (Chapter 8.1 page 91)
- Remove from package (Chapter 8.2 page 93)
- Undo package relation (Chapter 8.3 page 95)
- Change state (Chapter 8.4 page 97)
- Rebind original (Chapter 8.5 page 99)

If the submenu items have gray lettering instead of black, the required function is not available on the selected object, for example, on the pool roots.

8.1 Add to package (master project)

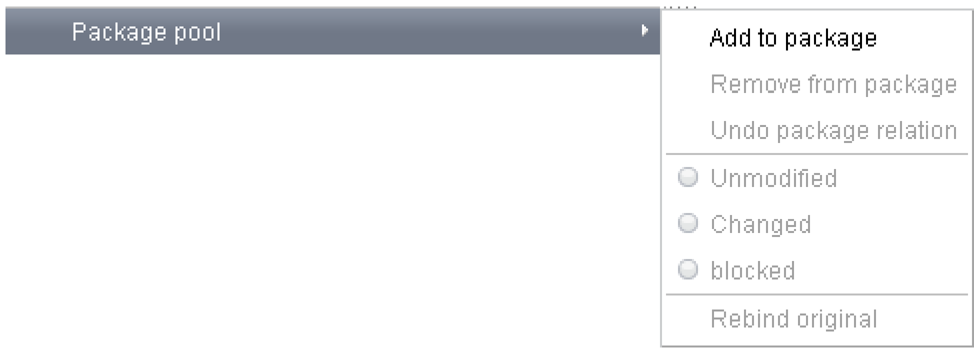


Figure 8-1: Context menu – Add to package

The "Add to package" menu item can be used to add a node or an object directly to an existing package. This function is only available in master projects and only, if the selected object is not already part of another package. Click this menu item to open the "Choose package" dialog box.

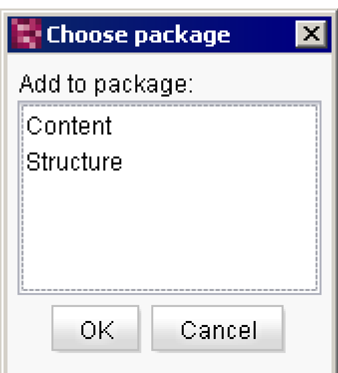


Figure 8-2: Package Selection dialog box

All packages available in the master project are listed in the "Choose package" dialog box. The selection displayed here naturally also depends on the context: If the context menu is opened on a page of the Page Store, only content packages are displayed in the "Choose package" dialog box, not template packages. The package to which the selected object is to be added is chosen from this list.

Cancel: Click this button to interrupt the process. The object is not added to a package and the box is closed.

OK: Click to confirm the selection and add the selected object to the content of the selected package. The box is closed. If the object has been added successfully to the selected package, the following information appears:

"Successfully added"

Each object can only be assigned to one package. When an object is added to a package, the system first checks whether the higher level nodes (parent nodes) or lower level nodes (child nodes) of the selected objects are already part of another package. In this case the package selection list (see Figure 8-2) is shortened accordingly. Under certain circumstances, only the package to which parent or child notes possibly already exist remain. A selection list is then no longer displayed, instead only one package is provided for the object to be added to:

"Do you really want to add this element and all child elements to package 'Content'?"

Following successful addition to the package the reference name of the previously selected object is displayed in the tree structure with the name extension (see Figure 5-5: Namespace extension for package content) and a blue dot or a package symbol behind the name. The dot or package symbol is only visible if the "Show symbols (metadata, packages, permissions)" option is active in the "Extras" menu item.

In FirstSpirit Version 4.1 and higher it is possible to disable the namespace

extension (see Chapter 4.1.6 page 34). In this case the reference name of the object added to the package remains without namespace extension.

If the context menu is implemented on a folder, all lower level objects are added to the package. If the folder already contains objects, which have already been integrated into another package, these objects are not added to the new package.

The "Add to package" context menu function therefore fulfils the same function as the "Add" button in the dialog box 4.2.7 page 49.

8.2 Remove from package (master project)

Figure 8-3: Context menu – Remove from package

The "Remove from package" menu item can be used to remove a node or an object directly from a package. The "Remove from package" function is naturally only available for objects which are already part of a package. Clicking the menu item opens a confirmation prompt:

"Really remove element from package 'xyz'?"

Yes: The selected element is removed from the package and the window is closed. In the tree view, a blue dot or package symbol is no longer shown behind the object name and the selected object is no longer part of the package.

No: Click this button to interrupt the process. The object is not removed from the package and the box is closed.

The "Remove from package" context menu function only removes the currently selected object – no lower level objects.

If an object is removed from a package, the namespace extension continues to exist see Chapter 4.2.1 page 41, "Delete" function). However, the blue dot or the package symbol behind the name, which shows the assignment to a package, disappears. The object can now be added to a new package again. In this case the namespace extension also changes (the @ is appended with the name of the new package) and a blue dot again appears behind the name in the project tree.

***In FirstSpirit Version 4.1 and higher** however it is possible to disable the namespace extension (see Chapter 4.1.6 page 34). In this case the reference name also remains unchanged if the object is removed.*

The "Remove from package" context menu function therefore fulfils the same function as the "Remove" button in the dialog box 4.2.7 page 49.

8.3 Undo package relation (target project)

Figure 8-4: Context menu – Undo package relation

While the first two context menu items are only relevant for master projects, i.e. for projects in which packages are created, the third context menu items "Undo package relation" is used in target projects. The menu item can be implemented on all subscribed objects, which have been imported into a target project from a package. These objects are displayed in the target project with a blue dot or a package symbol behind the name in the project tree.

Click the menu item to remove the package relationship of an imported object, this means, the relation of an object to a package is removed. This makes it possible to import objects from a package and to change them in the target project, although write protection was defined for the subscription. With the next update the object is re-created as a copy in the target project. The changed object continues to be retained.

If the package relationship of a subscribed to object is removed, the blue dot or package symbol behind the object name in the object tree disappears. The namespace extension on the other hand continues to be retained, so that no modifications have to be made in the referenced nodes.

***In FirstSpirit Version 4.1 and higher** the namespace extension can also be disabled (see Chapter 3.1.6 page 30). In this case the reference name remains unchanged.*

The "Undo package relation" context menu function only removes the currently selected object – no lower level objects.

8.4 Change state (target project)

Figure 8-5: Context menu – Change state

The "Change state" menu item can be used to set the change state of a node or an object. This function is only available in target projects and only for objects, which are already part of a package. The state values set in the target project are required for the conflict resolution for package import (see Chapter 5.1.2 page 61, "Conflict resolution" option). For example, if changes are made to an imported page, setting the change state "Changed" can cause a conflict in the next update of the subscription. Conflict resolution always depends on the state value set here:

- **Unmodified:** This state is the default for each object, which is included in the content of a package. The next package update overwrites the object with the content from the package. A conflict cannot occur with this setting.

> *As the state in the target project is set manually, it is possible to change the object, but to nevertheless set the change state to "not changed". The changed object is then overwritten again with the next package update. A conflict is not initiated!*

- **Changed:** By setting this value, when the package is updated a conflict is initiated, regardless of whether the package version has changed or not. The further conflict handling procedure depends on the conflict settings of the subscription (see Chapter 5.1.2 page 61).
    - **Overwrite conflict handling:** The conflict is resolved by overwriting the object changed in the target object with the object of the package version imported with the update (this can be a new version with content changed in the master project or the same version which has already been imported). The changes made to the object in the target project are lost. After the overwriting the object in the target project

corresponds to the object from the master project.

- **Cancel conflict handling:** The conflict is resolved by cancelling the update of the subscription with an error message. No objects are updated
- **Copy conflict handling:** The conflict is resolved by creating the object from the package version imported with the update as a copy: A number is appended to the reference name of the object. The original object from the target project is retained, but is automatically removed from the package binding (see also 8.3 page 95).

- **Blocked:** With this setting the object is blocked for an update, this means, it is therefore explicitly excluded from the subscription update. If a new package version is imported into the target project, a copy of the object is created. The changed object is retained in the target project, but is automatically removed from the package binding (see also 8.3 page 95). The new object is imported into the target project as a copy.

The results of conflict handling and change state in brief form:

Change state	Conflict resolution	Result
Unmodified	All	Only if the package version is changed: Object is updated; changes are lost.
Changed	Overwrite	If updating an already imported or a new package version: Object in the target project is updated with content from the master project; changes made to the object in the target project are lost.
Changed	Cancel	If updating an already imported or a new package version: Import is cancelled; object is not updated; changes from the target project are retained.
Changed	Copy	If updating an already imported or a new package version: Object in the target project is removed from the package relationship and changes are retained; new object is created from the master project as a copy.

Blocked	All	Only if the package version is changed: Object in the target project is removed from the package relationship and changes are retained; new object is created from the master project as a copy.

The change state is required, for example, for conflict handling on importing content into different project languages (see Chapter 10.2.1.2 page 117). If a master project contains, e.g. the project language English, but the target project contains German and English, the English language content is imported not only in the target project language English but also in the target project language German. The English language content then has to be translated into the German target project language in the target project. In this case the change state for the translated pages should be set to "changed" or to "blocked". Otherwise the already translated content would be overwritten again with the next subscription update.

If no change state is set for the object, changes are overwritten when the subscription is updated.

8.5 Rebind original (target project)

Figure 8-6: Context menu – Rebind original

Unlike the context menu items "Add to package" (see Chapter 8.1 page 91) and "Remove from package" (see Chapter 8.2 page 93), this function is *only* available in the subscribing target projects and only on objects which have a package binding. The "Rebind original" function removes the object node on which it was opened, from the package binding and instead integrates a new object node into the package. This should be an object, which was previously part of this package, but currently has no package binding.

The new object node to be integrated into the package is selected from a list of all the target project's objects. The selection list is limited by only displaying the store on which the context menu was opened (see Figure 8-7).

The "original" must be compatible with the object node, which is removed from the package binding, i.e. it must be involve the same type of object node, for example, a page from the Page Store, which is based on identical templates.

The selection of the object node is not checked automatically, but instead is the responsibility of the editor. The removal or addition of package binding is a sensitive action. If the wrong object is "re-integrated", possibly an object, which was never part of the package concerned, this can result in errors in the target project as, for example, the page or section templates are not suitable for the new object.

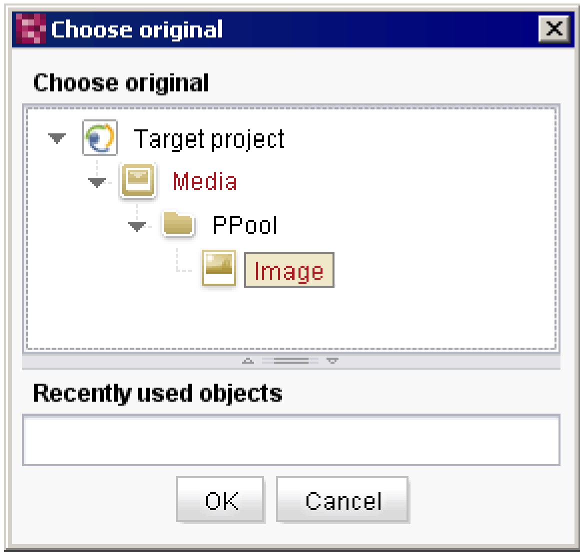


Figure 8-7: Choosing the new original node to be rebind

One possible area of use is the renewed integration of objects, e.g. pages, following translation into a language not contained in the package. To protect this page from renewed overwriting during the translation, the section change state in the target project is set to "Changed" or "Blocked" (see Chapter 8.4 page 97). If the subscription is updated, the "Copy" conflict handling set in the subscription then takes effect and creates a copy of the new imported page. The changes in the translated page are retained, at the same time however, the "old" page is removed from the package binding. Following the translation the page should be placed back under the package control. The "Rebind object" function is required for this. The function is opened on the currently imported page, the copy. "The translated original

page is then selected from the "Choose original" list, and after the selection has been confirmed is placed back under package control. The imported copy of the page loses the package binding and can, if required, be deleted from the target project (for details of translations, see Chapter 10.2 page 116).

9 Transfer existing projects into package master projects

To use the Package Pool function it is not necessary to generate an independent master project, in which only package content is managed. Each existing FirstSpirit project can adopt the role of a master project and provide package content for importing into other projects. In this way, for example, a corporation office can carry out the maintenance of the corporation presentation in its own website and therefore becomes the master project for the relevant package. Other offices then subscribe to the corporate presentation package from this project. The master project then continues to exist as a normal project.

The transfer of an existing project into a package master project must be carefully planned, as the restructuring can result in temporarily inconsistent interim states. It can also be possible that references in form and output tabs of templates, etc. have to be changed manually, so that the master and target project work without errors, as reference names can change due to the package function. The procedure explained in the following (from Chapter 9.1.1 up to and including Chapter 9.1.9) should be precisely followed, in order to avoid problems during the conversion.

Direct transformation into a package-master project is only possible for projects of FirstSpirit Version 3.1 or 4.0, as these projects use the new template syntax and the reference graph necessary for calculating dependencies exists. The transformation of FirstSpirit Version 3.0 projects requires manual adjustment steps – see Package Pool documentation for FirstSpirit Version 3.1.

9.1.1 Using the reference graph

As already explained in Chapter 2.1.2 page 7, packages can only be successfully imported and used in a target project, if they contain all the necessary objects. In addition to the objects explicitly added to a package by the package developer, it is also possible for dependent objects to exist, which are necessary for successful working with the package in the target project. Content dependencies are automatically resolved with the help of the so-called reference graph (see Figure 9-1). This means, if objects from the Page or Site Store are added to a package ("content package"), all dependent objects from the Site or Page Store and the Media Store are also automatically copied into the package at the same time ("implicit")

On the other hand, dependencies on objects from the Template and Content Store in this content package are not resolved automatically. Dependent objects from the Template or Content Store, e.g. a section template, which is required to maintain a section from the content package, must be packed in a separate package. The dependency between the content package and the template package is then defined in the content package. The reference graph can also be used to identify all dependencies between the content and template package.

Reference graphs can be requested via the **Extras / Display dependencies** context menu of an object (see Figure 9-1). Reference graphs of individual data records of the Content Store are opened via the context menu of the respective data record.

This function is available to project administrators only.

The tabs in which opening windows show the dependencies of the object in the form of incoming and outgoing edges, not only for the current state (Current status tab) but also for the most recently released state (Release status tab), provided the project uses the Release option:

Figure 9-1: Displaying dependencies via the reference graph

Each object on which a dependency exists is displayed with an ID and corresponding object icon. Double-click "Display the next elements" to show other dependent elements. Double-click an element to also show the references to this object.

For further information on the reference graph, please refer to the FirstSpirit Manual for Editors.

9.1.2 Structuring the package content

In order to simplify the creation of a package, all content, which is to be integrated in a package later, should be moved into separate folders in the master project. This is possible for all objects from the Page, Media and Template Store (not for objects from the Site Store). The folders are later used as structuring help for the content in the target project. With the help of the folders it is more quickly clear, which content

was imported from a master project via the Package Pool and which content bundled in folders clearly distinguishes itself from the original content of the target project. All objects, which are not integrated into the package in folders, are added to the target project at the highest level in the respective Store, and the structure is therefore lost. However, the structuring through folders is also beneficial for the clarity and transparency of the master project.

Apart from the explicitly added objects, there can also be objects which are implicitly added to the package, if dependencies exist between objects (see Chapter 2.1.2 page 7 and Chapter 9.1.1 page 102). These implicitly added objects must be checked by the creator of the package and are also stored in separate folders. Automatic calculation of the dependencies is not yet possible for template packages. Templates have to be manually compiled for a package.

Firstly, all the templates required must be saved in separate folders in the Template Store, this also applies to each subnode ("page", "section", "format templates", etc.). Media from the Media Store can be referenced within the templates. These media objects belonging to a template, so-called technical media, can be integrated in a template package and contain, for example, JavaScript files (*.js), cascading style sheets (*.css) or graphic layout files (see Chapter 2.1.1 page 7). To this end, all technical media belonging to the template should also be grouped together in a folder in the Media Store. Non-technical media are integrated in content packages and to this end should of course also be saved in separate folders.

Each object can be contained in a maximum of one package only.

For example, if technical media are required in more than one package, a second folder must be created for this package in the Media Store, which contains a copy of this object.

Successful package creation therefore always requires comprehensive project knowledge.

9.1.3 Limiting the picture selection in templates

Due to the automatic resolution of dependencies within content packages, for example, media files, which are integrated in a section via the DOM Editor input component, are implicitly added to the package as soon as the page with the

corresponding section is integrated in the package. Under certain circumstances, in this way, very many implicitly referenced media can be integrated in a package, which exist in different places in the master project (e.g. in different (sub) folders in the Media Store). On the one hand, this is unclear and on the other hand it can cause conflicts when the packages are imported. One solution is to limit the picture selection option for the "DOM Editor" and "Picture" input components.

For the **"Picture" input component,** this limitation is achieved with the help of the `<FOLDER>` and `<SOURCES>` tags within the section or page template. With the `<SOURCES>` tag it is possible to limit the election or display to defined folders (incl. subfolders). This is a positive list, i.e. only the given folders are allowed. To allow a folder, a `FOLDER` tag with the `name` parameter and a valid folder name must be given.

If, in addition to the picture selection, the upload option for media is also to be limited to a specific folder, the `uploadfolder` attribute is also required[1]:

```
<CMS_INPUT_PICTURE... upload="1" uploadfolder="test">

...

<SOURCES>
  <FOLDER name="test"/>
  <FOLDER name="test2"/>
</SOURCES>

</CMS_INPUT_PICTURE>
```

The limitation for the **"DOM Editor" input component** is achieved via the link configuration for internal links:

```
<CMS_LINK_CONFIG name="internalLink">
          <CMS_PARAM name="mediaref" value="folder:mediafolder"/>
          <CMS_PARAM name="sitestoreref" value="showmediastore"/>
</CMS_LINK_CONFIG>
```

In this way the picture selection can be limited to folders, which also exist in the package or have been structured for a package (see Chapter 9.1.2 page 104).

[1] see FirstSpirit Online Documentation ./vorlagenentwicklung/formular/cmsinput/cms_input_picture/picture.html

With this limitation it should be noted that the folder names can change due to the name extension and therefore have to be subsequently adjusted manually.

9.1.4 Limiting the template selection

Dependencies on templates are not resolved automatically. If input components are used in a package, which reference templates, these dependencies have to be resolved manually.

The **"Contentarealist" input component** is used to integrate section templates within the Content or Page Store. If this input component is to be used within the Package Pool, the creator of a package must ensure that they integrate all referenced templates into the package or create an independent template package with the referenced section templates. Here too, the template selection should be limited to increase clarity and to avoid user errors.

For the "Contentarealist" input component, this limitation of the template selection is achieved with the help of the `<TEMPLATE>` and `<SOURCES>` tags within the section or page template[2]. With the `<SOURCES>` tag it is possible to limit the selection or display to defined elements. It is a positive list, i.e. only the given elements are allowed. To allow a template, a `TEMPLATE` tag with the `name` parameter and a valid template reference name must be given. In the environment of the Package Pool, the limitation must be made using the reference names. To do this, the unique name of each section template must be given in a separate `TEMPLATE` tag.

```
<CMS_INPUT_CONTENTAREALIST name="cal" ...>

  <LANGINFOS>

    <LANGINFO lang="*" label="TEXT" description="TEXT"/>

  </LANGINFOS>

  <SOURCES>

    <TEMPLATE name="text_picture@t1"/>
    <TEMPLATE name="downloads@t2"/>
```

[2] see FirstSpirit Online Documentation
../vorlagenentwicklung/formular/cmsinput/cms_input_contentarealist/contentarealist.html

```
    </SOURCES>

</CMS_INPUT_CONTENTAREALIST>
```

As can be clearly seen in the example, in this case the namespace extension, which is created by the integration in a package, must also be taken into account. These changes must be made manually. In case of frequent changes, a script can be created, which automates this process.

In FirstSpirit Version 4.1 and higher, the namespace extension can be disabled; in this case the reference names do not change (Chapter 4.1.6 page 34).

9.1.5 Avoiding language-dependent structures in templates

In general, the multilingualism of templates is not supported by the Package Pool. As long as the packages from the master projects and the subscribing target projects contain uniform languages, such language-dependent structures do not pose any problems. Multilingualism in templates always leads to problems, if a language used in the master project does not occur in the master project and was therefore also not implemented in the templates. If templates are to be exchanged via the Package Pool in such a project environment, it is imperative to ensure that multilingualism is not implemented in the templates. A precise explanation is given in Chapter 10.2.4 page 120.

9.1.6 Automatic conversion in the Page Store

When an existing project is transformed into a package master project, the namespace extension causes the reference names to change (if namespace extension is not disabled, see Chapter 4.1.6 page 34). Reference names with namespace extension must also be changed in all places in which they are referenced in the project. In the Page Store, these references to package content are adjusted automatically.

For example, if a link to an object from the Site Store is stored within a page, this reference:

```
<CMS_LINK language="DE" linktemplate="Interner_Link.standard"
sitestoreref="pageref:thisPage" text="Dieser Verweis"
type="Interner Link"/>
```

is automatically adjusted to the namespace extension when the "thisPage" page is

added to a content package:

```
<CMS_LINK language="DE" linktemplate="Interner_Link.standard"
sitestoreref="pageref:thisPage@package" text="Dieser Verweis"
type="Interner Link"/>
```

9.1.7 Manual conversion of templates

The behavior of the automatic conversion of references described in Chapter 9.1.6 does not exist in the Template Store. These must be manually adjusted to the new package namespace extensions.

For example, if a page template (here: "onlycontent") is to be transferred into a package, which references a link template (here: "webeditincludejs"), the references within the template are automatically added to the template package:

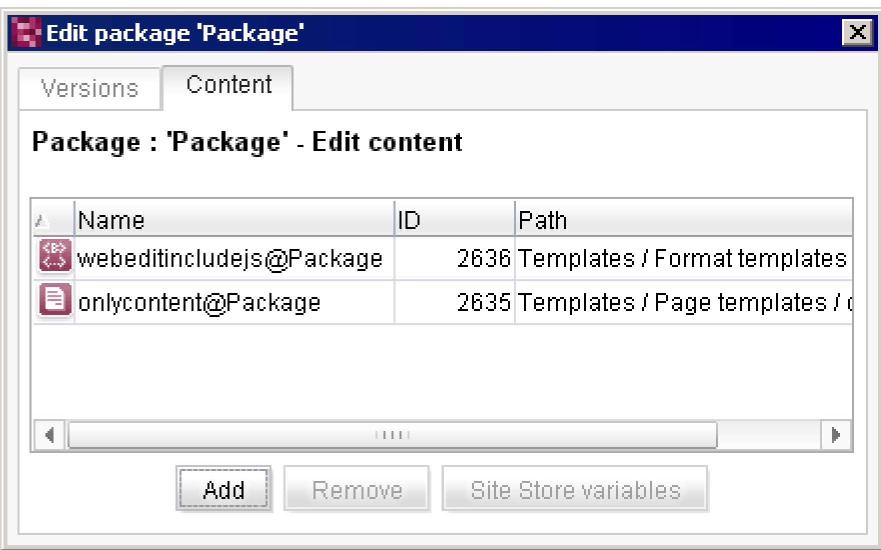


Figure 9-2: Package content on adding a page template with references

However, the references within the template are not adjusted automatically. The "onlycontent" page template therefore also references:

```
$CMS_RENDER(template:"webeditincludejs")$
```

The references within templates therefore have to be manually adjusted by the package developer:

```
$CMS_RENDER(template:"webeditincludejs@Package")$
```

The adjustment must be made for all uses of the link template in the master project. The uses in the project can best be found via the reference graph (see Chapter 9.1.1 page 102). For the example given, three references in three different page templates

have to be manually revised in the master project:

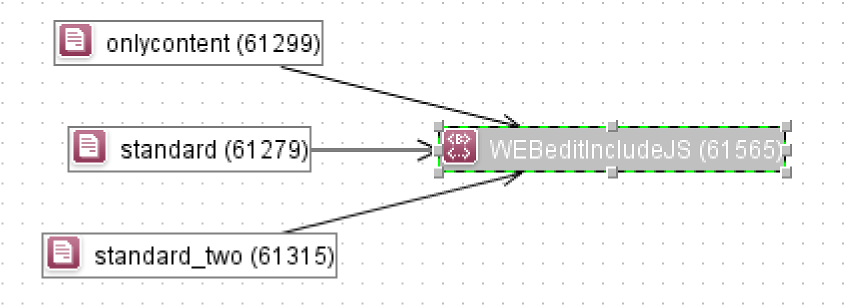


Figure 9-3: Dependencies of a format template

References in output channels: In templates, all references within the output channels, which are given via the instruction $CMS_REF(...)$ or $CMS_RENDER(...)$, have to be edited manually. This concerns the following object types:

- Media (media:...)
- Page references (pageref:...)
- Scripts (script:...)
- Templates (template:...)

Example:

```
src="$CMS_REF(media:"logo",abs:3)$"
```

has to be adjusted manually after adding the "logo" medium to the "package" package:

```
src="$CMS_REF(media:"logo@package",abs:3)$"
```

References in the form area: Within the form area, references also have to be revised manually. For example, if a format template, which is referenced within a DOM input component, is added to a package, the reference in the form area has to be adjusted manually to the new reference name:

```
    <CMS_INPUT_DOM name="st_text" rows="8">
        <FORMATS>
          <TEMPLATE name="format@package"/>
        </FORMATS>
```

In default format templates the namespace extension must be considered critically. If references to default format templates are changed, e.g. "b@package", they are also no longer recognized within the input component if the `<TEMPLATE name="b@package"/>` template is adjusted. For example, the assignment to the

corresponding buttons in the DOM Editor (here: "Bold") is lost. Errors can occur, not only in the master project but also in the target project.

In FirstSpirit Version 4.1 and higher advanced configuration options are available for packages. The namespace extension, which to date was assigned for all project content, can now be disabled by the template developer for all or for only certain object types (see Chapter 4.1.6 page 34). At the same time, the conflict handling on importing the content into a target project can also be adjusted (see Chapter 4.1.7 page 39).

9.1.8 Manual conversion in the Content Store

As in the Template Store, references are not converted automatically in the Content Store. This means, within the Content Store, all references to package content have to be adjusted manually.

For example, a link to an object from the Site Store is stored in an input component in the Content Store, this reference:

```
<CMS_LINK language="DE" linktemplate="Interner_Link.standard"
sitestoreref="pageref:thisPage" text="Dieser Verweis"
type="Interner Link"/>
```

is <u>not</u> automatically adjusted to the namespace extension when the "thisPage" page is added to a content package. The namespace extensions must be manually adjusted by the template developer (see example from Chapter 9.1.6).

In FirstSpirit Version 4.1 and higher advanced configuration options are available for packages. The namespace extension, which to date was assigned for all project content, can now be disabled by the template developer for all or for only certain object types (see Chapter 4.1.6 page 34). At the same time, the conflict handling on importing the content into a target project can also be adjusted (see Chapter 4.1.7 page 39).

If the steps to date have been completed successfully, all requirements are fulfilled for transforming the existing project into a package master project. In the next step, the first package can be created in the new master project (Chapter 4.1.1 page 19).

9.1.9 Checking the functionality in a test project

Creating and importing packages is a complex task. Before the importing of packages is used in a productive environment, the function should therefore be tested first in a test project.

After the first package has been created in the master project (Chapter 4.1.1 page 19), the package properties configured (Chapter 4.2.2 page 43), content added to the package (Chapter 4.2.7 page 49) and finally an initial package version has been generated (Chapter 4.2.4 page 44), it is necessary to first check in the master project whether the project still works correctly.

Whenever objects are implicitly or explicitly added to a package, extensive restructuring take effect, for example, due to the name extension (Chapter 4.1.6 page 34). If the name of a media object changes, the reference of the media file also has to be adjusted in all pages, sections, templates, etc. For content packages, this restructuring is in the project is adjusted automatically via the reference graph (Chapter 9.1.1 page 102). Nevertheless, in individual cases, it is possible that references cannot be resolved automatically by the system or the manual adjustment of the templates is faulty (see Chapter 9.1.7 page 109). In this case the master project no longer works as intended. For example, if a media file can no longer be referenced following the name extension, errors occur in the display of the page.

If, after the package has been created, errors occur during the generation in the master project, the master project must be repaired first, e.g. by changing reference names. If the master project functions without errors, the project can be imported into an "empty" target project for the first time. Then, in the target project, a check is made see whether the import was performed properly and completely or whether required templates or referenced objects are possibly missing in the package. If this is the case, these objects must be added to the package and a new package version created and imported.

Only after this initial test should the master project provide packages for the actual target projects. Event them extensive tests should still be performed on each package version (publication groups: Chapter 2.2 page 11).

9.2 For the same types of projects

If several projects are to share the same content, it is useful to set up a preconfigured project for the roll-out (here: distribution to several target projects of the same type). A default project structure and all the necessary subscriptions can

then be configured once, centrally, in this roll-out project. The project can then be exported and is available as the basic project for all target projects (e.g. offices of a company). When the project is imported, all the necessary subscriptions are created directly in the project at the same time. Use of a roll-out project is used for the case in which all the corporation's offices want to maintain their own independent website, but want to use the templates for designing the pages and the overall, corporation-wide uniform corporate presentation via centrally managed package content.

9.3 Export / Import

Exporting and importing via the server and project configuration is also possible for package master projects and subscribing projects. However, these functions affect the existing package and subscription structure.

9.3.1 Master package projects

> *If a package master project is exported and is then imported again, all package information to date is lost. After importing the project, the symbols behind the object names continue to be displayed in the project tree, and the name extensions are also retained (if they were not disabled). But the package information (as shown in Figure 4-17: Dialog box – Edit package) is no longer available. Packages are no longer displayed in the package overview. The project is therefore no longer a master project.*

The existing, original master project should therefore never be deleted. In this case, not only the package information would be lost, but also the subscriptions in the target projects.

The only way to restore the package information and therefore the master project is to perform a file system backup.

If subscriptions to content of other projects exist in the package master project, they are also retained after the import, but have to be updated manually (see following Chapter 9.3.2).

9.3.2 Subscribing projects

If a subscribing project is exported and then imported again, the content subscribed from other projects is retained and continue to be displayed with a blue color coding behind the object name in the project tree. The subscriptions that existed before the

import are all set to "not up-to-date" state and are assigned an orange color coding, even if no version change has previously taken place in the master project (compare Figure 5-8: Dialog box – Edit subscriptions).

> ! *After the target project is imported the subscriptions must be updated manually.*

10 PackagePool for developers

10.1 Individualizing the package content in the target projects

The Package Pool can be used to import the content of a master project into different projects. In many cases, however, this content is to be displayed differently in the individual target projects. Intervention via structure variables or directly via the templates is possible.

10.1.1 Layout changes via structure variables

The Package Pool supports working with structure variables, therefore, this option is the easiest and best way to intervene in the layout. For example, structure variables can be used to implement color assignments for the menu levels, in which each menu level is displayed with a different background color. The values for the background colors are saved in structure variables and are referenced and evaluated within a template in the project. The structure variables from the master project can be integrated in a package for each node from the Site Store (see Chapter 4.2.8 page 53) and would therefore uniformly copy the color values into all target projects. In the target projects, the values of the structure variables can be easily replaced by the required color values. The next time the subscriptions are updated, these values should of course not be overwritten again. The structure variables can be defined in the subscriptions as not "overwritable" (see Chapter 5.1.5 page 66), so that the values of the structure variables are retained in the target project.

10.1.2 Layout changes via templates

Another option for subsequently changing the layout in the target projects, is direct modifications in the templates. In this case, the package content must not be write protected, this means it must be set as "changeable", not only in the package version but also in the subscription (in the master project: Chapter 4.1.4 page 27, in the target project / subscription: Chapter 5.1.2 page 61).

If templates from a template package are changed in the target projects, this can lead to problems. On the one hand, the project-specific changes must be retraced again after each update of the subscription, on the other hand, conflicts can occur on updating with a new package version, as changes in the master project cannot necessarily also be connected with changed states in the target project. One solution

to these problems is appropriate conflict handling, which can be configured in the subscription (see Chapter 5.1.2 page 61). Here, under "Conflict handling", the "Copy" option must be selected, with which a copy of the changed template is created in the target project. The developer must now revise this copy manually in the target project. The changes in the layout therefore have to be manually retraced in the new template. If the old template is retained here, this can cause the project to no longer function properly.

> *Changes to templates in the target projects should only be made in exceptional cases! The reliable and secure way to individualize the content in the target projects is to make adjustments via structure variables.*

10.2 Multilingualism support

As the implementation of FirstSpirit has been very consistently designed for multilingual projects, these are also supported in the Project Pool. However, the different languages do not necessarily have to be maintained in the master project; the translations into the respective national languages can also be made in the individual offices. Here a differentiation is made between projects with homogeneous language structure and projects with heterogeneous language structure.

10.2.1 Page content

10.2.1.1 For projects with homogenous language structure

In the case of a homogeneous language structure, the package supports the union of all languages used in the subscribing projects.

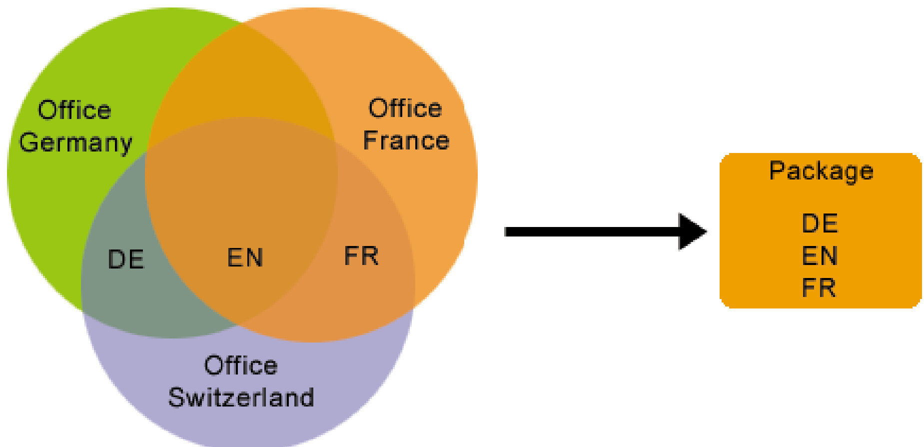


Figure 10-1: Packages with homogeneous language structure

For example:

- German office: DE, EN
- French office: FR, EN
- Swiss office: DE, EN, FR

The package with **homogeneous language structure** contains all three languages. Importing into the target projects is therefore uncomplicated, as each language required in the project is also included in the package. If a package contains more languages than are used in a target project, the surplus language is simply ignored in the target project. In the example given above, the project of the Swiss office office is an ideal candidate for the role of master project.

10.2.1.2 For projects with heterogeneous language structure

In the case of a **heterogeneous language structure,** not all the languages used in the subscribing projects are also contained in the package.

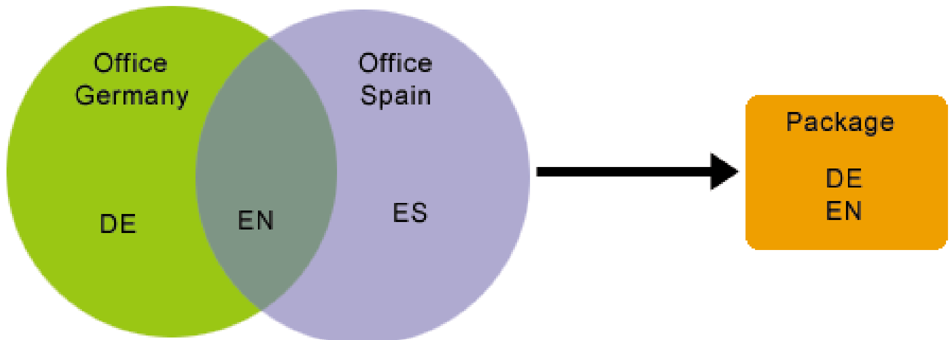


Figure 10-2: Packages with heterogeneous language structure

For example:

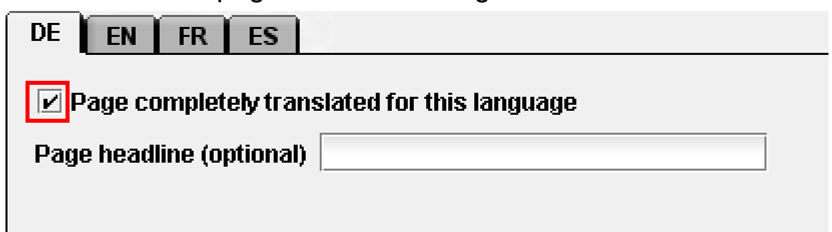
- German office: DE, EN
- Spanish office: ES, EN

The package only contains German (DE) and English (EN). This means, if a package is imported into the Spanish target project (ES), the Spanish language content must be translated separately. If this content is to be translated for the target project, this can be implemented via a workflow, which is started directly when the package is imported.

The following settings must be made for this:

1) Firstly, the project settings must be configured in the target project with the untranslated language. In the **Server and Project Configuration,** the "Use master language" option is selected in the project properties, under the item Substitutions for "**Language substitution**". In the case of the master language, it must be a language contained in the package, for example, English. If objects are now imported into the project, which do not exist in the matching language, only the English language objects are imported, which then have to be translated.

2) The actual translation can be started via a workflow following the initial importing. A "Translate new page" workflow can be used, e.g. to send the new imported page to a translation firm as an XML export and then the translated result can be imported back into the project. In this case, the "**Page completely translated for this language**" setting is important, which can be found at page level in the Page Store.



   - The checkbox must be *disabled* for all new pages, which are initially imported into the project! The workflow should set this setting for all new pages *before the import*. If the translation has been done, the checkbox is then *re-enabled* for all new pages.

- The checkbox must be *enabled* for untranslated changes to a page already existing in the target project. If the checkbox is disabled, the content is overwritten again with the next import.

10.2.2 Language-dependent media and files

The importing of language-dependent media and files into a project is not yet supported in the Package Pool 4.0.

10.2.3 Menu structures

All menu structures in a package, i.e. menu levels and page references, are copied from a package into the target projects. There is a decisive difference between projects with a homogeneous language structure and projects with a heterogeneous language structure.

10.2.3.1 For projects with homogenous language structure

For projects with a homogeneous language structure, all menu structures contained in the package, including the language-dependent labeling, is copied for each language. If a menu level from the Site Store is integrated in a content package, the page references below it and the corresponding pages from the Page Store are also added to the package. If the corresponding pages from the Page Store are filed in folders, only the referenced pages are copied into the package, not the higher-level folders.

If a package contains more languages than are used in a target project, the surplus language is simply ignored in the target project.

10.2.3.2 For projects with heterogeneous language structure

For projects with heterogeneous language structure, the same problems occur for menu structures as for page content (see Chapter 10.2.1.2 page 117). In the target project, languages are supported for which the menu structures are available in the package, but the respective menu headings are not translated.

In this case, setting language substitution by the master language that exists in the package does not have any effect. When the menu structures of a package (EN only) is imported into a target project (EN and DE), there is *no substitution* of the German menu names. In the case of projects with a heterogeneous language structure, for all languages which exist in the target project, but are not integrated in

the package (here: German), the menu structures may not be displayed either in the navigation menu or in the navigation overview. Therefore, the "Display" setting must be disabled at the folder level. When the menu structures are imported into a target project, this setting is automatically made for the unsupported language for each structure folder contained in the package.

Figure 10-3: Do not display in the navigation menu

After translating the labeling the checkboxes have to be re-enabled manually, to make the navigations visible.

10.2.4 Templates

In general, the multilingualism of templates is not covered by the Package Pool. If templates are to be exchanged via the Package Pool, it is imperative to ensure that multilingualism is not implemented in the templates. Multilingualism always leads to problems if a language used in the project was not implemented in the templates, i.e. in target projects with non-heterogeneous language structure.

In order to circumvent multilingualism in templates, for example, all **labeling** can be written in English. The situation is more difficult with templates, which have **language-dependent return values**. For example, with FirstSpirit it is possible to implement a language-dependent combobox, whose return values are also language-dependent.

Language	Label	Displayed value	Returned value
DE:	Farbe :	Rot	Rot
		Blau	Blau
EN :	Color:	Red	Red
		Blue	Blue
ES:	?	?	?

If a language is now added, which is not contained in the package, the form must be extended to include this new language and must be adjusted in all offices. If the Package Pool is to be used in projects with heterogeneous language structures, such template changes should be avoided. This is made possible by the two methods explained in the following.

10.2.4.1 Via joint database access

One option for centrally maintaining language-dependent return values is to use a translation table in the Content Store.

Unlike the usual procedure for the maintenance of multi-lingual content in the Content Store, here all languages are maintained via their own input fields. To do this, a column must be created in the content schema of the master project for each individual target project language and an input component assigned to this column. The labeling of the individual input components is only planned in the master language, in most cases, "English". The language-dependent return values can now be maintained centrally in the master project. All target projects can (read) access this language-dependent content with read access via joint database access (see Chapter 11, page 129).

Assuming the preceding example of the combobox, the master project initially has two input components of the type "Text" for the languages DE and EN. In the table, the language-dependent display value for each language contained in the target projects, e.g. "Rot", is assigned a language-independent return value, e.g. "1". Only the language-independent return value "1" is now stored in the template. The language-dependent assignment is then made for each language for each language using the translation table in the Content Store:

Example: Return value in the template "1" and key "DE" = return value "Rot"

	DE	EN
1	Rot	Red
2	Blau	Blue

If a new language is added, for example, due to a new, Spanish office, the table schema in the master project must be extended to include a column for ES and another input component of the type "Text". The table then looks like this:

	DE	EN	ES
1	Rot	Red	NULL
2	Blau	Blue	NULL

The language-dependent return values for ES can now be added in the master project. No more changes have to be made in the templates.

	DE	EN	ES
1	Rot	Red	Rojo
2	Blau	Blue	Azul

The master language of the target project must be available in the package.

A joint database layer must be defined for all target projects in the project settings in the server and project configuration. In addition, the "No schema sync" and "Write protected" checkboxes must be enabled for the database layer (see Chapter 11.1).

Substitution of the labeling is in turn only possible via a template change.

```
<CMS_LIST lang="ES">
<CMS_LIST_ENTRY label="rojo" selected="0">1</CMS_LIST_ENTRY>
<CMS_LIST_ENTRY label="azul" selected="0">2</CMS_LIST_ENTRY>
</CMS_LIST>
```

10.2.4.2 Via structure variables

Another option for maintaining language-dependent return values in the target projects is to use language-dependent structure variables. In the combobox example, the **language-dependent return values** could be saved in **structure variables**. The value of the required variable is then referenced in the template and is evaluated in the respective project. The structure variables are integrated in the package in the master project and are imported into the target project (see Chapter 4.2.8 page 53). There the values of the structure variables can then be translated from the master language into the required target project language. The structure variables in the subscriptions must be defined as being not "overwritable" so that the language-dependent values are not overwritten again with the next update of the subscriptions. (see Chapter 5.1.5 Page 66),

10.2.4.3 Local differences in the same language

Conflicts can occur when templates are imported, if the same language is used not only in the package but also in the target project. While different countries can use a common language, for example English, there are nevertheless a range of aspects in the countries, which can differ. A prominent example is local formatting differences, for example, different **date** or **currency formats** in countries which otherwise have the same language.

Example:

- Date in Germany: Dienstag 14.08.2001 16:47:48
- Date in Switzerland: Dienstag 2001-08-14 16:47:58

When a package from a German master project is imported into a "Swiss" target project, only the same language "DE" is recognized. However, country-specific formatting is not taken into account.

These problems can be circumvented by introducing a "new" language, which takes into account such local differences, in the example, the new language "CH" was introduced in the target project.

10.3 Using workflows and events

Predefined events can be assigned to workflows within the Package Pool. The assigned workflows are then run if the event occurs during or after the updating of a subscription in the target project (see Chapter 5.1.4 page 65).

One possible use is the release of all objected imported via a subscription using a workflow. As, in principle, a workflow can only be started on one object and not on several objects simultaneously, a script is required, in order to determine all the affected nodes (see Chapter 10.3.1 page 124).

> **!** *In order for both the workflow and the script to be run in the target project, both, the workflow and the script, must also be available in the target project.*

10.3.1 Determining the affected nodes

Within a script in a workflow a user is in the WorkflowScriptContext.

Firstly, the current session is required. This is obtained with

```
m_session = context.getSession();
```

The ImportInfo object is then got from the session:

```
m_importInfo = m_session.get("importInfo");
```

Finally, the UserService is required and the ImportInfo Object is initialized:

```
m_userService = context.getUserService();
m_importInfo.setUserService(m_userService);
```

The initialized ImportInfo object can now be used to determined the quantity

- of new `(getNewNodeCount())`,
- of changed `(getUpdatedNodeCount())`,
- of deleted `(getRemovedNodeCount())` and
- nodes on which a conflict has occurred `(getConflictNodeCount())`

The determined quantity is required in order, with the help of a loop, to iterate across all nodes and to return index-related nodes.

```
NewNode = m_importInfo.getNewNode(index);
```

For example, if the script is to return the first new node, the call looks like this:

```
firstNewNode = m_importInfo.getNewNode(0);
```

The Access API can be used to run other operations on the determined node.

For the complete syntax of import info, please refer to the API documentation.

After performing all operations, the workflow must be switched by the script. This is done using the method `doTransition`:

```
context.doTransition(NAME_OF_THE_TRANSITION);
```

10.3.2 Exemplary workflow for the release

An exemplary workflow for the release of imported objects is shown in Figure 10-4: Release workflow.

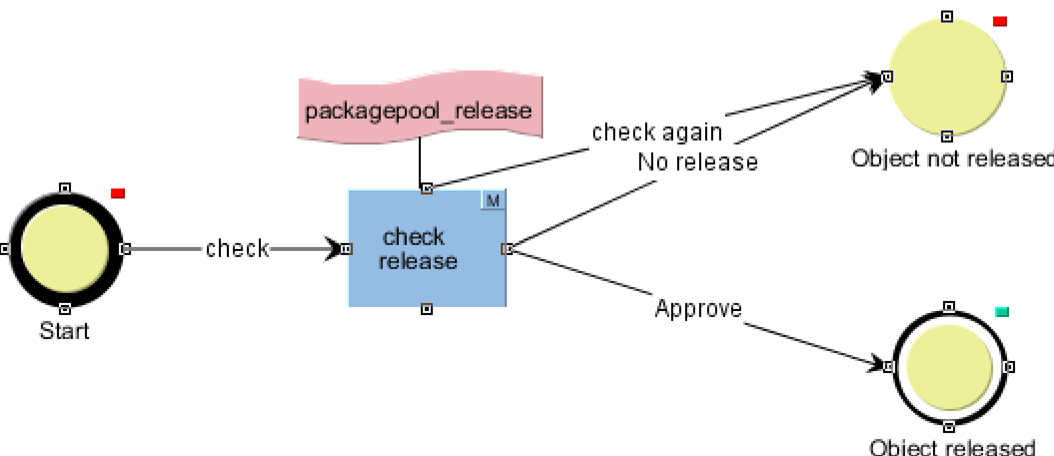


Figure 10-4: Release workflow

To use the release via a workflow in the release target project, the release via a workflow must be set in the subscription (see also Chapter 5.1.2 page 61):

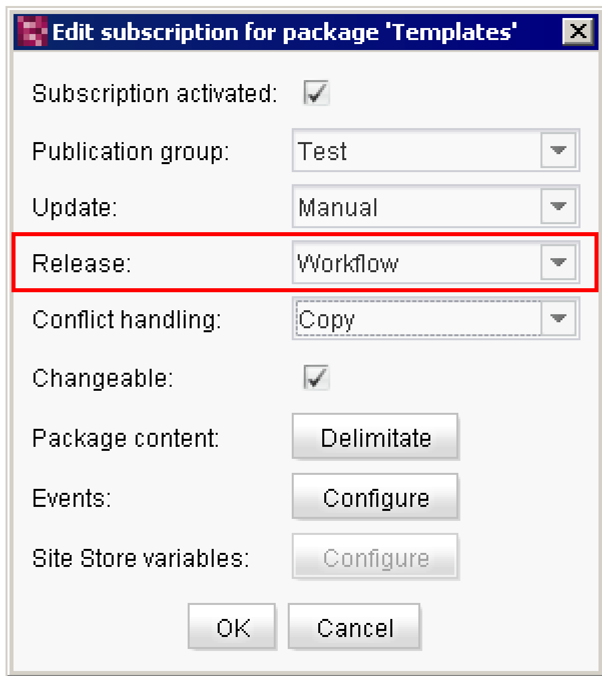


Figure 10-5: Release setting in the subscription

In addition, under Events: **Configure** the "Release" event of the workflow shown under Figure 10-4: Release workflow must be given (see also Chapter 5.1.4 page 65):

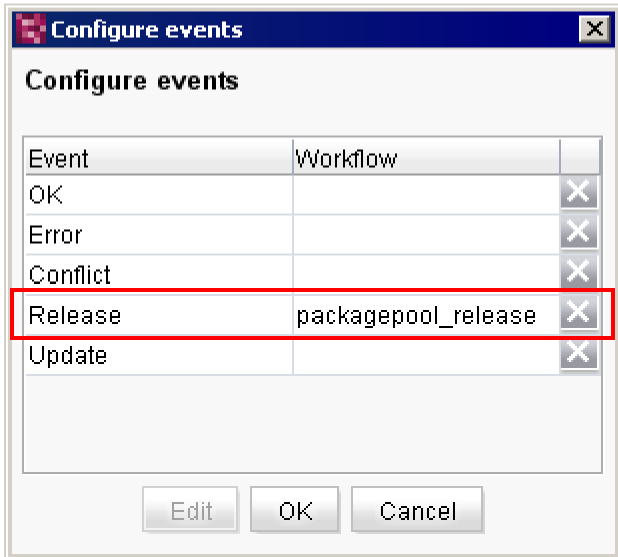


Figure 10-6: Configuring events

If release via a workflow is set in the release target project, this is started as a context-free workflow as soon as new or updated nodes exist in the project. This means, the release is not given context-related on an object in the project tree, but

context-free via the task list.

Figure 10-7: Checking the release task list

If the editor switches the workflow to the next step with "Check", the "packagePoolRelease" script determines the quantity of new or changed nodes in the target project. If there is at least one new or changed node, a list dialog opens, in which the changed nodes are displayed:

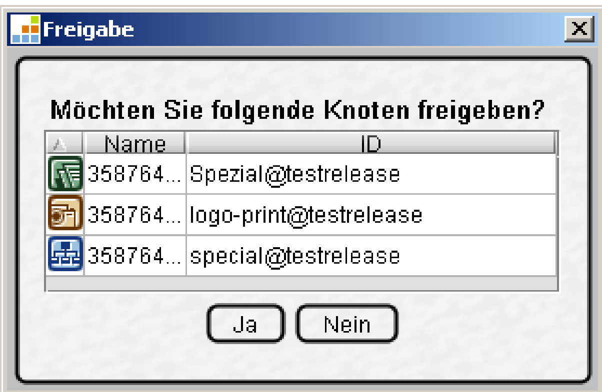


Figure 10-8: Dialog box – Release following nodes

In the list dialog, an entry is double-clicked to display the corresponding object.

Yes: Click the button to release all objects at once.

No: If this button is clicked the objects are not released. However, they can be tested "anew" in the task list (see Figure 10-4: Release workflow).

If the new imported nodes have been released, the Stores should be updated; the new or changed nodes are then shown as being "released" (black lettering).

11 Joint database access

FirstSpirit has efficient mechanisms for linking databases (see FirstSpirit Manual for Administrators). Within the editing environment, the linked databases are called content sources. The data records managed in the content sources can be integrated in the web pages (via the Page and Site Store) and be seamlessly edited in FirstSpirit (via the Content Store), without having to leave the editing environment.

The tables, which are displayed within the Content Store, merely represent views of the database. To do this, a database schema must be created first in the FirstSpirit Template Store (new or generated from an existing database). The project administrator can use a graphic editor to crate the required tables in the selected database in the FirstSpirit JavaClient and to relate them to each other (or to copy them from a linked database). A table template can be created (below the schema node) for each table modeled within the schema. These table templates are used to define the input components via which the editor can subsequently enter data in the corresponding tables and via which input element the editor can accept data of a reference table. The "Mapping" tab can also be used to assign the content entered via the input component to a database table of the physical database.

Depending on the settings of the project administrator for the configured database, the changes within a schema in the JavaClient, for example, inserting a table, can be accepted in the physical database ("Sync") or can be prevented ("no Sync"). The content maintained by the editor within the Content Store, can also be written in the database (also depending on the configuration) or not (write-protected).

For further information, see FirstSpirit Manual for Developers (Basics).

The following content can be integrated in a **template package** and distributed in other FirstSpirit projects via the Package Pool:

- FirstSpirit database schemata
- FirstSpirit table templates
- FirstSpirit database queries

The following content can be integrated in a **content package** and distributed in other FirstSpirit projects via the Package Pool:

- Views of the database (nodes of the Content Store)
- Pages or page references, which have a connection with a content source of the Content Store

The following applies:

Joint access to the database (read access only): In order to exchange database content via the Package Pool, joint access must be configured in the project settings (server and project configuration) *of all projects involved (master projects and target projects)*.
The Package Pool supports the distribution of database views (nodes of the Content Store) in several target projects for joint, read access to the corresponding database content. This means, when the relevant database layer is configured, the "Write-protected" checkbox and "No schema sync" checkbox must be enabled for the target projects. The configuration for joint use is described in the following chapters (see Chapter 11.1 ff., from page 131).

Take into account dependencies: If the database views (nodes of the Content Store) are to be distributed in several target projects via the Package Pool, it is necessary to first ensure that dependent objects, for example, the corresponding database schemata, table templates and queries from the master project are also part of the package (or of a dependent package). The order in which they are added can be decisive. If these dependencies are not taken into account, errors can occur on packing or importing a package. Example: A section template is added to a template package; the section template contains a content list (FirstSpirit input component for the selection and output of data records). An error occurs if the corresponding database schema was not added to the package beforehand.
These dependencies cannot be resolved automatically, like in the content packages (see Chapter 2.1.2), as the effects would be very far-reaching. Staying with the example named above, for example, when the section template was added the schema and all table templates and table queries

below it would automatically become part of the package. However, in general, this will not be the required behavior. The package developer should therefore think in advance about the most effective possible package structure.

11.1 Configuring the target projects (read DB access)

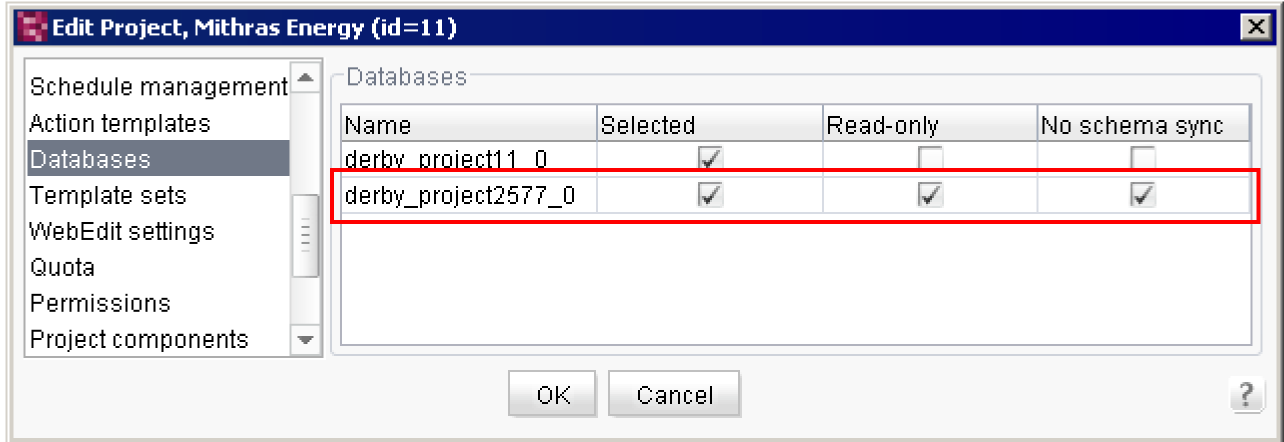


Figure 11-1: Configuring a database layer in the target projects

Firstly, the database layer of the master project must be activated under the "Databases" item, surrounded in red in Figure 11-1. To do this, the relevant checkbox is enabled in the "Selected" column.

In addition, the "No schema sync" and "Write protected" checkboxes must be enabled for this database layer.

Due to the "No schema sync" setting, when a template package is imported new database tables are not created in the database.

By enabling the "Write-protected" checkbox, joint write access to the database from the target projects is prevented. Read access to the database content is then possible in all target projects (views of the database), however, it is not possible to change database content from the target projects.

Further information on the "Multilingualism" use case with regard to joint database access via the Package Pool is given in Chapter 10.2.4.1 page 121.

> *Changes to the database schema must always be made in the master project, as here the "No schema sync" option is not enabled and must be distributed from there to the target projects.*

> *Incompatible schema changes in the master project lead to problems in the target projects, even if the subscription has not yet been updated!*

> *The master and target project should always use the same database schema.*

> *For multilingual projects: When a schema is transferred into a package, both the language structures of the master project and the language structures of the target project must be taken into account in the master project (see Chapter 11.5.1 page 135 and Chapter 11.5.2 page 136).*

11.2 With existing databases

Several adjustments are necessary if joint database access is to be implemented for projects with an existing database or existing data records.

For example, a data record exists in the database, which references an object from the Media Store on the basis of the name. Therefore, on entering a data record, the "test" medium was selected, which is not yet part of a package and is referenced in the data record by "media:test". If joint database access is now to be implemented for several projects, all referenced objects must of course by available in a package. As soon as the "test" medium is added to a package, its name changes to "test@PackageName" (provided the namespace extension is not disabled, see Chapter 4.1.6 page 34). However, the existing reference in the data record continues to reference "media:test", with the result that the medium can no longer be found for this data record. For the medium to be displayed again in the display of the data record, the reference must be adjusted to the new name ("media:test@PackageName") by a script, either manually or automated.

All referenced objects in an existing database must be available in the target projects. Therefore, for joint use of a database, it is advisable to provided all objects to the target projects through a package. The references are then adjusted in the database. In this case, the **Limitation of the media selection** explained in Chapter 9.1.3 page 105 must be implemented for all templates used in the Content Store. These media may only be selected from defined package directories (see 9.1.1 page 102), as this is the only way to ensure that the required media are available in all projects.

If new objects are to be added, for example, media, they are to be added to the master project and made available to the target projects by creating a new package version. This can be achieved by automatic updating via publishing (Chapter 4.2.4 page 44 and Chapter 4.3 page 56) or by manual updating in the target project (Chapter 5.3 page 69).

In addition, when transferring a schema of a multilingual project into a package in the master project, the mapping of the languages of the master project and the target projects should be taken into account (see 11.5 page 135).

11.3 New databases

Unlike use of an existing database, the referential integrity of a new database is of no consequence, as the database does not yet contain any data.

When transferring a schema of a multilingual project into a package, the mapping of the languages of the master project and the target projects must be taken into account in the master project (see 11.5 page 135). The **Limitation of the media selection** explained in Chapter 9.1.3 page 105 for all templates used in the Content Store is also recommended for new databases. These media may only be selected from defined package directories (see 9.1.1 page 102), as this is the only way to ensure that the required media are available in all projects.

11.4 “contentSelect" function

The "contentSelect" function requires particular attention in projects with joint database access. The adjustments within a function in the <CMS_PARAM> tags must be made manually. This applies to all templates of the master project, i.e. including for the templates, which are not integrated in a package. The reason for this lies in the name extension of the jointly used database schema. If the schema is distributed in the target projects via the Package Pool, the schema name changes:

```
<CMS_PARAM name="schema" value="News"/>
```

becomes:

```
<CMS_PARAM name="schema" value="News@MyPaket"/>
```

In this case, all templates of the master project, which use the "contentSelect" function have to be adjusted manually. The templates that are not used in a package must also access the `News@MyPaket` schema with immediate effect.

Advantage: If the templates in the master project have been adjusted, no further changes are necessary in the target projects. They adopt the already updated templates via the Package Pool.

```
<CMS_FUNCTION name="contentSelect" resultname="fr_sc_news">
          <CMS_PARAM name="schema" value="News"/>
          <QUERY entityType="News">
          <ORDER>
            <ORDERCRITERIA attribute="Date" descending="1"/>
          </ORDER>
          </QUERY>
</CMS_FUNCTION>
```

becomes:

```
<CMS_FUNCTION name="contentSelect" resultname="fr_sc_news">
          <CMS_PARAM name="schema" value="News@MyPackage"/>
          <QUERY entityType="News.Overview@MyPackage">
          <ORDER>
             <ORDERCRITERIA attribute="Date" descending="1"/>
          </ORDER>
          </QUERY>
</CMS_FUNCTION>
```

In FirstSpirit Version 4.1 and higher, the namespace extension can be disabled; in this case the schema names do not change (Chapter 4.1.6 page 34).

11.5 Language-dependent content

The data of the individual input components, which are visible in the Content Store, are stored in a database table for joint database access. As the schema should not be changed in the target project, the languages for an input component should be defined in the master project.

Two procedures are available to choose from for mapping of the languages:

1. Implicit modeling of the language dependency
2. Explicit modeling of the language dependency

11.5.1 Implicit modeling of the language dependency

In the case of implicit modeling of the language dependency, all languages of the target projects must be added to the master project languages. This union set of all project languages is then taken into account when a database schema is created.

The languages are added to the server and project configuration in the "Project properties" under the "Languages" item. The "Generate language" option should be disabled, to prevent the languages added to the target projects from being used for the generation of the master project too.

A column for each language must then be created in the database schema and the columns referenced in mappings of the table template.

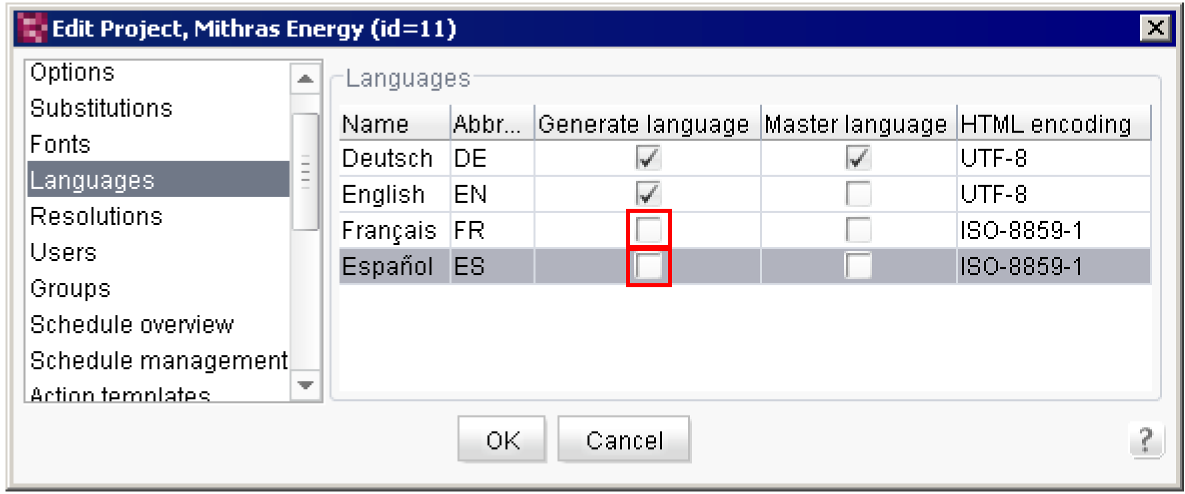


Figure 11-2: Implicit modeling of the language dependency

Example:

If the master project contains German and English, the first target project contains

Spanish and English and the second target project French and English:

1. Spanish and French must be added to the project properties of the master project,
2. Columns for German, English, French and Spanish must be created in the database component for the input component and the input component must be referenced in the mappings.

11.5.2 Explicit modeling of the language dependency

Unlike implicit modeling, in explicit modeling the languages are not mapped via the project property, but solely via the database schema. This means, a column is created in the database schema for each input component of a language. An input component must then be defined for each column in the Form tab of the table template and referenced in the mappings.

11.6 Different database layer in the master and target project

In the productive environment, direct access to a database by the target projects is frequently not wanted. If the target projects are not to be able to directly access a database, but a copy of this database, in most cases the database layer of the master project is managed by FirstSpirit and the copy for the target projects by an export by the database administrator. In this case the master project works on a database layer managed by FirstSpirit, and the target projects on a layer, which has to be manually updated by the database administrator. As a result, the states of the master and target project are frequently asynchronous and errors occur in the target projects.

In an initial import, the "No schema sync" option must be disabled in the project settings (server and project configuration) under the "Databases" item. Following the initial input this option must then be re-enabled (see Chapter 11 page 129).

If the target projects are to work on a copy of the original database, the database schema should be duplicated in the master project. For this database schema, a separate database layer is now assigned for the target projects. If now, only the duplicated database schema is published, the master and target projects always work on one state.

To circumvent these problems, the original database schema should be duplicated in the master project and another layer assigned to the duplicate. On publishing, only the duplicate schema is made available to the target projects.

Advantage: The database is managed through FirstSpirit.